

**FACULTATEA DE INGINERIE ELECTRICĂ ȘI ȘTIINȚA CALCULATOARELOR
DEPARTAMENTUL DE ELECTRONICĂ ȘI CALCULATOARE**

Prof. dr. ing. Gheorghe TOACŞE

**ARHITECTURA ȘI ORGANIZAREA
MICROPROCESOARELOR**

**(NOTE DE CURS PENTRU SPECIALIZĂRILE DE :
– ELECTRONICĂ;
– CALCULATOARE).**

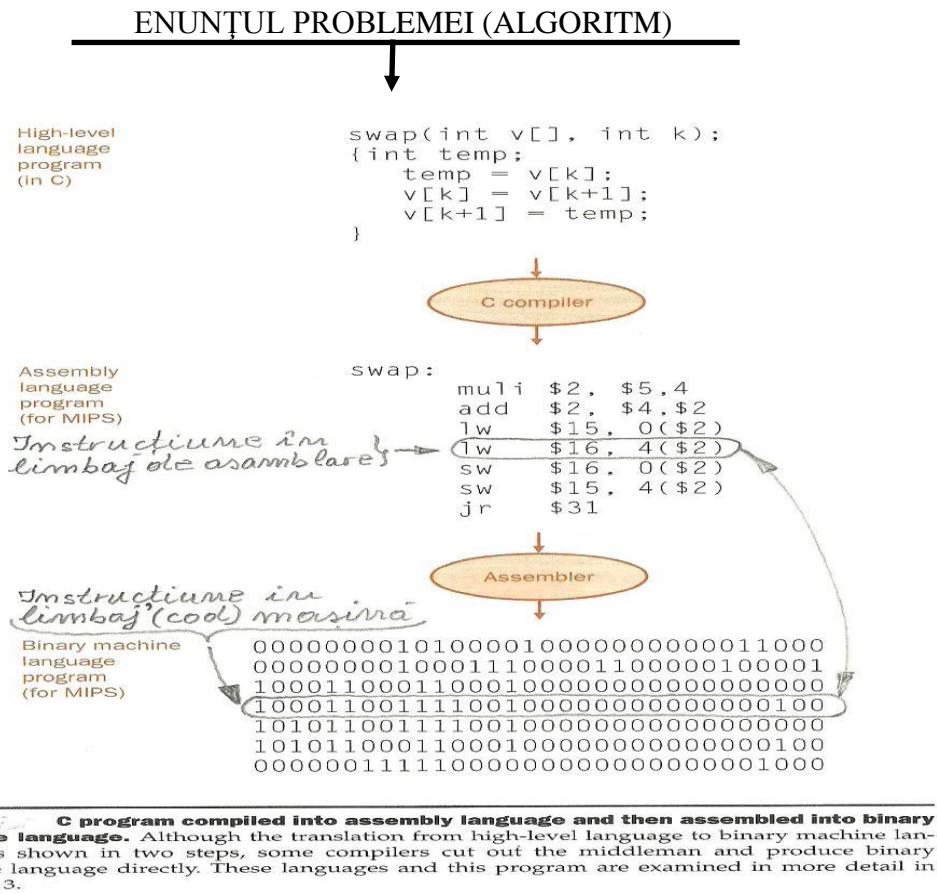
Anul universitar 2012-2013

BRAŞOV–2013

INTRODUCERE

Ce este un calculator?

Un calculator este o mașină programabilă care primește o intrare, stochează și proceseză conform unui program date/informăție generând ieșire într-un format utilizabil.



- Reprezentarea informației în calculator

Orice problemă, ca să fie rezolvată pe calculator, trebuie să fie adusă în format limbaj mașină! (cuvânt binar, siruri de 1 și 0).

Un sir de biți formează un cuvânt, care este caracterizat prin numărul de biți (lungimea cuvântului). Lungimile de cuvint uzuale pentru calculatoare sunt :

		<u>Numărul total de cuvinte diferențe cu această lungime</u>									
Bit	1										
Nibble/Tetradă	3	1011	0								
Byte/octet	7	10110101	0								
Cuvânt de 16 biți (semicuvânt)	15	10100011000110101	0								
Cuvânt de 32 biți	31	24123161158170	0								
Cuvânt de 64 biți (dublu cuvânt)	63	565548474039321									
Cuvânt de 128 biți	127	12011911211110410396									
	95	88878075797164									
	63	565548474039321									
	31	24123161158170									

$$* \quad 2^{10} = 1024 \rightarrow 1K; \quad 2^{20} = 1048576 \rightarrow 1M; \quad 2^{30} = 1073741824 \rightarrow 1G$$

Pentru stocare în procesor sau memorie

- 1 Bit = Binary Digit
- 8 Bits = 1 Byte
- $1024 \text{ Bytes} = 1 \text{ Kilobyte} = 2^{10} \text{ bytes}$
- $1024 \text{ Kilobytes} = 1 \text{ Megabyte} = 2^{20} \text{ bytes}$
- $1024 \text{ Megabytes} = 1 \text{ Gigabyte} = 2^{30} \text{ bytes}$
- $1024 \text{ Gigabytes} = 1 \text{ Terabyte} = 2^{40} \text{ bytes}$
- $1024 \text{ Terabytes} = 1 \text{ Petabyte} = 2^{50} \text{ bytes}$
- $1024 \text{ Petabytes} = 1 \text{ Exabyte} = 2^{60} \text{ bytes}$
- $1024 \text{ Exabytes} = 1 \text{ Zettabyte} = 2^{70} \text{ bytes}$
- $1024 \text{ Zettabytes} = 1 \text{ Yottabyte} = 2^{80} \text{ bytes}$
- $1024 \text{ Yottabytes} = 1 \text{ Brontobyte} = 2^{90} \text{ bytes}$
- $1024 \text{ Brontobytes} = 1 \text{ Geopbyte} = 2^{100} \text{ bytes}$

Pentru stocare în hard disk

- 1 Bit = Binary Digit
- 8 Bits = 1 Byte
- $1000 \text{ Bytes} = 1 \text{ Kilobyte} = 10^3 \text{ bytes}$
- $1000 \text{ Kilobytes} = 1 \text{ Megabyte} = 10^6 \text{ bytes}$
- $1000 \text{ Megabytes} = 1 \text{ Gigabyte} = 10^9 \text{ bytes}$
- $1000 \text{ Gigabytes} = 1 \text{ Terabyte} = 10^{12} \text{ bytes}$
- $1000 \text{ Terabytes} = 1 \text{ Petabyte} = 10^{15} \text{ bytes}$
- $1000 \text{ Petabytes} = 1 \text{ Exabyte} = 10^{18} \text{ bytes}$
- $1000 \text{ Exabytes} = 1 \text{ Zettabyte} = 10^{21} \text{ bytes}$
- $1000 \text{ Zettabytes} = 1 \text{ Yottabyte} = 10^{24} \text{ bytes}$
- $1000 \text{ Yottabytes} = 1 \text{ Brontobyte} = 10^{27} \text{ bytes}$
- $1000 \text{ Brontobytes} = 1 \text{ Geopbyte} = 10^{30} \text{ bytes}$

În calculator un cuvânt binary poate reprezenta o **instrucțiune** sau o **data**:

1. Cuvânt instrucțiune
2. Cuvânt data - adresă, character, dată logică, sir
 - număr: întreg (cu semn sau fără semn) sau zecimal (în virgulă fixă sau virgulă flotantă).

- Termenul de **Microprocessor** (procesor realizat la scară integrată (microelectronică)) acoperă (în general):
 - Microprocesoarele de uz general și cele specializezate/coprocesoare, **μ P**.
 - Microcontrolerelor (care pot fi microcalculatoare, **μ C**)
 - Procesoarele de semnal, **DSP** (Digital Signal Processing)
- Dezvoltarea calculatoarelor nu a fost revoluționară ci una evoluționară. Evoluția în performanțe la μ P s-a bazat pe:
 1. Tehnologia de integrare (legea lui Moore);
 2. Îmbunătățirile arhitectural-structurale.

Începând cu 1985 îmbunătățirile/inovațiile structurale au generat creșteri anuale de performanță mai mari decât creșterile datorate legii lui Moore. Majoritatea îmbunătățirilor architectural-structurale la μ P s-au bazat pe aplicarea unor concepte deja cunoscute, până în anii '70, pentru mainframe și mai ales pentru supercalculatoare (În 1965 Gordon Moore, co-founder la Intel, a emis aserțiunea “ numărul de tranzistoare pe unitatea de suprafață se dublează în fiecare an”, care este referită ca legea lui Moore. Începând cu anii '90 acest ritm de creștere s-a încetinit avânt o dublare cam la 18 luni. Se prezice că această rată de creștere, pentru tehnologia CMOS se continuă cam până în jur de 2018)

Scurt istoric

- 1946 (anunțare publică) – Primul calculator electronic (realizat cu tuburi cu vid, ENIAC- Electronic Numerical Integrator and Computer). A fost realizat de către inginerii John Mauchly și John Presper Eckert la Moore School of the University of Pensilvania, pentru calculul automat al tabelelor de tragere utilizate de artilleria americană. Apoi, conceptualizarea sistematică pentru un nou calculator, ce urma a fi realizat de către University of Pensilvania (U-Pen) și University of Princeton (proiect EDVAC- Electronic Discrete Variable Automatic Computer, proiect care s-a întâmplat), s-a făcut prin conceptul de **arhitectură de mașină cu program stocat în memorie** (stored programme computer). Pentru această nouă mașină s-a elaborat un memo, care a fost sistematizat de către ilustrul matematician John von Neumann (*First Draft of a Report on the EDVAC*; surprinzător, acest material rămâne foarte actual și în prezent!), implicat în proiectul Manhattan (prima bombă nucleară), dar care includea în acel material și ideile elaborate de Mauchly și Eckert în proiectul ENIAC. Acest memo a fost difuzat de către Herman Goldstine, dar fără a înscrie și numele celor doi ingineri! Acest material a influențat toată concepția care a stat la dezvoltarea calculatorului, prin impunerea a ceea ce se numește arhitectura de tip “von Neumann”.
- 1971- Primul microprocessor (4004), **μ P**, anunțat de firma INTEL (înființată în 1967), care a fost ideea lui T. D. Hoff și proiectat în siliciu de Frederico Faggin. Pentru apariția microprocesorului nu s-a făcut o activitate de cercetare planificată, a fost o apariție naturală. De ce naturală? Pentru că tehnologia de integrare, dezvoltată pentru domeniul militar (în anii '60 era război rece!) și după dezastrul programului Apollo (debarcarea unui om pe lună, 1969), a început să fie transferată și în domeniul activităților civile, deci așa a apărut posibilitatea tehnologică de realizare a primului microprocesor. Microprocesorul, prezentat (greșit) la început ca “computer-on-a-chip” nu a avut succes de piață, periclitând existența firmei Intel. Deoarece acest “computer-on-a-chip” necesita pentru funcționare o programare, electroniștii îl evitau să și-l însușească, la fel au procedat și calculatoriștii deoarece acesta se programa într-un limbaj rudimentar (cod mașină sau limbaj de asamblare), la vremea aceea limbajele curente erau Cobol și Fortran. De fapt, corect, μ P era Unitatea Centrală de Procesare, **CPU** (Central Processing Unit), adică procesorul unui calculator, dar realizat la scară micro (integrat). Prezentat, apoi, ca fiind elementul cu care se poate realiza un calculator, dacă se jonctionează cu memoria și cu elementele de intrare-iesire I/O (perifericele) plus softul SO (Sistemul de Operare), μ P a avut imediat succes de piață. Iar pentru realizarea unui calculator pe bază de μ P (mai târziu s-a realizat și computer-on-a-chip) s-au format specialiști care jonctionau pregătirea (întrepătrunsă) atât de hard cât și de soft.

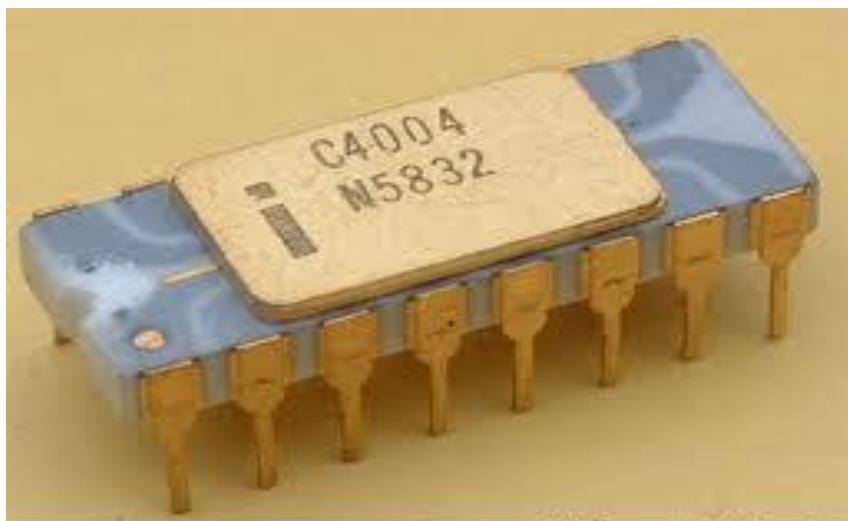
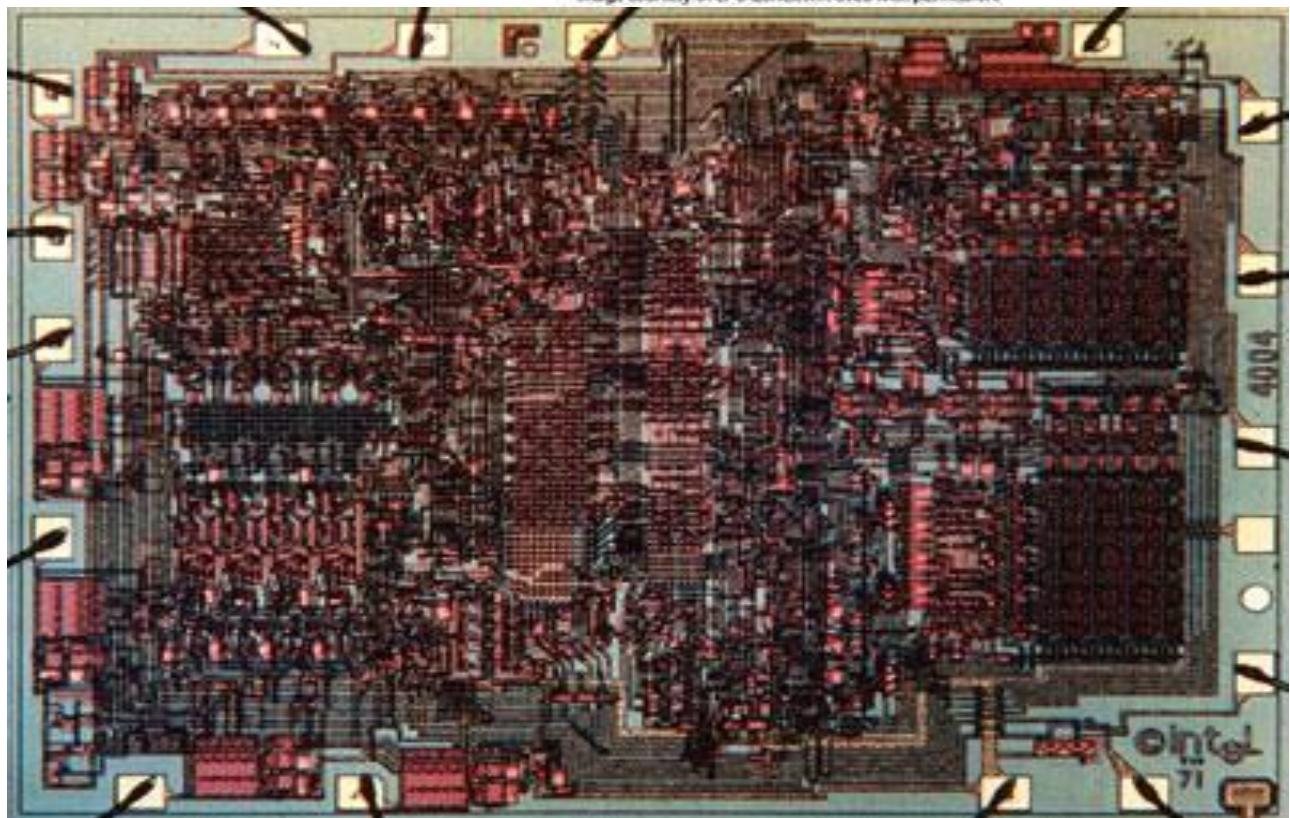


Image courtesy of CPU-Zone.com. Used with permission.



Microfotografia layoutului pentru µP 4004 (primul microprocesor)

- După 4004 la firma Intel a urmat: 8008 (1972); 8080 (1974); 8086 (1978); 80286 (1982); 80386 (1985) Pentium (1993); PentiumII (1995- introduce noua arhitectură referită P6); Pentium III (1999); Pentium 4 (2001); Pentium M (2004). Alături de Intel au început să realizeze µP și alte firme (IBM , Motorola, National Semiconductor, AMD etc) care au conceput arhitecturi fără greșelile de început ale procesoarelor Intel.

Core Architecture (2005 – apare procesorul multicore la Intel), IBM (Power 4 în 2001 și Power 5 în 2005) care se pare a deschis noua direcție de dezvoltare a microprocesoarelor, microprocesoare multiprocesor (multicore) sau CMP (Chip MultiProcessors).

Cresterea performanțelor microprocesoarelor (viteză de calcul, frecvență de ceas) sunt prezentate în diagramele următoare [1]

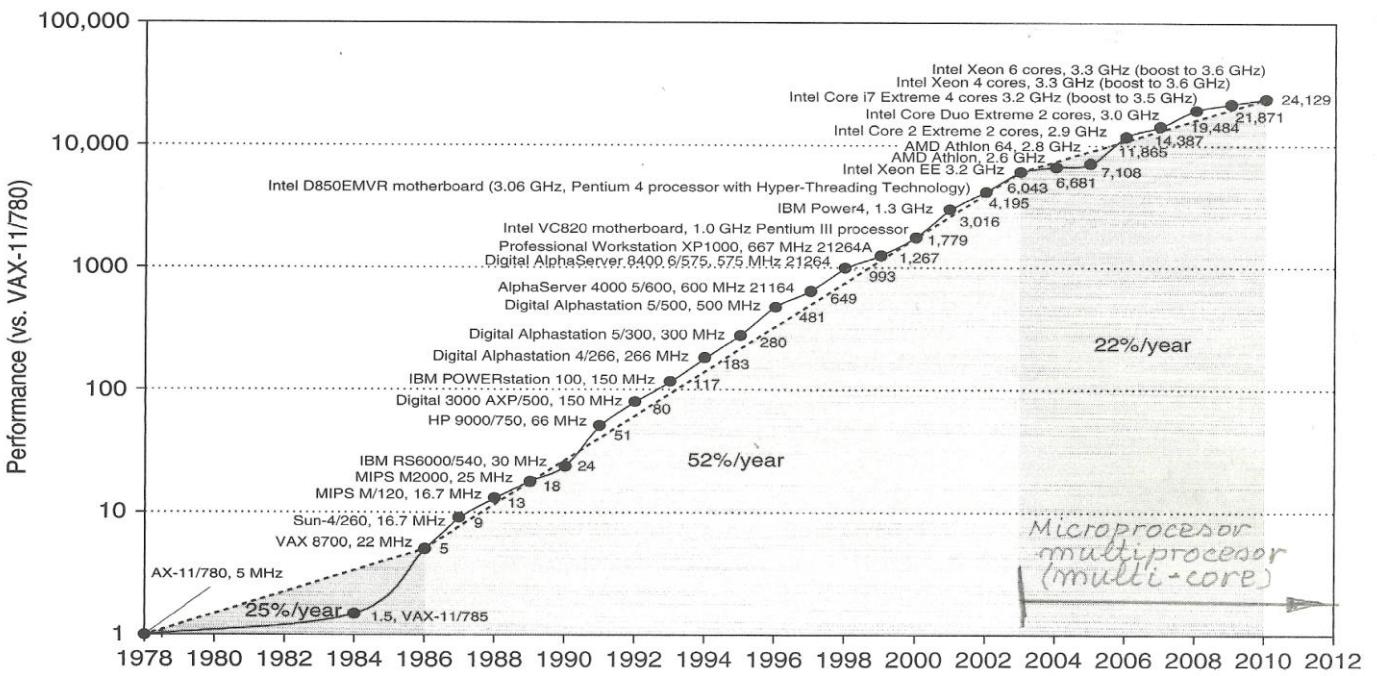


Figure 1.1 Growth in processor performance since the late 1970s. This chart plots performance relative to the VAX 11/780 as measured by the SPEC benchmarks (see Section 1.8). Prior to the mid-1980s, processor performance growth was largely technology driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. By 2003, this growth led to a difference in performance of about a factor of 25 versus if we had continued at the 25% rate. Performance for floating-point-oriented calculations has increased even faster. Since 2003, the limits of power and available instruction-level parallelism have slowed uniprocessor performance, to no more than 22% per year, or about 5 times slower than had we continued at 52% per year. (The fastest SPEC performance since 2007 has had automatic parallelization turned on with increasing number of cores per chip each year, so uniprocessor speed is harder to gauge. These results are limited to single-socket systems to reduce the impact of automatic parallelization.) Figure 1.11 on page 24 shows the improvement in clock rates for these same three eras. Since SPEC has changed over the years, performance of newer machines is estimated by a scaling factor that relates the performance for two different versions of SPEC (e.g., SPEC89, SPEC92, SPEC95, SPEC2000, and SPEC2006).

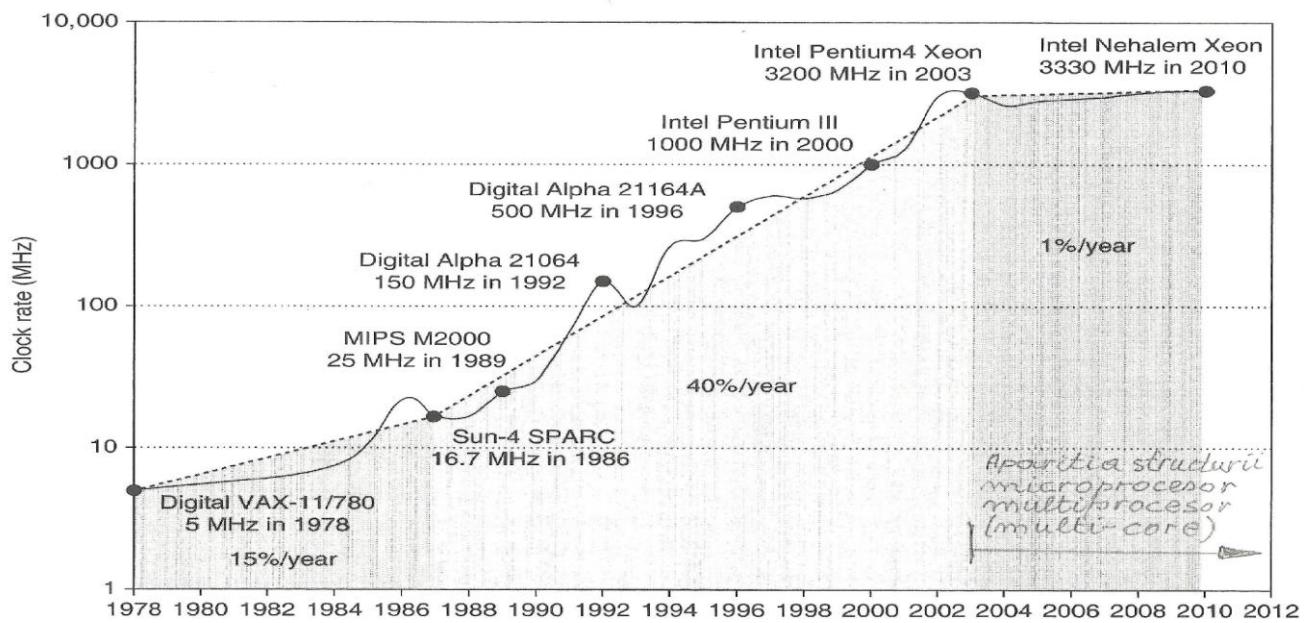


Figure 1.11 Growth in clock rate of microprocessors in Figure 1.1. Between 1978 and 1986, the clock rate improved less than 15% per year while performance improved by 25% per year. During the “renaissance period” of 52% performance improvement per year between 1986 and 2003, clock rates shot up almost 40% per year. Since then, the clock rate has been nearly flat, growing at less than 1% per year, while single processor performance improved at less than 22% per year.

- **Clase (de aplicații) pentru calculatoare:**

Oricare calculator este construit pe baza unui procesor, dar în piața calculatoarelor (computing market) acestea sunt caracterizate în funcție de aplicație, de performanțe și de tehnologia de realizare. În acest sens se pot distinge următoarele cinci clase de calculatoare:

1. *Dispozitive/calculatoare mobile personale, PMD (Personal Mobile Devices)*. Cu această abreviație sunt referite dispozitivele de tip wireless cu o interfață utilizator de tip multimedia, cum sunt: telefoanele celulare, tabletele digitale etc);
2. *Desktop computers*. Calculatorul personal (**PC**), utilizat pentru un singur utilizator; cuprinde întreaga scară de la low-end, cum sunt notebook-urile, până la cele high-end cu o configurație bogată, cum sunt stațiile de lucru (work stations);
3. *Servere*. Modern, prin servere se face referință la ceea ce înainte era reprezentat de: calculatoare mari (mainframe), minicalculatoare și supercalculatoare, uzual accesul acestora este numai prin rețea. Acestea sunt coloana vertebrală a mediului de calcul al unei întreprinderi, tehnologia lor, fundamental, este ca și cea a desktop computers dar sunt direcționate pentru o mare expandabilitate (scalare) și largă capacitate de intrare/ieșire (I/O). Se întâlnesc:
 - servere de date
 - servere de fișier
 - servere de rețea;
4. *Clusters/ Warehouse-Scale Computers (WSC)* (Cluster = ciorchine; Warehous = depozit de mărfuri, magazie mare). Clusterul este o colecție de desktop computer sau de servere, ca și elemente componente (noduri) conectate local printr-o rețea locală și care în totalitate , funcțional, este considerat un singur calculator. Fiecare nod rulează propriu său sistem de operare, comunicare între noduri se realizează printr-un anumit protocol. Clusterele cele mai mari, cuprindând zeci sau sute de mii de servere, sunt referite prin abreviația WSC. WSC sunt direcționate spre aplicații de tip SaaS (Software as a Service) cum sunt: search, social networking, video sharing, online shopping etc.

O sub clasă de WSC sunt **supercalculatoarele**, care sunt realizate spre putere ridicată de calcul (în general în virgulă flotantă) pentru programe mari și cu comunicații intense de date între noduri ce pot necesita tempi de rulare de ordinul săptămânilor. În consecință, supercalculatoarele necesită rețele de comunicare internă de viteză foarte ridicate, dar mai puțină bandă spre Internet cum necesită WSC-urile;
5. *Embedded systems* (sisteme integrate pe bază de calculator). Reprezintă mare masă de aplicații ale calculatorului ca de exemplu: automobile, periferice, jocuri electronice, aparatură electrocasnică, comandă pentru procese de producție etc. În general acestea rulează o singură sau un set restrâns de aplicații (software dezvoltat intern, special pentru aplicația respectivă). De fapt, și PMD sunt tot embedded systems, dar se consideră PMD ca o clasă separată deoarece acestea au multe caracteristici similare cu clasa de desktop computere, ca de exemplu pot rula soft dezvoltat în exterior.

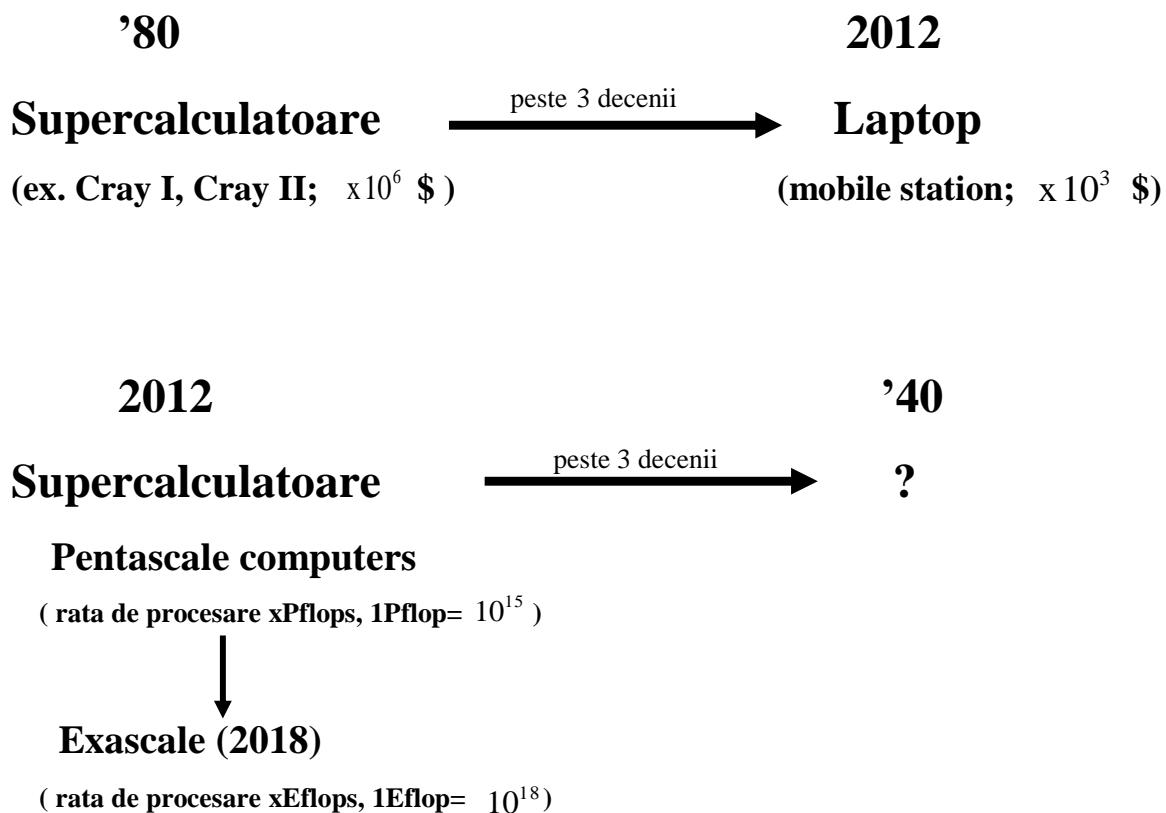
Caracteristicile principale ale acestor clase de calculatoare sunt prezentate în tabelul următor [1]

Feature	Personal mobile device (PMD)	Desktop	Server	Clusters/warehouse-scale computer	Embedded
Price of system	\$100–\$1000	\$300–\$2500	\$5000–\$10,000,000	\$100,000–\$200,000,000	\$10–\$100,000
Price of micro-processor	\$10–\$100	\$50–\$500	\$200–\$2000	\$50–\$250	\$0.01–\$100
Critical system design issues	Cost, energy, media performance, responsiveness	Price-performance, energy, graphics performance	Throughput, availability, scalability, energy	Price-performance, throughput, energy proportionality	Price, energy, application-specific performance

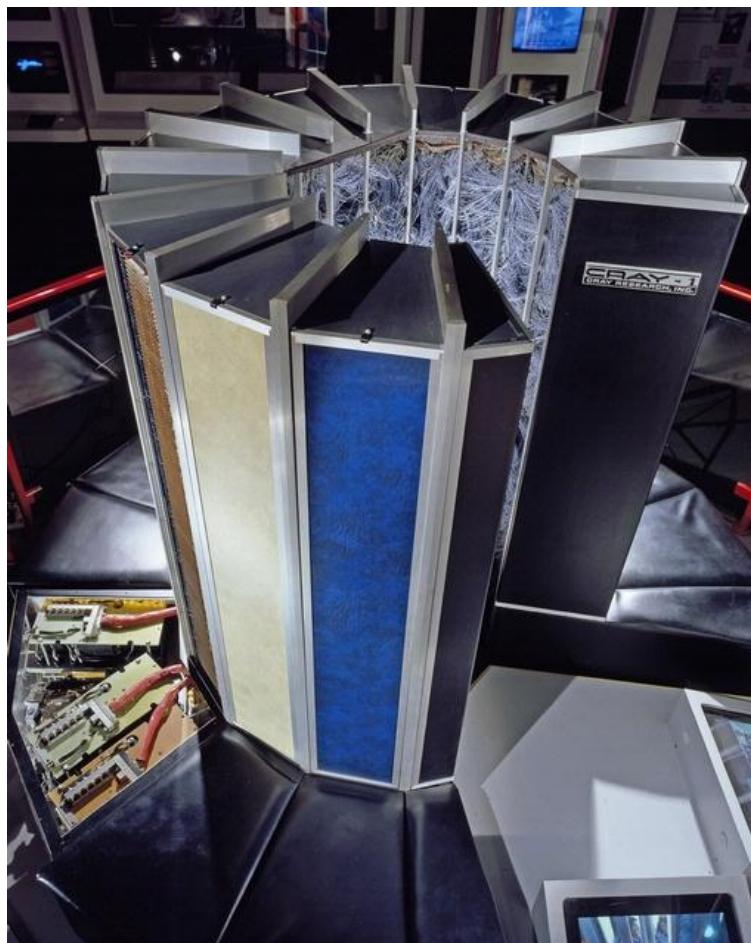
Figure 1.2 A summary of the five mainstream computing classes and their system characteristics. Sales in 2010 included about 1.8 billion PMDs (90% cell phones), 350 million desktop PCs, and 20 million servers. The total number of embedded processors sold was nearly 19 billion. In total, 6.1 billion ARM-technology based chips were shipped in 2010. Note the wide range in system price for servers and embedded systems, which go from USB keys to network routers. For servers, this range arises from the need for very large-scale multiprocessor systems for high-end transaction processing.

- **Evoluția calculatoarelor**

QVO VADIS COMPUTER?



. Cray Supercomputers (începutul anilor '80)



SUPERCALCULATOARELE 2012

Top 500 (www.top500.org/)

Green 500

1. Sequia-BlueGene/Q

Tip procesor : Power BQC; 1,6GHz
 Nr. Procesoare = 1 572 864

Rata maximă = 20, 1327 Pflops
 Rata medie = 16,3248 Pflops
 Memorie = 1 572 864 GB
 Putere = 7,890 MWatt
 Sist. de operare: LINUX
 Producător IBM

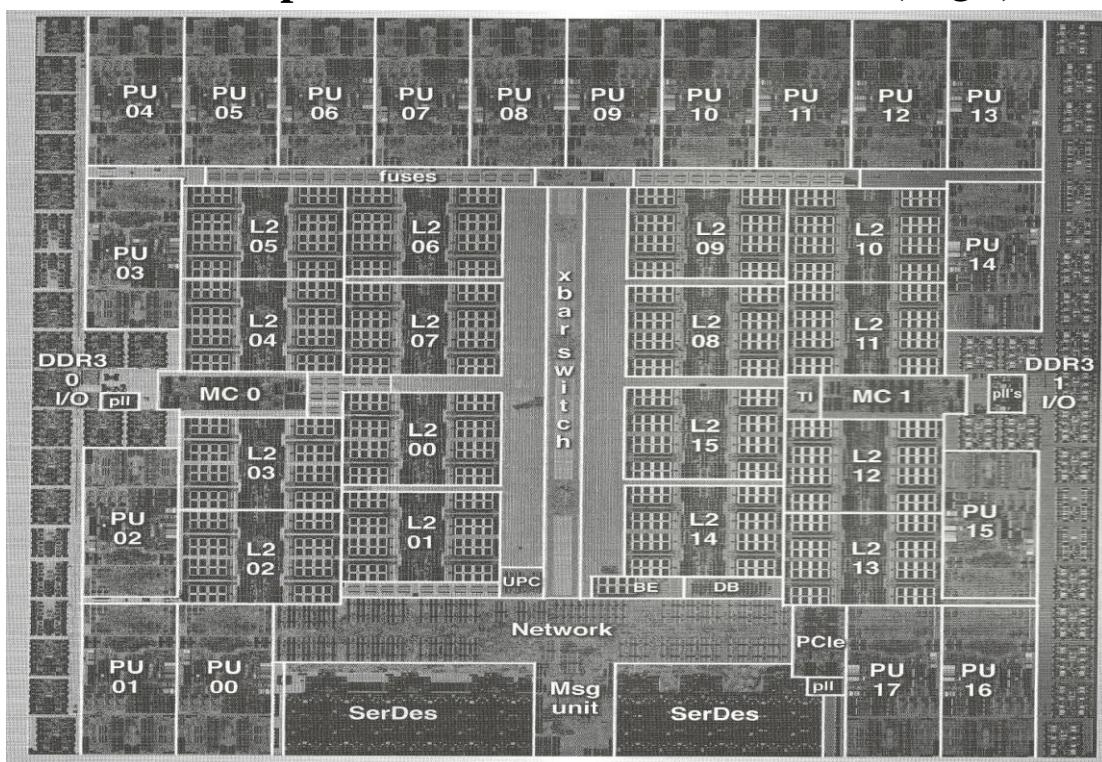
?

2. K computer

Tip procesor : SPARC 64 VIIIfx; 2GHz
 Nr. Procesoare = 705 024
 Rata maximă = 10,51Pflops
 Rata medie = 8,162 Pflops
 Sist. de operare: LINUX
 Putere = 12,659 MWatt
 Producător Fujitsu

Locul 6

Micropresesorul multicore BlueGene (BQC)



Tehnologie 45nm; Arie 18,96 x18,96mm ; Număr tranzistoare= $1,47 \cdot 10^9$

Număr de core = 18; L₁ - cache = 16KB; L₂ - cache = 32 MB

Sequia-BlueGene/Q



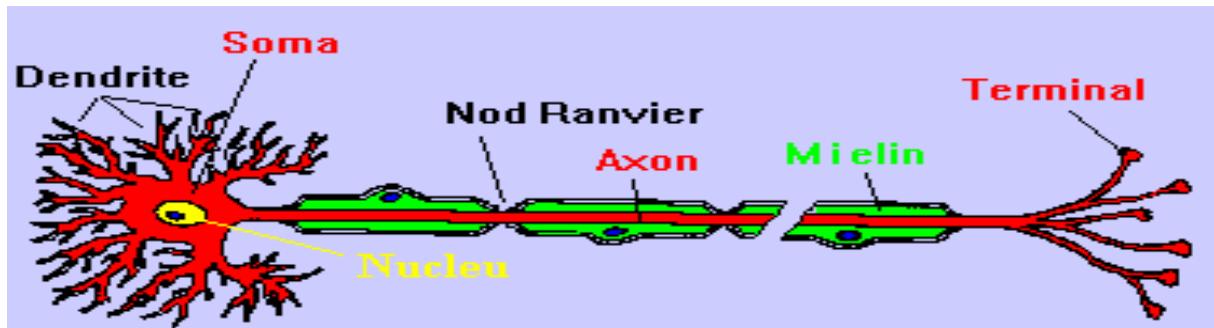
(Photo: Lawrence Livermore National Laboratory)

K computer (Japonia)



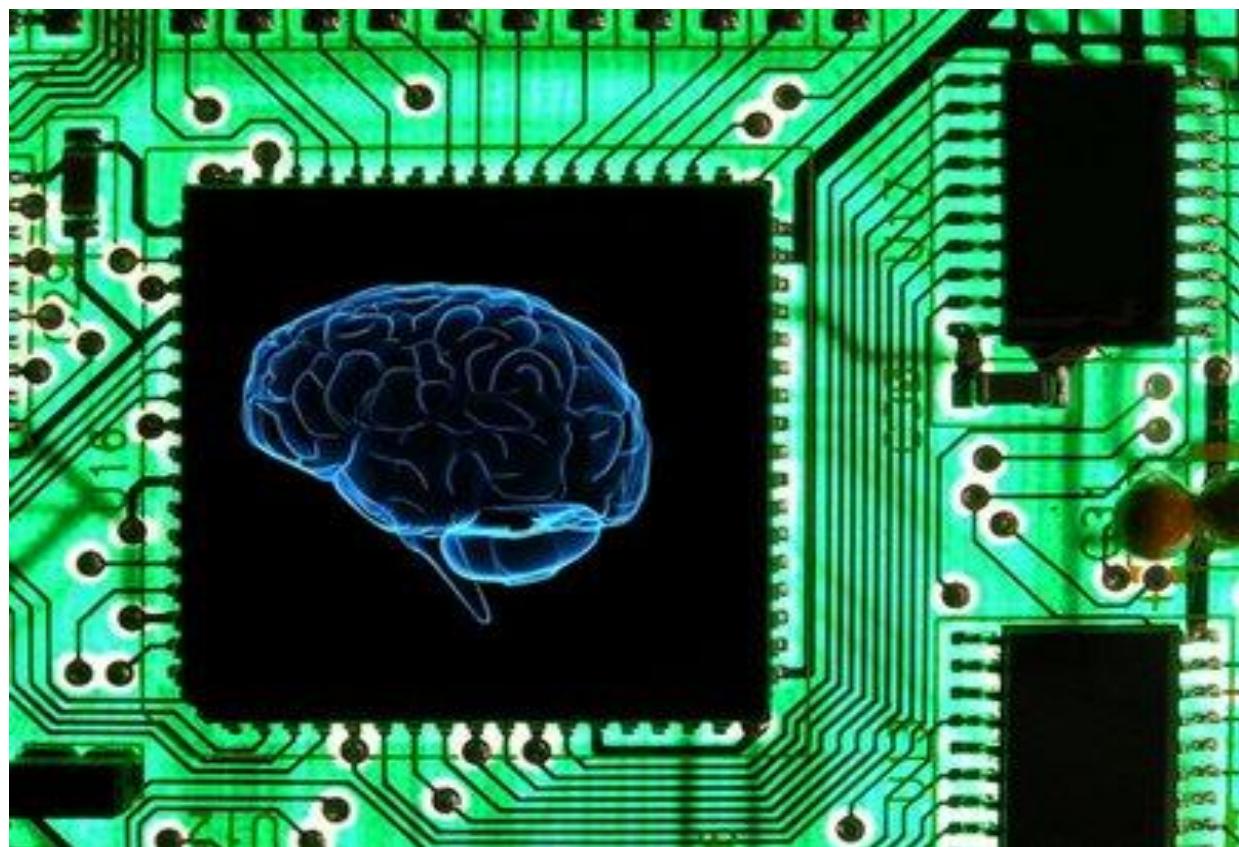
RIKEN Advanced Institute for Computational Science (AICS)

CIRCUITE NEUROMORFE



TRANZISTOR → NANOFIR(Nanowire) → MEMREZISTOR

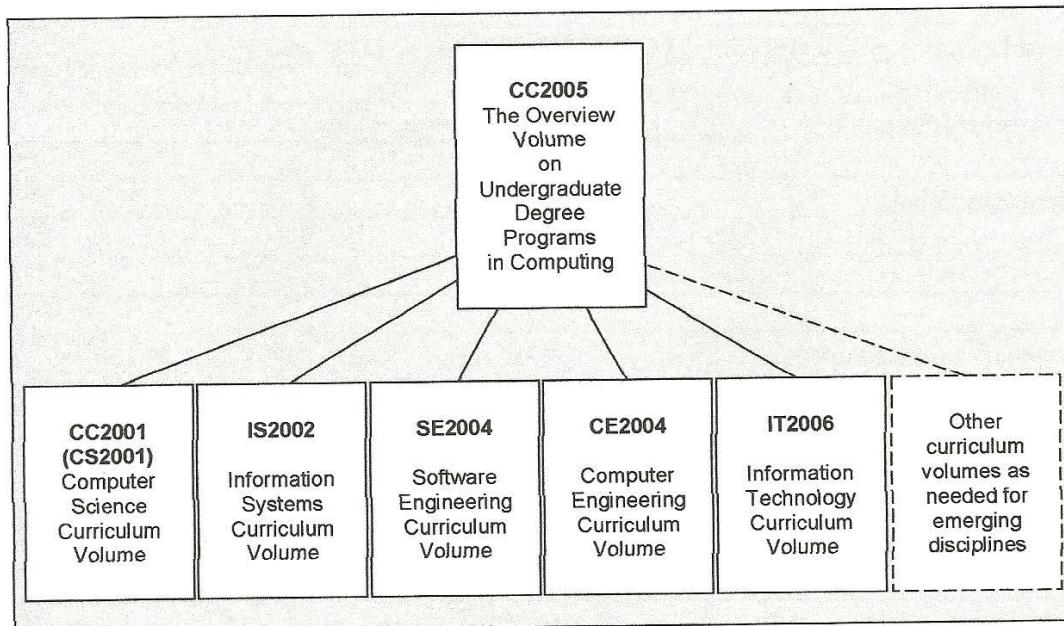
Oare când va fi ?



• Computing Curriculum

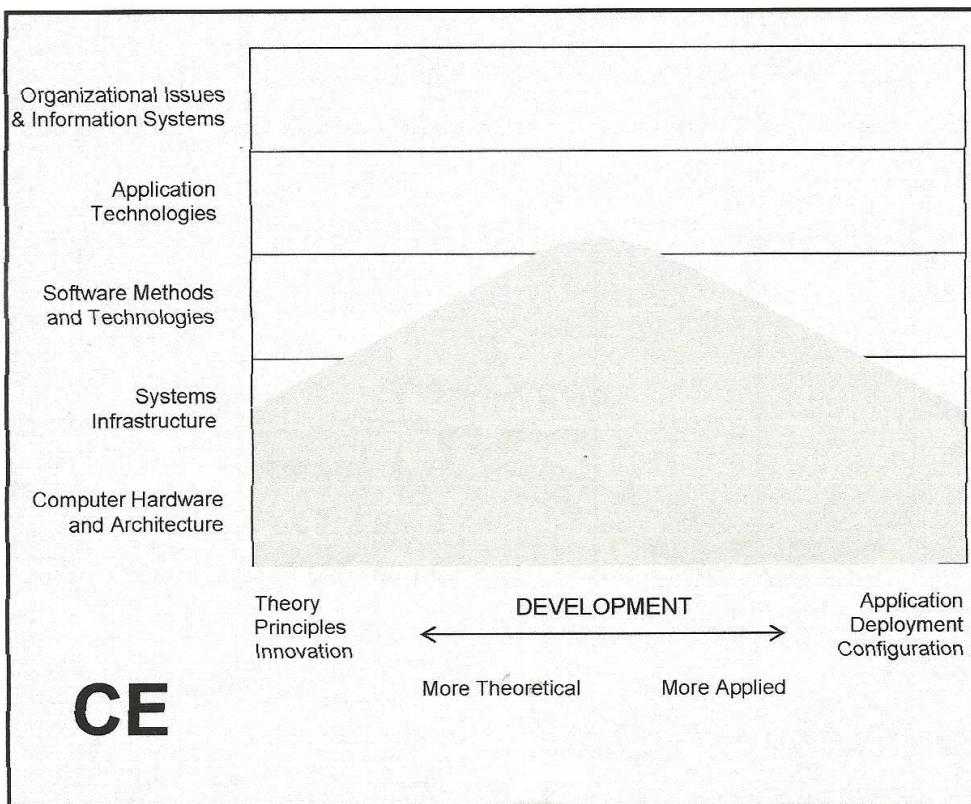
În USA pe câte o durată de o decadă s-a elaborat, de către colective naționale compuse din reprezentanți ai universităților și ai mediului economic, curriculum pentru educația în domeniul calculatoarelor. Pentru decadele '80 și '90 curriculum se referea la domeniul COMPUTERS, iar pentru prima decadă a acestui secol se referă la domeniul COMPUTING. Curriculum pentru Computing conține cinci specializări: Computer Science (CS), Computer engineering (CE), Software engineering (SE), Information technology (IT) și Information Systems (IF)

copies of the computing curricula volumes can be found at <http://www.acm.org/education/curricula.html>
and <http://computer.org/curriculum>.



Computer Engineering

The shaded portion in Figure 2.3 represents the computer engineering discipline. It is broad across the bottom because computer engineering covers the range from theory and principles to the practical application of designing and implementing products using hardware and software. It narrows towards the center as we move upwards because a computer engineer's interests narrow as we move away from the hardware. By the time we get up to the level of software development, we see that the computer engineer's interest has narrowed to the horizontal center because they care about software only inasmuch as they need it to develop integrated devices.



BIBLIOGRAFIE

- [1] John L. Hennessy, David A. Patterson "Computer Architecture- A Quantitative Approach" Morgan Kauffman Publishers, fifth edition, 2012, ISBN 13: 978-0-12-383872-8.
- [2] David A. Patterson, John L. Hennessy, "Computer Organization and Design- The hardware/software interface" Morgan Kauffman Publishers, fourth edition, 2009, ISBN 978-0-12-374493-7.
- [3] Raymond Greenlaw, James H. Hoover "Fundamentals of the Theory of Computation- Principles and Practice", Morgan Kauffman Publishers, fourth edition, 1998 , ISBN 1-55860-547-9
- [4] Jean-Loup Baer "Microprocessor Architectures- From Simple Pipelines to Chip Multiprocessors", Cambridge University Press, 2010, ISBN 9780-521-76992-1.
- [5] Milles J. Murdocca, Vincent P. Heuring "Computer Architecture and Organization: An Integrated Approach" John Wiley and Sons Inc, 2007, ISBN 978-0-471-73388-1.
- [6] Wayne Wolf "Computers as Components- Principles of Embedded Computing System Design" Morgan Kauffman Publishers, second edition, 2008, ISBN 978-0-12-374379-8.
- [7] Lucian N. Vințan "Prediction Techniques in Advance Computing Architectures" Matrix Rom, București, 2007, ISBN 978-973-755-137-5.
- [8] Gheorghe Toacșe " Introducere în Microprocesoare" Editura Stiințifică și Enciclopedică, București, 1986, ediția două, ISBN
- [9] Gheorghe Stefan: "One-Chip TeraArchitecture", in Proceedings of the 8th Applications and Principles of Information Science Conference, Okinawa, Japan on 11-12 January 2009.

CAP 1. ORGANIZAREA UNUI SISTEM PE BAZĂ DE MICROPROCESOR (uP)

- 1.1 Ierarhizarea nivelurilor (hardware/software) într-un calculator
- 1.2 Gap-ul fundamental într-un calculator
- 1.3 Memoria calculatorului
 - 1.3.1 Registre
 - 1.3.2 Memoria principală
 - 1.3.3. Memoria cache
 - 1.3.3.1 Memoria cache cu mapare directă
 - 1.3.3.2 Memoria cache complet asociativă
 - 1.3.3.3 Memoria cache set-asociativă
 - 1.3.3.4 memoria cache multinivel
- 1.4 Perifericele (I/O)
- 1.5 Organizarea unui sistem pe bază de magistrale
 - 1.5.1 Sistemul cu trei magistrale
 - 1.5.2. Caracteristicile magistralei
 - 1.5.3 Magistrale sincrone
 - 1.5.4 Magistrale asincrone
 - 1.5.5 Structurarea ierarhizată a magistralelor unui sistem
 - 1.5.6. Prezent și tendințe în strucurarea pa bază de magistrale
- 1.6 Lega lui Amdahl
- 1.7 Procesorul – Mașină de Procesare a Programelor
 - 1.7.1 Noțiuni: Alfabet, Sir, Limbaj
 - 1.7.2 Structurarea pe niveluri a procesării pe calculator. Mașini virtuale
 - 1.7.3 Organizarea și funcționarea de principiu a unui microprocesor
 - 1.7.4 Limbajul de asamblare
 - 1.7.5 Setul (parțial) de instrucțiuni al procesorului MIPS (R2000)
 - 1.8.6. Etapele în realizarea unui program executabil

CAP 2. ARHITECTURA SETULUI DE INSTRUCȚIUNI

- 2.1 Componentele (arhitecturale ale) microprocesorului
- 2.2 Metrici de performanță pentru microprocessor
- 2.3 Spațiul de adresare.
- 2.4 Moduri de adresare
- 2.5. Tipuri de instrucțiuni
 - 2.5.1 Instrucțiuni de deplasare/ transfer a datelor
 - 2.5.2 Instrucțiuni de transformare a datelor
 - 2.5.3 Instrucțiuni de control al programului
 - 2.5.3.1 Instrucțiuni de ramificație
 - 2.5.3.2 Instrucțiuni cu execuție condiționată
 - 2.5.3.3 Instrucțiuni de lucru cu subroutines
 - 2.5.3.4 Evenimente de excepție, EIT (Exceptions, Interrupts, Traps)
 - 2.5.4. Modalități de lucru ale procesorului cu perifericele
 - 2.5.5. Instrucțiuni de control al procesorului
 - 2.5.6 Instrucțiuni de nivel înalt, HLL
- 2.7 Formatul instrucțiunilor
- 2.8 Intredependența arhitectura setului de instrucțiuni-compilator
- 2.9 Arhitecturi CISC și RISC

CAP 3. CALEA DE DATE

- 3.1 Organizarea de principiu a căii de date
- 3.2 Reprezentarea numerelor în calculator- overflow (depăşirea)
- 3.3 Unitatea aritmetică și logică, alu (arithmetic and logic unit)
 - 3.3.1. Sumatorul.
 - 3.3.1.1 Sumatorul cu Transport Progresiv, STP
 - 3.3.1.2 Sumatorul cu transport anticipat, CLA (Carry-Look-Ahead adder)
 - 3.3.1.3 Sumatorul cu selectarea transportului, CSA (Carry Select Adder)
 - 3.3.2 Multiplicatorul - pentru numere fără semn (Multiply unsigned- multu)
 - 3.3.3 Unitatea de procesare în virgulă flotantă
 - 3.3.3.1 Reprezentarea în virgulă flotantă
 - 3.3.3.2 Reprezentarea numerelor conform standardului IEEE-754
 - 3.3.3.3 Operații în virgulă flotantă
- 3.4 Registre
 - 3.4.1 Structurarea registrelor într-un microprocesor.

CAP 4. CALEA DE CONTROL

- 4.1 Funcția unității de control
- 4.2 Unitatea de control cablată
- 4.3 Unitatea de control micropogramată

CAP 5. TEHNICI ȘI STRUCTURI PENTRU CREȘTEREA PERFORMANȚELOR

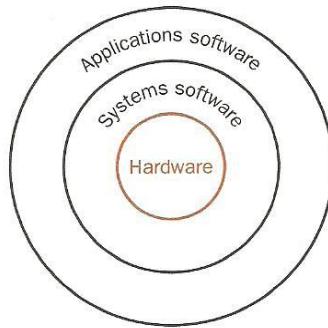
- 5.1 Procesarea de tip pipeline
 - 5.1.1 Organizarea de principiu pentru un pipeline
 - 5.1.2 Procesarea de tip pipeline - Noțiuni fundamentale
 - 5.1.3 Procesarea în pipeline la procesorul MIPS
 - 5.1.4 Hazardul în pipeline
 - 5.1.4.1 Hazardul structural
 - 5.1.4.2 Hazardul de date
 - 5.1.4.3 Hazardul de nume.
 - 5.1.4.4 Hazardul de control
- 5.2 Microprocesoare cu execuții multiple
 - 5.1.1 Microprocesoare superscalare
 - 5.1.2 Microprocesoare de tip VLIW (Very Long Instruction Word)
 - 5.1.3 Procesoare vectoriale
- 5.3 Procesarea de tip multithread
- 5.4 Procesoare grafice

CAP 1.

ORGANIZAREA UNUI SISTEM PE BAZĂ DE MICROPROCESOR (μ P)

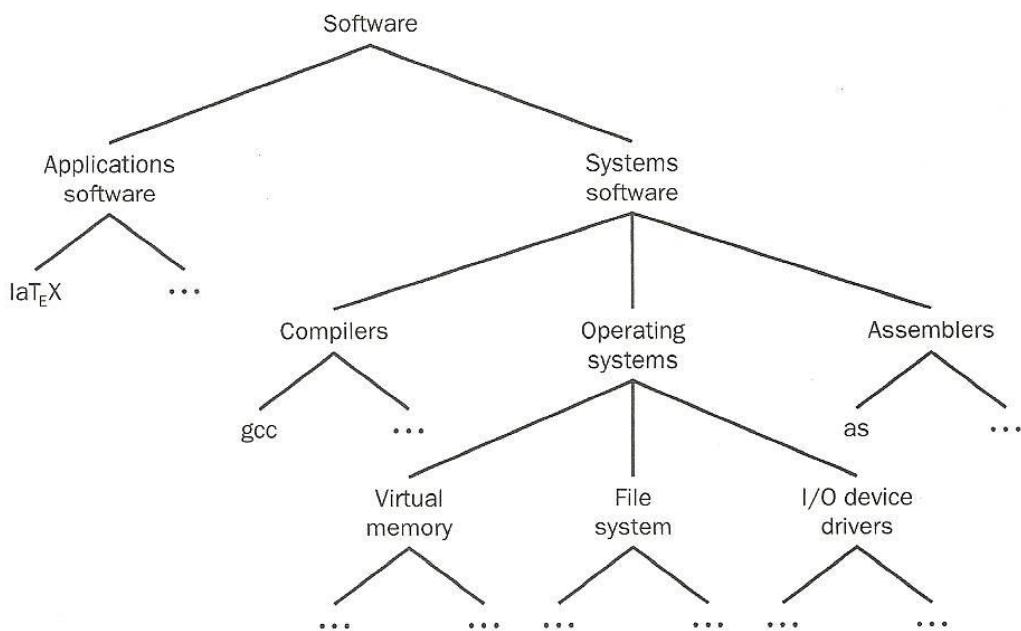
1.1 IERAHIZAREA NIVELURILOR (HARDWARE/SOFTWARE) INTR-UN CALCULATOR

- Ierahizare sistem



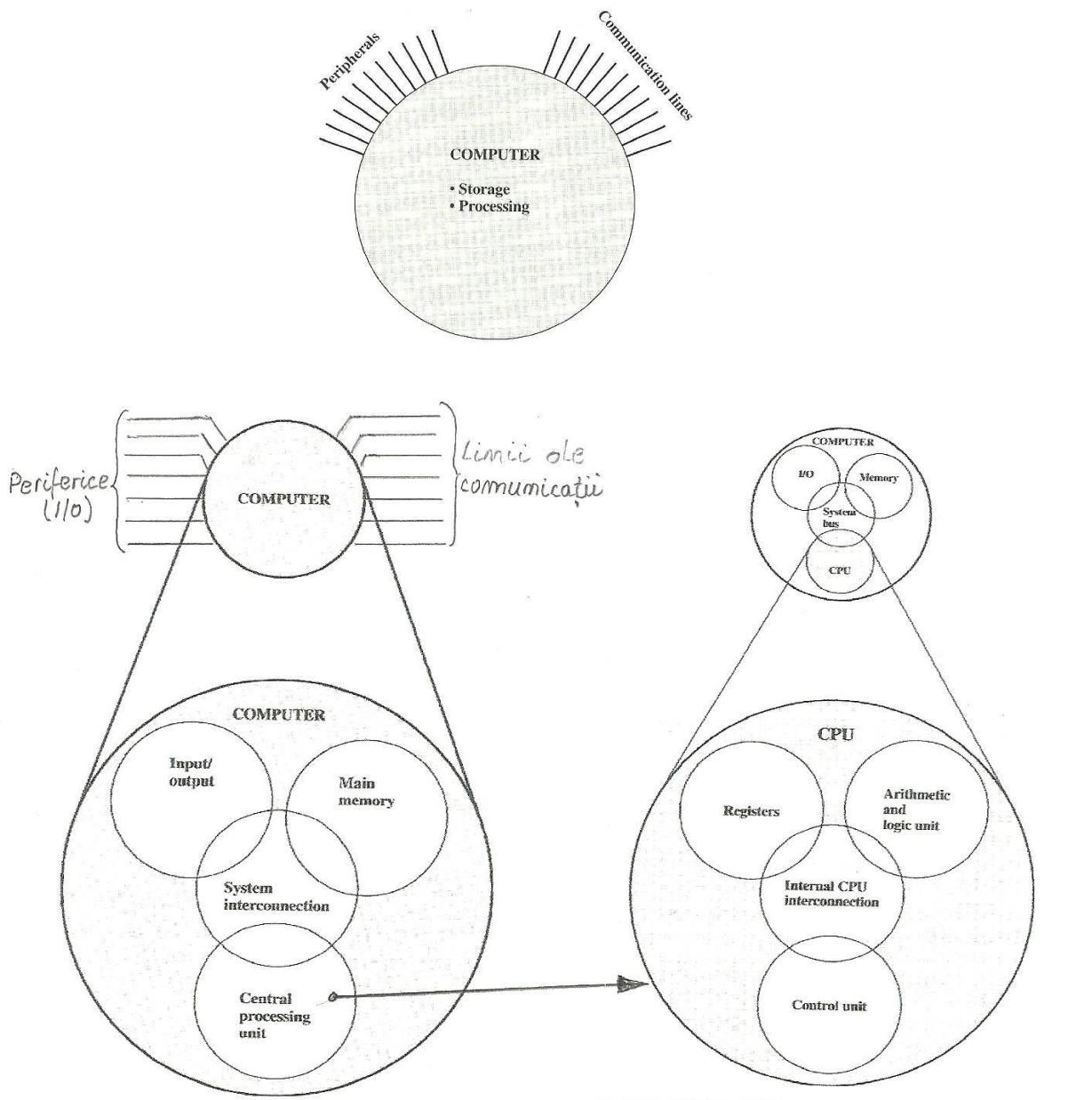
A simplified view of hardware and software as hierarchical layers, classically shown as concentric rings building up from the core of hardware to the software closest to the user.

- Structurare/organizare software

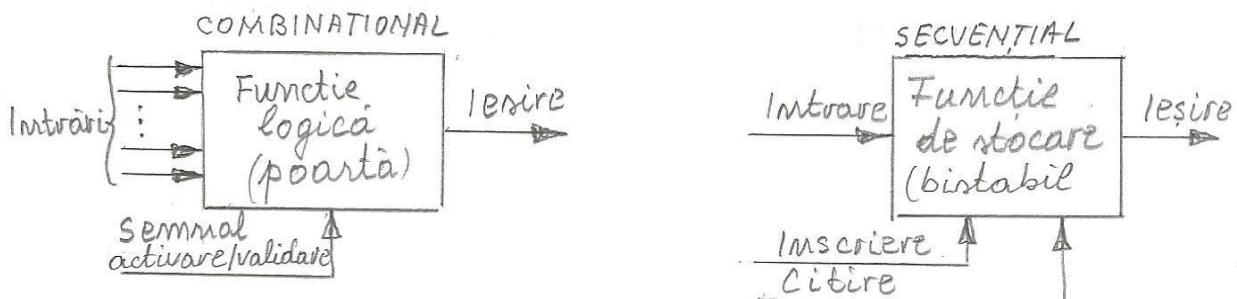


An example of the decomposability of computer systems. The terms in the middle of the chart, such as LaTeX and gcc, are examples of Unix programs. The terms lower in the chart, such as virtual memory, will be introduced in Chapters 7 and 8.

- Structurare/organizare hardware



- Elemente hardware fundamentale (combinționale, secvențiale).



1.2 GAP-UL FUNDAMENTAL ÎNTR-UN CALCULATOR

Rolul memoriei principale din componența calculatorului este de a stoca și de a livra, în fiecare ciclu de ceas, T_{clk} , care comandă funcționare procesorului, informația (instructiuni și date) necesară pentru procesare. Dar, apare o discrepanță/gap între viteza de procesare a procesorului și **latența memoriei** (timpul de răspuns al memoriei din momentul când s-a apelat o informație și momentul când această informație este livrată/disponibilă procesorului). Dacă se consideră că procesorul necesită informație în fiecare ciclu de ceas, latența memoriei este de mai multe cicluri de ceas. Raportul dintre latența memoriei și T_{clk} era de 5:1 în 1990, iar în prezent a ajuns la mai mult de două ordine de mărime. De exemplu, dacă $f_{clk} = 2.5\text{GHz} \rightarrow T_{clk} = 1/f_{clk} = 0.4\text{ns}$, la o latență de 40ns a memorilor DRAM actuale rezultă o latență exprimată în cicluri egală cu $40\text{ns}/0.4\text{ns} = 100$ de cicluri de ceas. Viteza procesorului crește cu 60%/pe an pe când latența memoriei scade doar cu 7%/pe an, ceea ce se reflectă din diagrama următoare. Odată cu procesoarele multicore deoarece frecvența de ceas pare a se fi stabilizat sub 3-3.5 GHz, pentru aceste procesoare, aspectul de alimentare al procesorului de către memorie apare sub forma creșterii lărgimii de bandă pentru accesul la memorie încât toate core-urile să primească informația necesară (instructiuni/date).

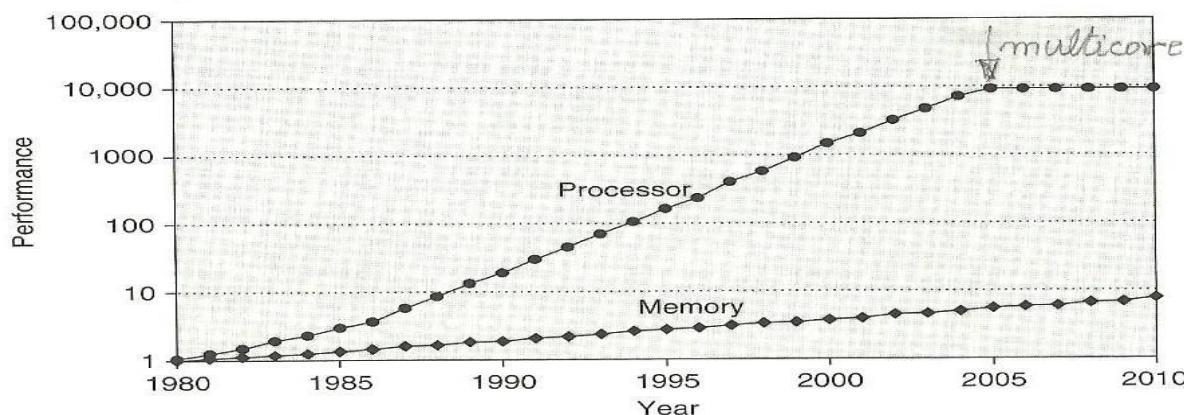


Figure 2.2 Starting with 1980 performance as a baseline, the gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over time. Note that the vertical axis must be on a logarithmic scale to record the size of the processor–DRAM performance gap. The memory baseline is 64 KB DRAM in 1980, with a 1.07 per year performance improvement in latency (see Figure 2.13 on page 99). The processor line assumes a 1.25 improvement per year until 1986, a 1.52 improvement until 2000, a 1.20 improvement between 2000 and 2005, and no change in processor performance (on a per-core basis) between 2005 and 2010; see Figure 1.1 in Chapter 1.

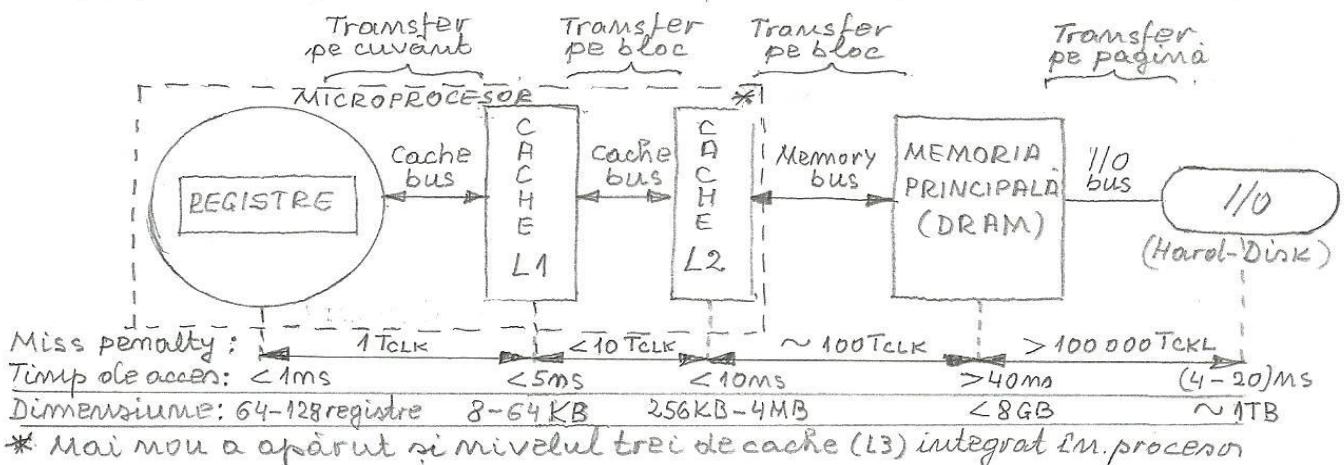
Acest gap nu poate fi redus fizic, dar poate fi ascuns prin introducerea unui buffer între μP și memorie, ceea ce se realizează prin **memoria cache**, care se structurează pe două niveluri L1 și L2 integrate în procesor, mai nou chiar și al treilea nivel, L3, este integrat (procesoarelor dedicate pentru desktop sau servere). Memorie cache este o memorie RAM statică, deci permite un timp de acces de ordinul ns (în funcție de capacitatea sa). Această “ascundere” a latenței memoriei principale se realizează prin aducerea informației (instructiuni, date), care este necesară în prezent și în timpul următor, din memoria principală în memoria cache, ce prezintă o latență mai redusă (pentru L1 latența poate fi de $1T_{clk}$). Prin această aducere a informației necesare de pe hard-disk și înscrierea sa în memoria principală, apoi din memoria principală se transferă în L2 și apoi se transferă în L1 se realizează o incluziune multinivel, adică informația din L1 există în L2, totă informația din L2 există în Memoria principală (Mem) și evident totă aceasta informație există în Hard-disk (HD), **L1 ⊂ L2 ⊂ Mem ⊂ HD**. Acest “trick” prin care se ascunde latență memoriei se sprijină pe **principiul localizării în timp și în spațiu**:

- codul (instructiunile) și datele utilizate recent de procesor, cu o probabilitate destul de ridicată, vor fi reutilizate și într-un timp imediat (**localizarea temporală**);

- codul (instructiunile) și datele utilizate de procesor care au fost accesate dintr-un spațiu (o zonă de adrese din memorie), cu o probabilitate destul de ridicată, această zonă va fi accesată din nou (**localizarea spațială**).

Dacă informația solicitată de procesor nu se află în L1 atunci se accesează L2, dacă nu se află nici aici atunci se accesează Mem, sau chiar HD, apoi cu informația căutată găsită se parcurge traseul în sens invers, înscriindu-se succesiv informația în fiecare nivel până ajunge în procesor. Evident că penalizarea în timp (procesorul așteaptă) este cu atât mai mare cu cât informația căutată se află într-un nivel mai depărtat de procesor. De exemplu, conform datelor din figura următoare, o informație care se află în memoria principală pentru a fi adusă succesiv din nivel în nivel va necesita $111T_{clk}$ ($100+10+1=111$).

• Ierarhizarea nivelurilor de memorie într-un calculator.

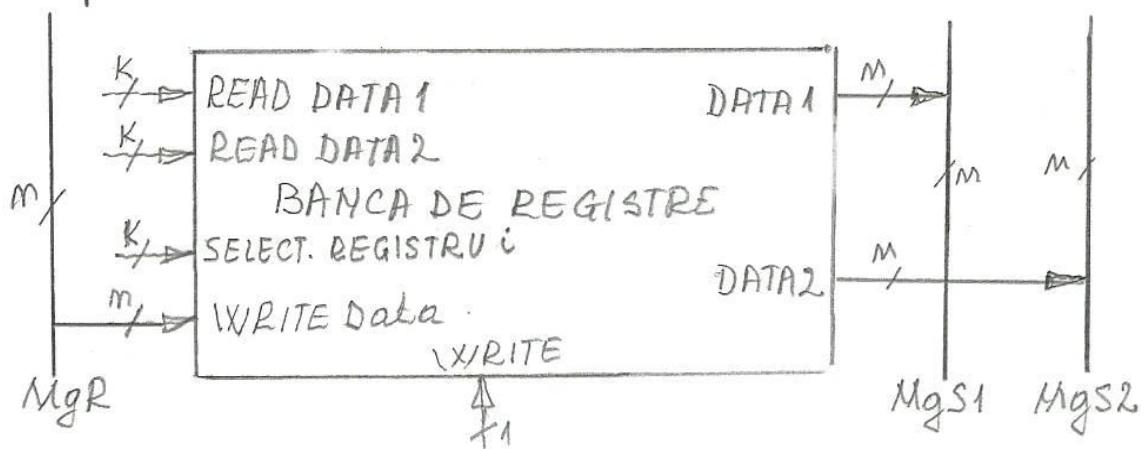


1.3 MEMORIA CALCULATORULUI

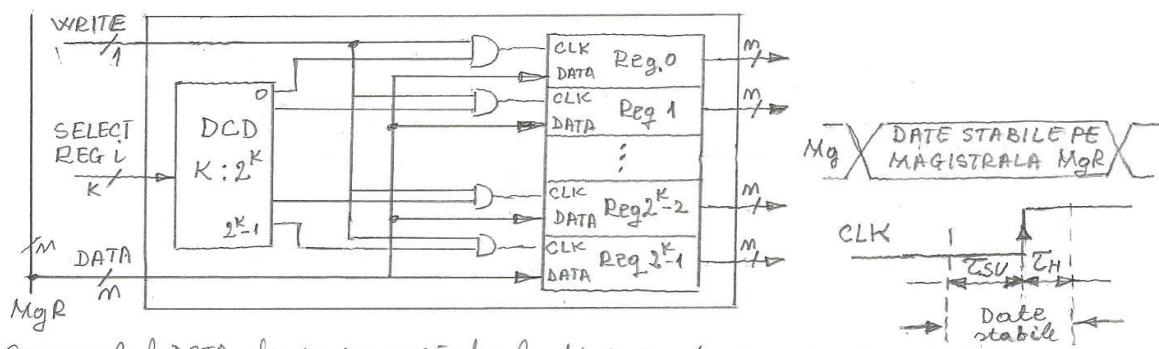
1.3.1 Registre

Registrul este memoria internă a procesorului, are viteza de înscrisere și de citire cea mai ridicată din sistem, poate fi $< T_{clk}$. Registrele sunt organizate în bănci de registre, într-o bancă fiind, ușual, un număr de registre egal cu puteri al lui 2 (16, 32, 64, 128, 256); pentru o lungime de n biți, a fiecărui registru, rezultă capacitatea băncii, $C = 2^k \cdot n$ biți. Structurarea unei bănci de registre în care se poate înscrive informația de pe o magistrală de intrare (magistrală rezultat, MgR), iar conținutul dintr-un registru se poate citi și aplica simultan pe două magistrale de ieșire (magistrale sursă, MgS1, MgS2) este cea din figura următoare, (fiecare registru este dublu port pe ieșire, se pot efectua simultan două citiri). (O operație între doi operanzi A și B produce un rezultat, $R \leftarrow A$ Operație B, utilizează cei doi operanzi de pe magistralele sursă MgS1, MgS2, care sunt citiți simultan din banca de registre, iar rezultatul R al Operației este pus pe magistrala rezultat, MgR, și înscris în banca de registre).

• Reprezentare schema bloc

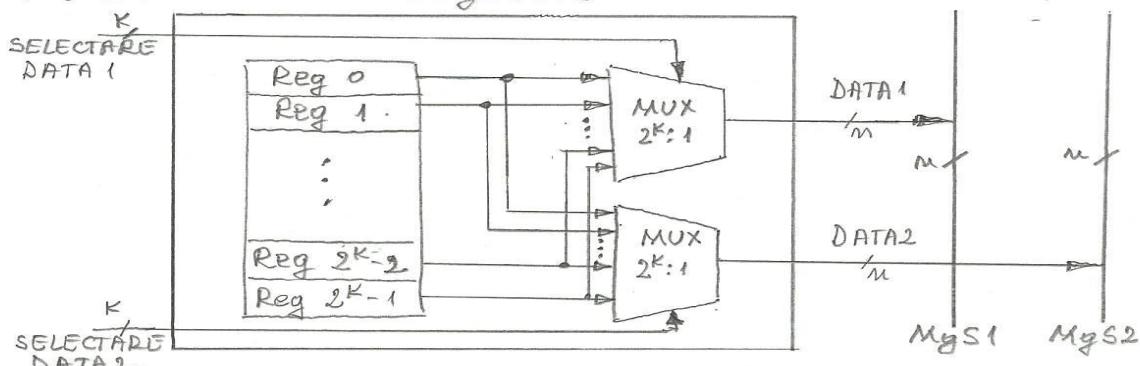


- structurarea pe intrare (pentru înscrivere) a unei bănci de registre



Semnalul DATA de pe magistrala MgR trebuie să fie stabil pe intervalul $\Delta = TSU + TH$, pentru a fi înscris corect într-un registru

- structurarea pe ieșire (pentru citire) dublu port a unei bănci de registre.



uzual, ieșirea registrelor este de tip TSL

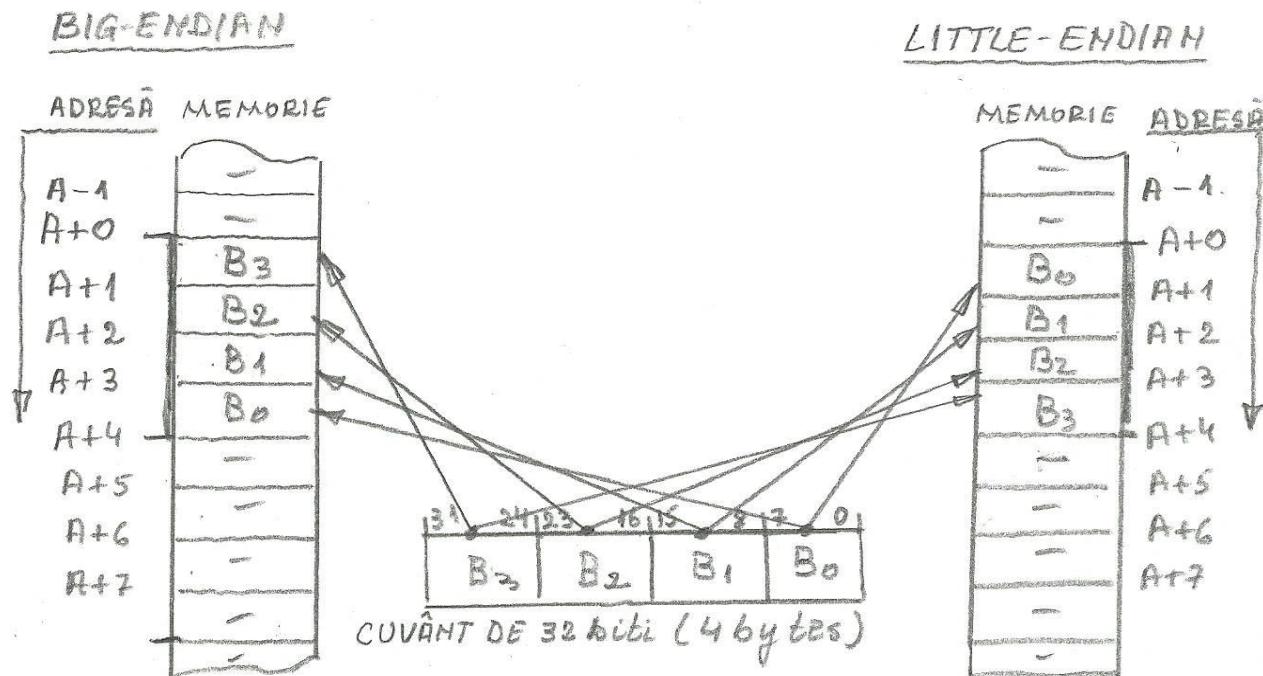
1.3.2 Memoria principală

- Plasarea cuvintelor în memorie.

Uzual, memoria este organizată pe byte, adică în locația corespunzătoare unei adrese, A, este stocat un cuvânt cu lungimea de un byte, ceea ce înseamnă că un cuvânt de 32 biți (4 bytes) ocupă în memorie patru locații consecutive la adresele A+0, A+1, A+2, A+3. Evident, că o accesare de memorie la o adresă A care nu este un multiplu de patru (A modulo 4 ≠ 0) va citi eronat o dată sau o instrucțiune, o parte din cuvântul citit are bytes din

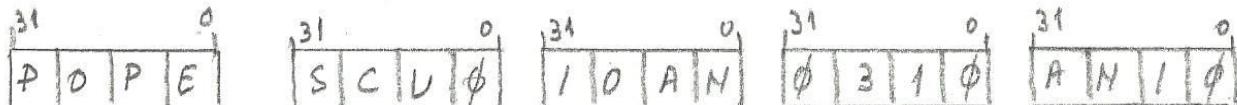
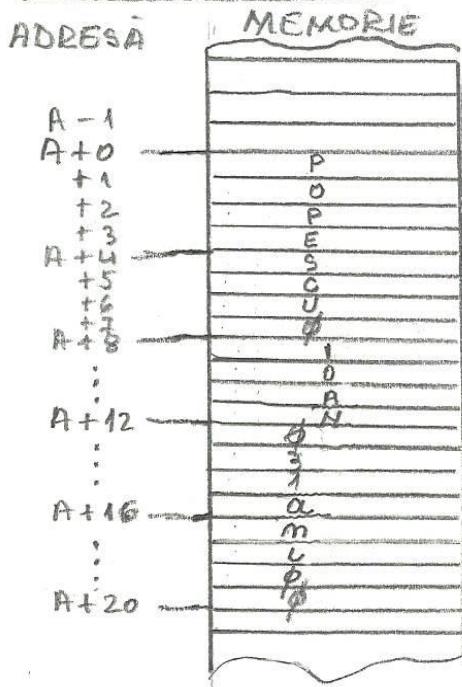
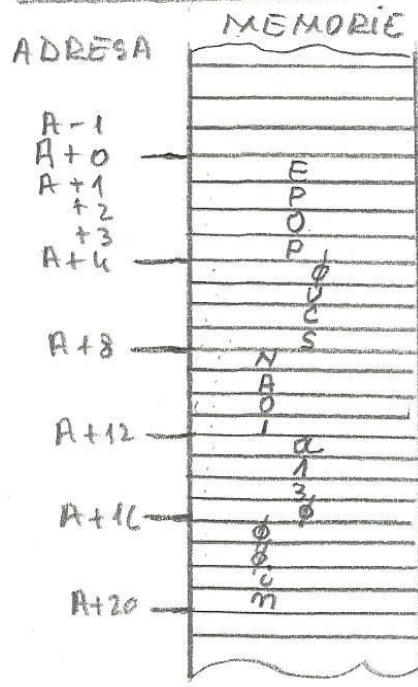
cuvântul ce trebuie citit iar restul de bytes de la cuvântul următor (μ P are un mecanism prin care semnalizează o citire care nu respectă această regulă); pentru un cuvânt de n bytes regula generată de accesare corectă la memorie (de aliniere) este **A modulo n = 0**.

Există două convenții de plasare în memorie pentru bytes unui cuvânt: big-endian și little-endian. După **convenția little-endian**, byte-ul cel mai puțin semnificativ al cuvântului este plasat în memorie la adresa cea mai mică ($A+0$), pe când după **convenția big-endian**, byte-ul cel mai semnificativ al cuvântului este plasat în memorie la adresa cea mai mică ($A+0$). În figura următoare B_3 este byte-ul cel mai semnificativ (big), iar B_0 cel mai puțin semnificativ (little). La procesoarele actuale se poate programa, pentru plasarea în memorie, oricare din cele două convenții.

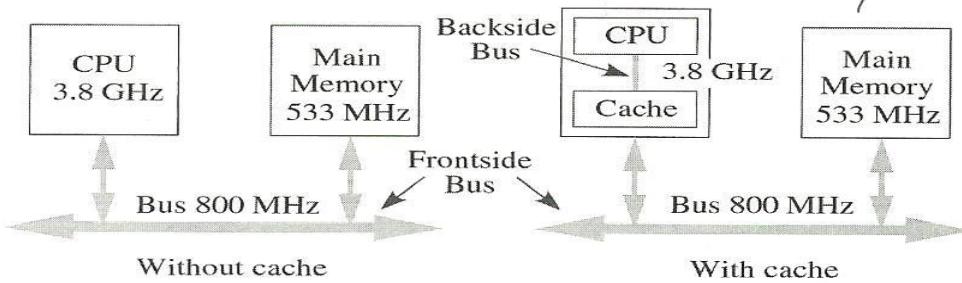


Exemplul 1.1. Numele POPESCU IOAN 31 ani să fie scris în memorie conform celor două convenții de plasare.

Soluție. Fiecare character este codificat în ASCII, deci necesită 8 biți, se codifică și blancul dintre cuvinte, \emptyset . Plasarea rezultată este următoarea:

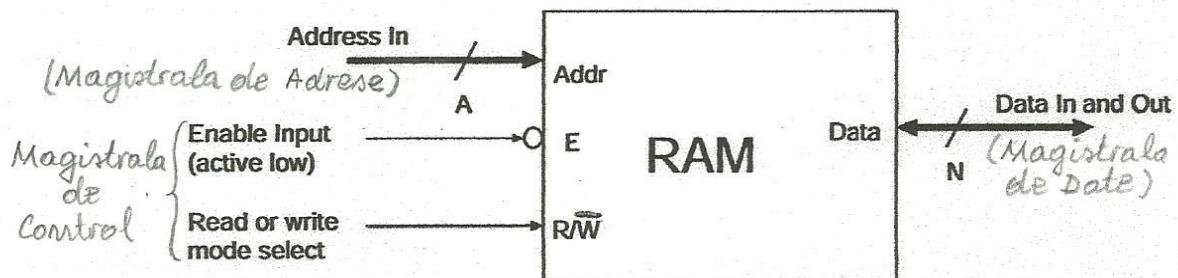
BIG-ENDIANLITTLE-ENDIAN

- Modul de conectare al memoriei la procesor

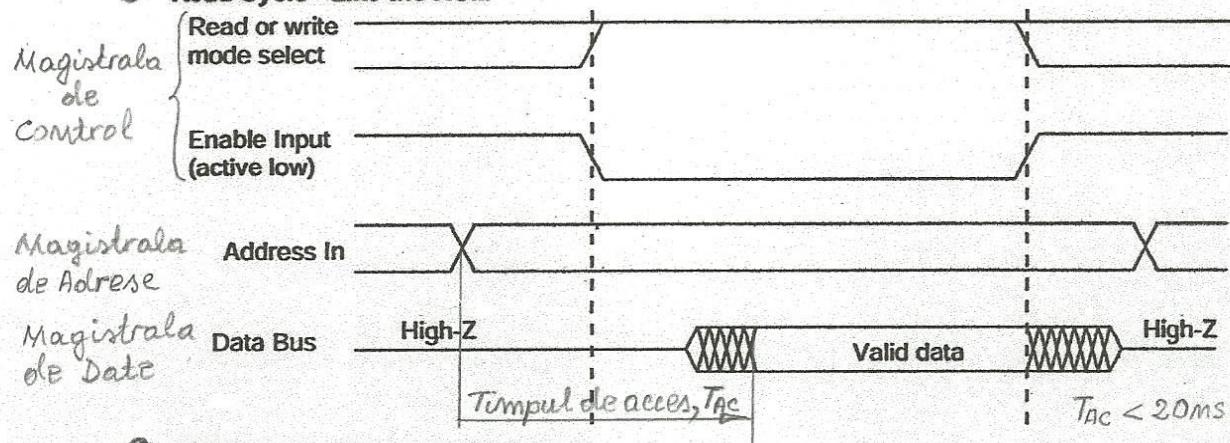


Memoria RAM statică

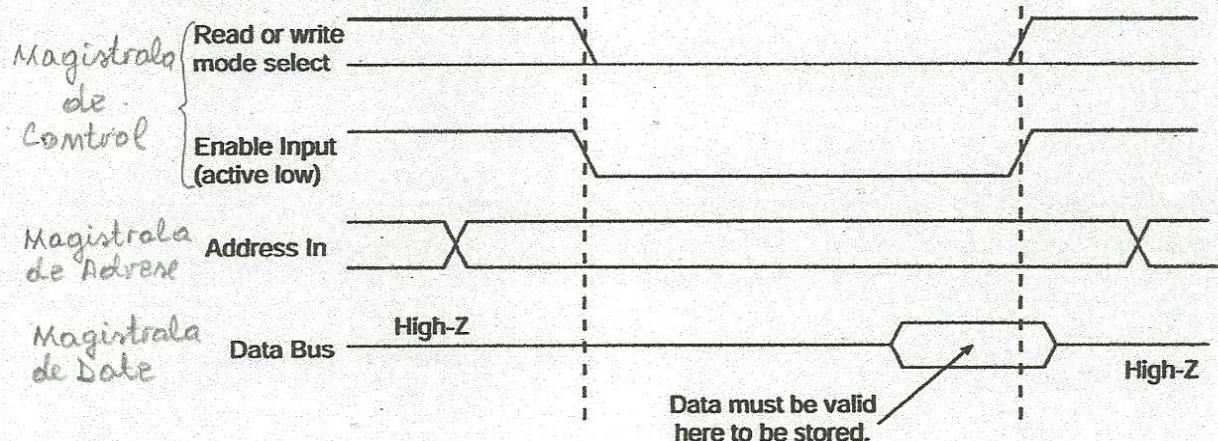
• Reprezentare schematică bloc



• Read Cycle - Like the ROM

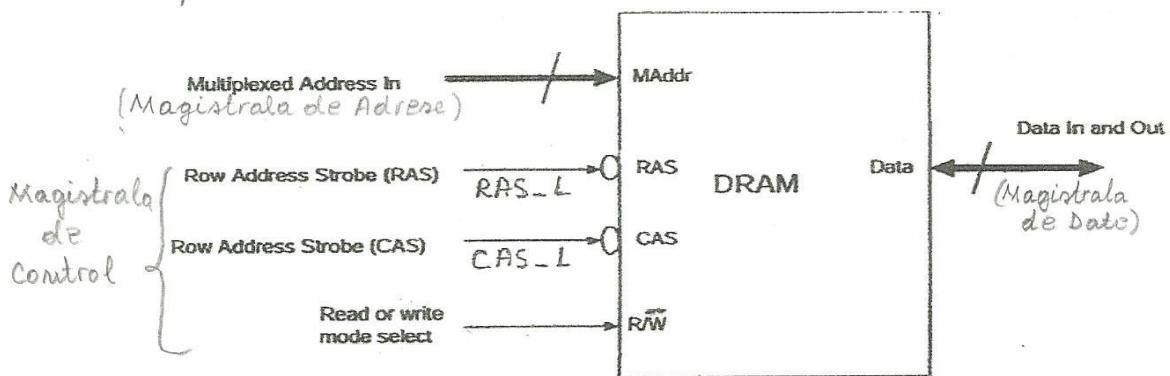


• Write Cycle - Data stored internally



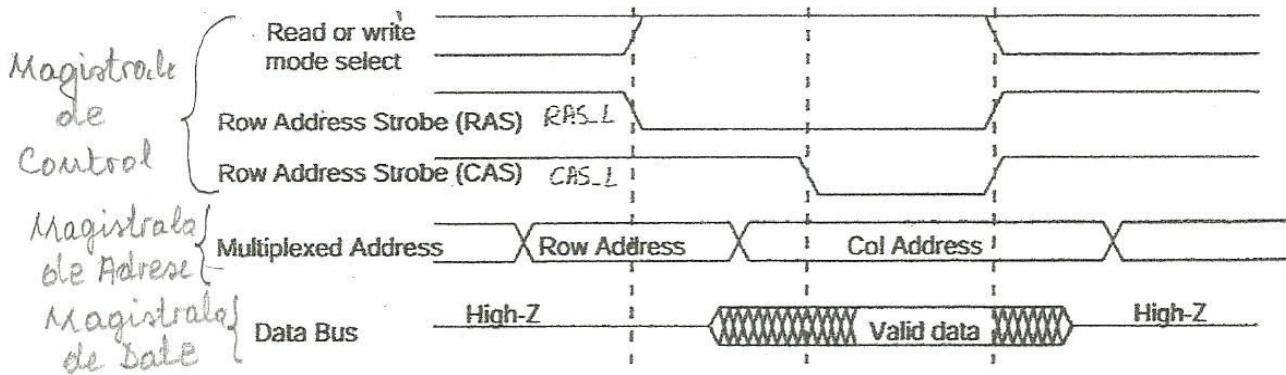
Memoria RAM dinamică (DRAM)

• Reprezentare schematică bloc



Cuvântul de adresă $A = A_{m-1}A_{m-2}\dots A_{\frac{m}{2}}A_{\frac{m}{2}-1}\dots A_1A_0$ se divide în semicuvântul $A_{n-1}A_{n-2}\dots A_{\frac{n}{2}}, A_m$, care este strobosat cu semnalul RAS (Row Address Strobe) și va selecta o linie a matricei de memorie, apoi pe magistrala de adrese se aplică semicuvântul inferior $A_{m-1}, A_{m-2}\dots A_1, A_0$, care este strobosat cu semnalul CAS (Column Address Strobe) și va selecta o coloană a matricei de memorie. Pentru un cuvânt de adresă de n biti, magistrala de adrese necesită doar $n/2$ linii; mărind numărul de linii de adresare cu 1, capacitatea memoriei crește de patru ori $[2^{(n/2)+1} \times 2^{(n/2)+1}] = [2^{n/2} \times 2^{n/2}] = 2^n = 4$, rezultă că pentru capacitatea de memorie DRAM capacitatele sunt multiplu de patru.

• Read Cycle (write is similar)

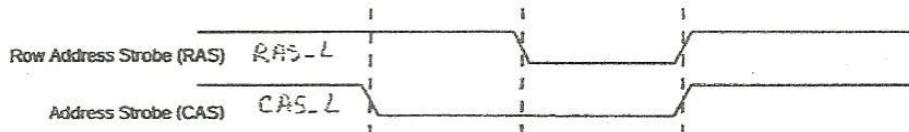


A DRAM has a multiplexed address bus and the address is presented in two halves, known as row and column addresses. So the capacity is $4^A \times D$. A 4 Mbit DRAM might have $A=10$ and $D=4$.

When a processor (or its cache) wishes to read many locations in sequence, only one row address needs to be given and multiple column addresses can be given quickly to access data in the same row. This is known as 'page mode' access.

EDO (extended data out) DRAM is now quite common. This guarantees data to be valid for an extended period after CAS, thus helping system timing design at high CAS rates.

• Refresh Cycle - must happen sufficiently often!



No data enters or leaves the DRAM during refresh, so it 'eats memory bandwidth'. Typically 512 cycles of refresh must be done every 8 milliseconds.

- Din diagramele anterioare rezultă că funcționarea memoriei, atât cea statică cât și cea dinamică au o funcționare asincronă (în aceste diagrame operațiile nu sunt coordonate prin semnal de ceas, CLK). Pentru ca memoria DRAM să poată fi integrată în sistemele cu μP, care sunt sisteme sincrone (sunt “pilotate” de un ceas, sau mai multe) s-a realizat memoria DRAM sincronă, **SDRAM**, la care operațiile de citire și înscriere sunt sincronizate.
- La memorile SDRAM, clasic, operațiile de citire sau înscriere se realizează pe un front (crescător sau descrescător) al semnalului de ceas, ceea ce înseamnă că se realizează o operație pe perioada T_{clk} . În prezent, usual, memorile SDRAM utilizează ambele fronturi ale semnalului de ceas, ceea ce înseamnă că se realizează o rată dublă de date, **DDR**(Double Data Rate) obținându-se memoria **DDR SDRAM**, pentru operațiile de citire și înscriere; practic, se utilizează pe lângă frontul crescător de la semnalul T_{clk} și frontul crescător de la semnalul de ceas negat, \bar{T}_{clk} , ca în reprezentarea următoare



Production year	Chip size	DRAM type	Row access strobe (RAS)		Column access strobe (CAS)/ data transfer time (ns)	Cycle time (ns)
			Slowest DRAM (ns)	Fastest DRAM (ns)		
1980	64K bit	DRAM	180	150	75	250
1983	256K bit	DRAM	150	120	50	220
1986	1M bit	DRAM	120	100	25	190
1989	4M bit	DRAM	100	80	20	165
1992	16M bit	DRAM	80	60	15	120
1996	64M bit	SDRAM	70	50	12	110
1998	128M bit	SDRAM	70	50	10	100
2000	256M bit	DDR1	65	45	7	90
2002	512M bit	DDR1	60	40	5	80
2004	1G bit	DDR2	55	35	5	70
2006	2G bit	DDR2	50	30	2.5	60
2010	4G bit	DDR3	36	28	1	37
2012	8G bit	DDR3	30	24	0.5	31

Figure 2.13 Times of fast and slow DRAMs vary with each generation. (Cycle time is defined on page 97.) Performance improvement of row access time is about 5% per year. The improvement by a factor of 2 in column access in 1986 accompanied the switch from NMOS DRAMs to CMOS DRAMs. The introduction of various burst transfer modes in the mid-1990s and SDRAMs in the late 1990s has significantly complicated the calculation of access time for blocks of data; we discuss this later in this section when we talk about SDRAM access time and power. The DDR4 designs are due for introduction in mid- to late 2012. We discuss these various forms of DRAMs in the next few pages.

Standard	Clock rate (MHz)	M transfers per second	DRAM name	MB/sec /DIMM	DIMM name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10,664	PC10700
DDR3	800	1600	DDR3-1600	12,800	PC12800
DDR4	1066–1600	2133–3200	DDR4-3200	17,056–25,600	PC25600

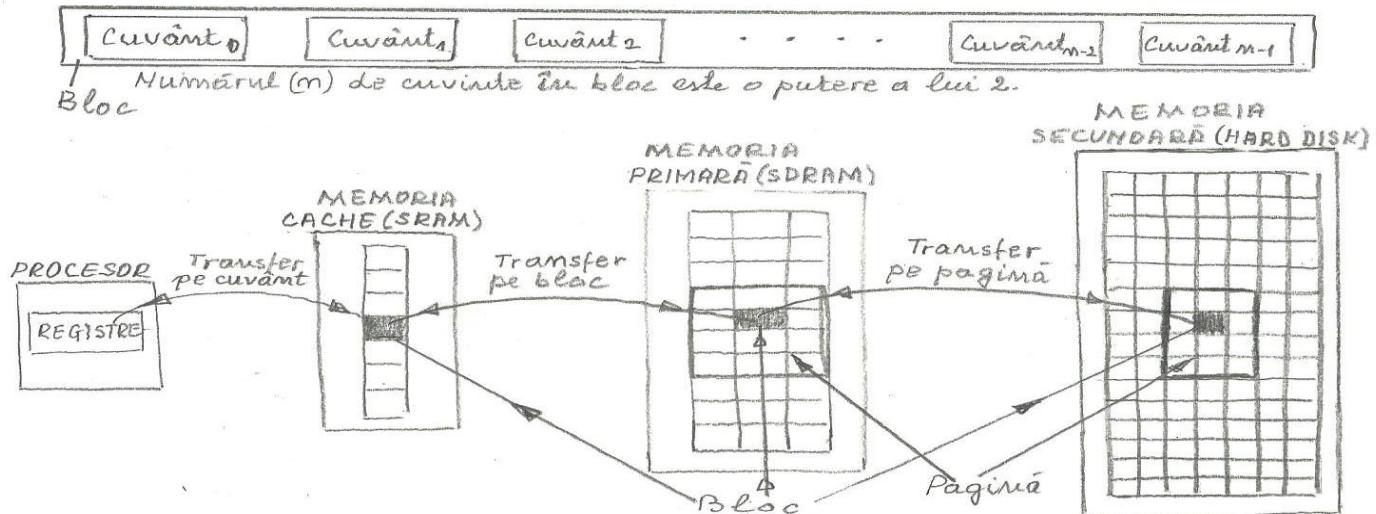
Figure 2.14 Clock rates, bandwidth, and names of DDR DRAMs and DIMMs in 2010. Note the numerical relationship between the columns. The third column is twice the second, and the fourth uses the number from the third column in the name of the DRAM chip. The fifth column is eight times the third column, and a rounded version of this number is used in the name of the DIMM. Although not shown in this figure, DDRs also specify latency in clock cycles as four numbers, which are specified by the DDR standard. For example, DDR3-2000 CL 9 has latencies of 9-9-9-28. What does this mean? With a 1 ns clock (clock cycle is one-half the transfer rate), this indicates 9 ns for row to columns address (RAS time), 9 ns for column access to data (CAS time), and a minimum read time of 28 ns. Closing the row takes 9 ns for precharge but happens only when the reads from that row are finished. In burst mode, transfers occur on every clock on both edges, when the first RAS and CAS times have elapsed. Furthermore, the precharge is not needed until the entire row is read. DDR4 will be produced in 2012 and is expected to reach clock rates of 1600 MHz in 2014, when DDR5 is expected to take over. The exercises explore these details further.

Uzual, pentru calculatoare (desktop și servere), capacitatea de memorie este realizată din mai multe circuite care sunt implantate pe ambele părți ale unei mici placete de circuit imprimat referit modul DIMM (Dual In line Memory Modul). Tipic, un DIMM conține 4-16 circuite DRAM și este organizat pe cuvinte de 8 biți + ECC (Error Correcting Code)

- Pentru memoriile RAM statice există varianta QDR SRAM, care mărește de patru ori debitul total de date, la operațiile de citire și înscrriere; prezintă atât pe intrare câte două porturi (dublu port), pe intrare pentru înscrriere și două porturi (dublu port) pe ieșire pentru citire, unul din porturi este sincronizat cu T_{clk}, iar celălat cu \bar{T}_{clk} .
- Costurile per bit se încadrează:
 - SRAM 2000-5000\$/GB
 - DRAM 20-75\$/GB
 - Disk magnetic: Timpul de acces, T_{AC} = 5 000 000- 20 000 000ns; preț 0,20-2\$/GB
- Aspectele care sunt în atenția proiectantului de sistem pe bază de μP pentru optimizarea memoriei principale sunt:
 1. **Latență** (pentru valorile anterioare T_{AC} = 40-70ns, la frecvența procesorului de 3GHz înseamnă că accesul la memorie necesită (120-210)xT_{clk});
 2. **Capacitatea pe chip** (8Gbit/chip, la nivelul 2008);
 3. **Lățimea de bandă, Bandwidth** (vezi magistrale).

1.3.3. Memoria cache.

Introducerea memoriei cache (cache— a safe place for hiding or storing things) este modalitatea prin care se "ascunde" latența memoriei. Procesorul necesită în fiecare perioadă/tact de ceas, T_{clk}, să fie alimentat cu informație (instrucțiune și/sau data) care este stocată în memorie, dar accesarea la memoria principală (SDRAM) necesită un timp de ordinul sute de tacte, aceasta înseamnă că procesorul trebuie să stea (stall, oprește procesarea) sute de tacte până primește informația necesară. Pentru a elimina această oprire din funcționare a μP se introduce între memoria principală și μP un buffer de mare viteză, adică un nivel (L1) suplimentar de memorie, sau două (L1+ L2), care formează memoria cache. Memoria cache este în fond o memorie SRAM de dimensiuni reduse care are un timp de acces 1-2 tacte. Rațiunea și eficiența memoriei cache se bazează pe localizarea (în timp și spațiu) existentă în programele rulate pe calculator și pe incluziunea informației stocate (blocul existent în cache există și în nivelurile inferioare, memoria principală (Mem), memoria secundară (HD), L1 ⊂ L2 ⊂ Mem ⊂ HD). Structurarea nivelurilor de memorie în prezența memoriei cache este prezentată în figura următoare.



Operația de citire. Procesorul trimite la memoria cache adresa cuvântului (instrucțiune/data) solicitat și dacă acesta există (**hit**) este trimis la procesor în 1-2 tacte de ceas (transfer pe cuvânt). Dacă acest cuvânt nu este în cache (**miss**) se accesează nivelul inferior (memoria primară/principală) pentru care se consumă de ordinul sute de tacte de ceas (**miss penalty**); dacă cuvântul căutat se află în memoria principală (**hit**) atunci blocul, în care se află

cuvântul respectiv, este transferat (transfer pe bloc) în memoria cache de unde, apoi, se citește cuvântul de adresă căutat și se trimită în procesor, în totalitate procesul de citire din memoria principală consumă de ordinul sute de tacte de ceas. Dacă blocul căutat nu se află (miss) în memoria principală atunci se accesează nivelul inferior (hard disk), iar pagina de pe hard disk în care se află blocul căutat este transferată (transfer pe pagină) și înscrisă în memoria principală, iar din memoria principală blocul respectiv transferat (transfer pe bloc) și înscris în memoria cache, de unde cuvântul căutat se trimită la procesor, tot acest proces consumă milioane de tacte de ceas. (Se observă că se merge (la miss) înspre niveluri inferioare din nivel în nivel, apoi se aduce cuvântul căutat parcurgând aceste niveluri de memorie în sens invers).

Operația de înscriere. Procesorul trimite la adresa din memoria cache cuvântul care trebuie înscris. Pentru înscriere există două metode: write-through și write-back. Prin **write-through** cuvântul este înscris întâi la adresa din memoria cache și apoi se continuă și cu înscrierea în memoria principală; deoarece se face acces la memoria principală se consumă un timp similar cu cel exprimat prin miss penalty de la procesul de citire. Prin **write-back** se înscrise numai în memoria cache, urmând ca înscrierea în memoria principală să se realizeze doar atunci când blocul respectiv, din memoria cache în care s-a înscris, este trimis înapoi la memoria principală (aceasta se realizează doar când respectivul bloc este înlocuit în memoria cache de către un alt bloc adus în memoria principală); evident că până la înscrierea în memoria principală conținutul bocului din memoria cache și conținutul blocului de aceeași adresă din memoria principală nu este același, deci conținuturile din cele două locații de aceeași adresă nu sunt consistente!

Dacă la înscriere există miss, adică locația de adresă la care procesorul vrea să înscrive nu a fost adusă în memoria cache, atunci se caută blocul în memoria principală, se înscrise cuvântul respectiv în memoria principală, iar blocul respectiv se transferă și în memoria cache.

Metrica pentru performanța memoriei cache este **hit ratio** definit prin raportul

$$h = \frac{\text{Numarul de accese cu succes la memoria cache}}{\text{Numarul total de accese la memoria cache}}$$

Deoarece h are valori în intervalul 0,9-0,99 este mai atractiv să se lucreze cu termenul **miss rate** care se exprimă prin $(1-h)$; creșterea pentru hit rate de la 0,96 la 0,97 corespunde cu 1%, dar descreșterea pentru miss rate este de la 0,05 la 0,04 adică 20%. Timpul mediu de acces la memoria cache (**nivelul 1, L1**) se calculază cu relația

$$\text{Tacces mediu} = T_{L1}/h$$

în care T_{L1} este timpul (exprimat în tacte de ceas) de acces la nivelul L1 cache ($T_{L1} \geq 1$ tact).

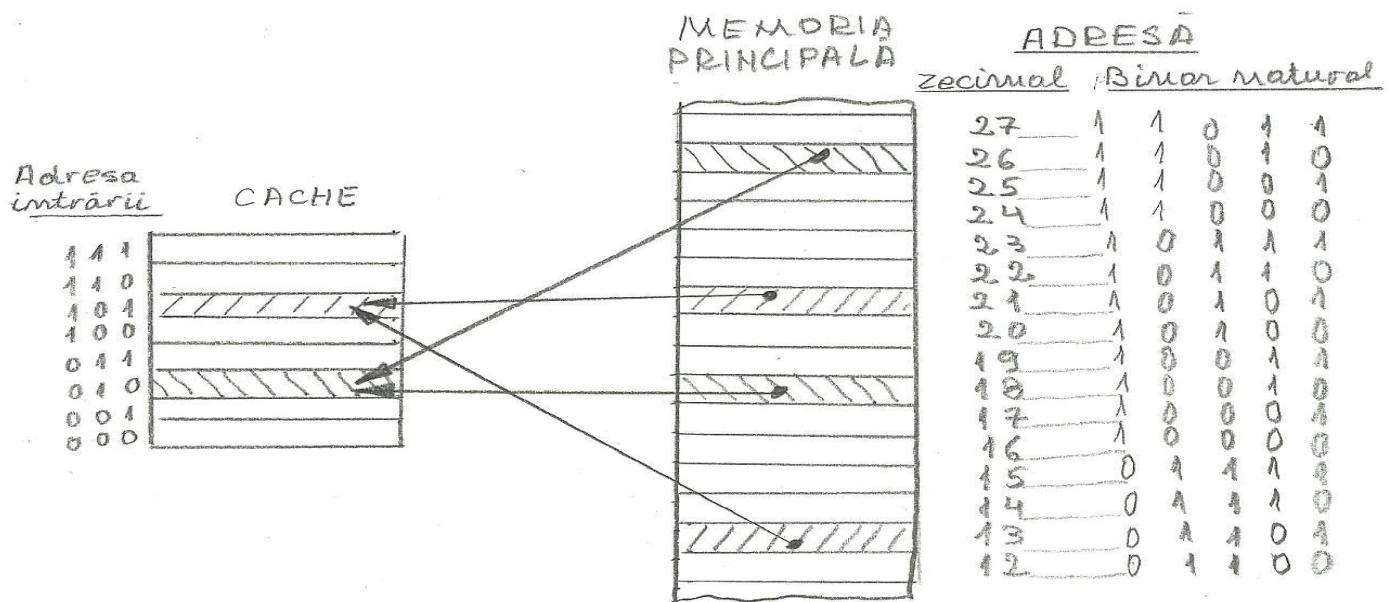
Entitatea de transfer între memoria principală și memoria cache este blocul care poate conține un cuvânt sau mai multe (în general puteri ale lui 2, vezi figura anterioară). Prin transfer se realizează o mapare a unui bloc din memorie, care are o anumită adresă în memorie, pe o adresă de bloc din memoria cache (pe care o să-o referim ca intrare/linie cache sau bloc cache), deci se face o aplicație între cele două mulțimi (mulțimea de adrese ale blocurilor din memorie și mulțimea de intrări din memoria cache); evident că numărul de blocuri din memoria principală este cu mult mai mare decât numărul de intrări ale memoriei cache. Regula după care se face această aplicație determină/definește trei modalități de organizare: memoria cache cu mapare directă, memoria cache asociativă și memoria cache set-asociativă.

1.3.3.1 Memoria cache cu mapare directă.

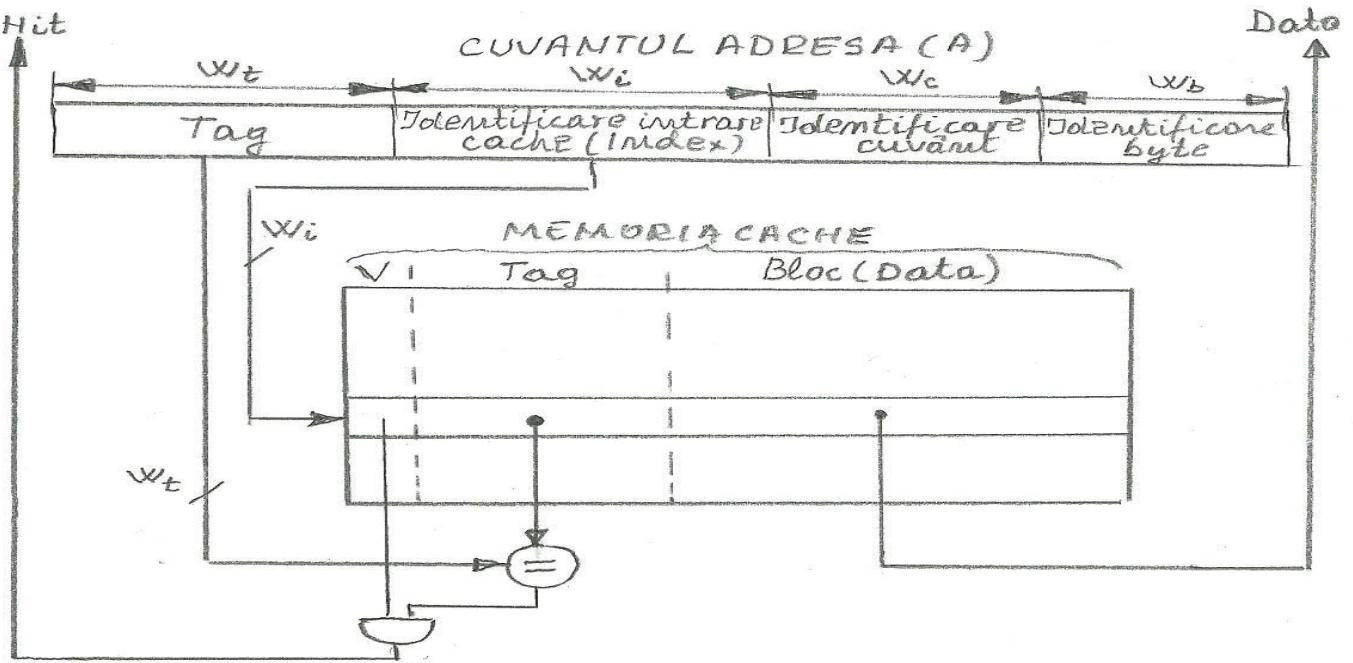
Regula de mapare, pentru o organizare de memorie cache cu mapare directă, permite ca adresa unui bloc din memorie să fie plasată doar într-o singură intrare/linie cache (din numărul total de intrări ale memoriei cache), aplicație ce se exprimă prin relația

(adresă bloc din memorie) modulo (număr intrări cache)

In figura următoare se reprezintă grafic o mapare directă, corespondența dintre adresele blocurilor din memoria principală și adresele intrărilor/blocurilor din memoria cache se determină cu relația anterioară. Se consideră o memorie cache cu 8 intrări (deci în total opt adrese), iar din memoria principală se reprezintă doar adresele din intervalul 12_{10} (01000_2) — 27_{10} (11010_2), lungimea unui bloc (pentru simplitate) se consideră format dintr-un singur cuvânt (deci în acest caz adresa unui bloc din memoria principală corespunde cu adresa unui cuvânt). Se observă că toate adresele de bloc de memorie a căror clasă de resturi este 5_{10} (00101_2) modulo opt se înscriu în intrarea cache de adresă $5_{10} = 101$ (ceea ce corespunde cu ultimii trei biți din adresa blocului din memorie), aceste adrese sunt 13_{10} (01101_2), 21_{10} (10101_2), iar în adresa de intrare cache 2_{10} (00010_2) se plasează toate adresele din memorie care fac parte din clasa de resturi doi modulo opt, adică 18_{10} (10010_2), 26_{10} (11010_2), care au ultimii trei biți 010 (2_{10}) în adresa blocului din memorie.



Pentru accesarea unui cuvânt în memoria cache μP utilizează un cuvânt de adresă A identic cu cel prin care ar accesa respectivul cuvânt în memoria principală, dar pentru cache se folosesc specific anumite subcâmpuri din cuvântul adresă A. Blocul din cache este plasat într-o anumită intrare (deci trebuie selectată respectiva intrare prin **subcâmpul index**), în cadrul blocului trebuie selectat cuvântul (când blocul este format din mai multe cuvinte, deci trebuie să existe un subcâmp **selectare cuvânt**), iar în cadrul cuvântului trebuie selectat oricare byte (de către câmpul **selectare byte**). Dar pentru că aceleași valori de selectare index, cuvânt, byte pot să fie identice pentru mai multe cuvinte din memoria principală, diferențierea în identificarea diferitelor blocuri/cuvinte/bytes se face prin restul de biți din cuvântul adresa A, specificat în subcâmpul referit **tag**. Într-o intrare a memoriei cache, pe lângă blocul plasat se mai înscrie și biții din subcâmpul tag al adresei, iar la accesarea memoriei cache blocul plasat în memoria cache este cel căutat numai când tag-ul din cache este identic cu cel din subcâmpul tag al adresei A. Identificarea între tag-ul înscris într-o intrare/linie a memoriei cache și a tag-ului din cuvântul adresă se face prin intermediul unui circuit comparator, la o identitate a celor două taguri se generează semnalul de **hit** (data va fi transmisă la μP), iar la o neidentitate se generează miss (va trebui accesat nivelul inferior, adică memoria principală, ceea ce implică oprirea și intrarea în stare de aşteptare a μP pe durata a cel puțin 100 de tacte de clock, acest consum de tacte de clock este referit ca **miss penalty**). În fiecare intrare de cache se mai înscrie, pe lângă blocul respectiv și tagul blocului respectiv, încă un bit de validare, V, care pentru valoarea V=1 indică faptul că în intrarea respectivă a fost înscris un bloc, iar pentru V=0 în intrarea respectivă nu este plasat încă un bloc.



Relația între lungimile subcâmpurilor (exprimate în număr de biți) din cuvântul de adresă A, cu lungimea de W_A biți, este:

$$W_A = W_t + W_i + W_c + W_b$$

în care: - $W_i = \lceil \log_2 (\text{numărul de intrări, blocuri în memoria cache}) \rceil$

- $W_c = \lceil \log_2 (\text{numărul de cuvinte/ bloc}) \rceil$

- $W_b = \lceil \log_2 (\text{numărul de byte/ cuvânt}) \rceil$,

iar lungimea tag-ului rezultă

$$- W_t = W_A - W_i - W_c - W_b$$

Un exemplu de structurare a memoriei cache cu mapare directă este prezentat în figura următoare

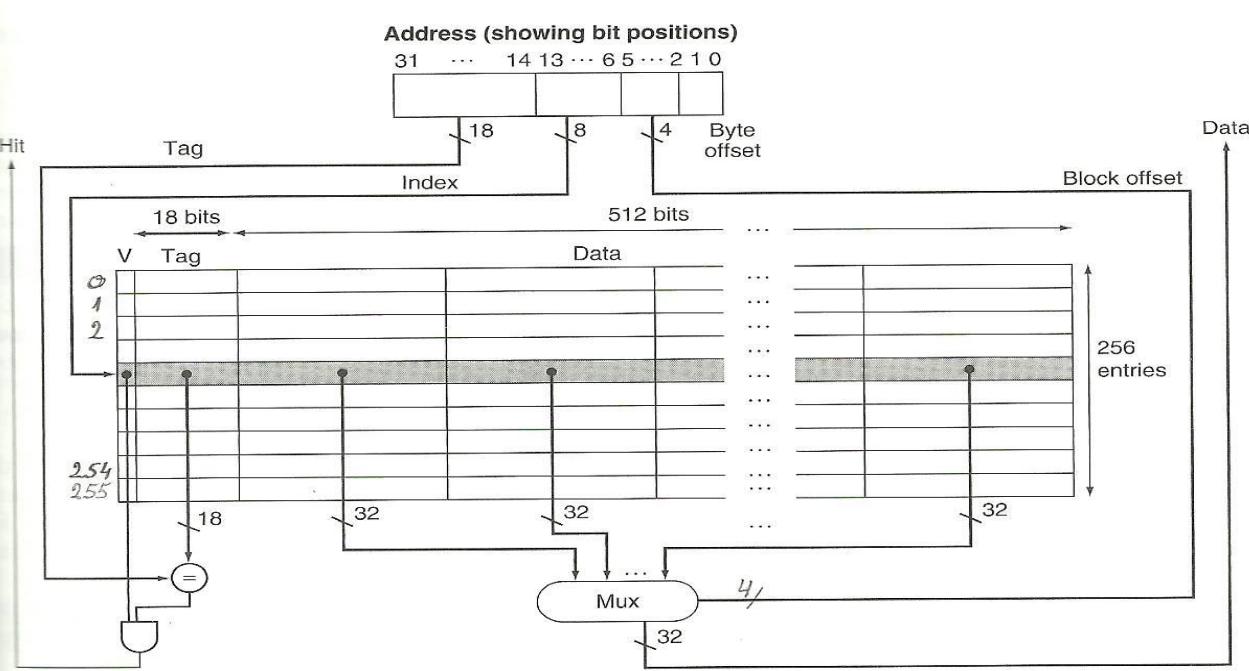


FIGURE 5.9 The 16 KB caches in the Intrinsicity FastMATH each contain 256 blocks with 16 words per block. The tag field is 18 bits wide and the index field is 8 bits wide, while a 4-bit field (bits 5–2) is used to index the block and select the word from the block using a 16-to-1 multiplexor. In practice, to eliminate the multiplexor, caches use a separate large RAM for the data and a smaller RAM for the tags, with the block offset supplying the extra address bits for the large data RAM. In this case, the large RAM is 32 bits wide and must have 16 times as many words as blocks in the cache.

Exemplul 1.2 La un procesor, care lucrează cu cuvinte de 32 biți, există o memorie cache de capacitate Ccache = 16KB ($2^{14}[\text{bytes}]$), cu organizarea de tip mapare directă iar lungimea unui bloc este de 4 cuvinte, adresarea se realizează cu cuvinte de adresă de 32 biți. Să se determine câți biți are capacitatea totală, Ctotal, a memoriei cache.

Soluție. Capacitatea memoriei cache Ccache (numai biți pentru date) se calculează cu relația

$$\text{Ccache[bytes]} = \text{Nr.blocuri} \times \text{Nr.cuvinte/bloc} \times \text{Nr.bytes/cuvânt [bytes]}$$

$$2^{14}[\text{bytes}] = \text{Nr.blocuri} \cdot 2^2 \cdot 2^2 [\text{bytes}] \rightarrow \text{Nr blocuri(intrări)} = 2^{10}$$

$$w_i = \log_2(2^{10}) = 10 \text{biți}$$

$$w_c = \log_2(2^2) = 2 \text{biți}$$

$$w_b = \log_2(2^2) = 2 \text{biți}$$

$$W_t = W_A - W_i - W_c - W_b = 32 - (10+2+2) = 18 \text{ biți}$$

Lungimea unei intrări Li din memoria cache este

$$L_i = 1(\text{valid bit}) + W_t + L_b(\text{lungime bloc}) = 1 + 18 + 4 \times 32 = 147 \text{ biți}$$

Rezultă:

$$C_{\text{total}} = \text{Nr. intrări} \times L_i = 2^{10} \times 147 \text{ biți} = 147 \times 8 = 18,4 \text{KB}$$

În acestă memorie numărul total de biți este de $18,4 \text{KB}/16 \text{KB} = 1,15$ ori mai mult decât sunt necesari pentru stocarea datelor (o regie de $19 \text{biți} \times 2^{10}/8 \times 2^{10} = 2,375 \text{KB}$).

Exemplul 1.3. Se consideră o memorie cache, cu organizare de tip mapare directă, care are 64 de blocuri (intrări) iar lungimea unui bloc este de 16bytes; memoria principală este organizată pe byte. În ce intrare/linie a memoriei cache va fi plasat conținutul locației de memorie cu adresa de 1209.

Soluție. Deoarece lungimea unui bloc, Lb, este de 16 bytes, înseamnă că datele de la 16 locații de adrese consecutive din memoria principală vor intra într-un bloc de memorie cache. Numărul blocului din memoria principală, care conține locația de adresă 1209 este 75 ($1209:16 = 75$ rest 9, numerotarea blocurilor este: 0, 1, 2,...).

Cele 16 locații a căror conținut intră în blocul de memorie cu numărul 75 sunt în intervalul de adrese 1200 - 1215 (care includ adresa 1209).

Aplicând regula de adresare la organizarea de cache cu maparea directă

(adresă bloc din memorie) **modulo** (număr intrări cache)

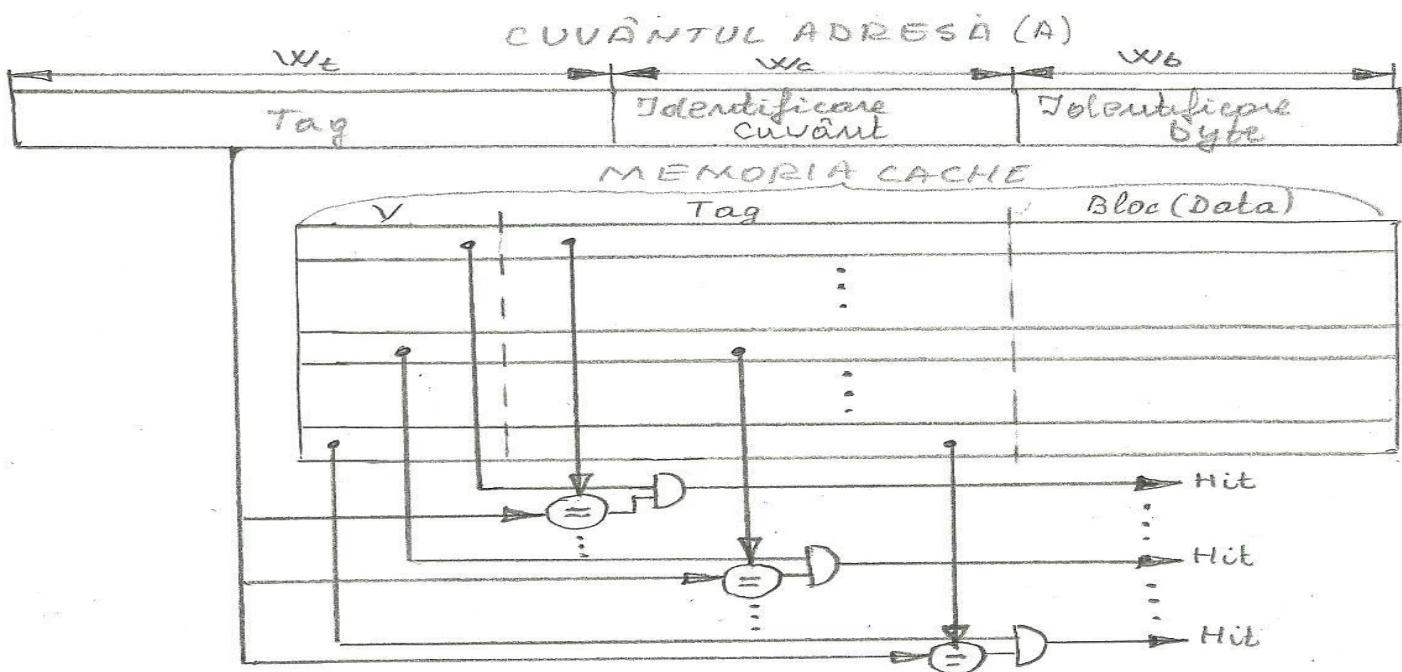
rezultă

$$75 \text{ modulo } 64 \rightarrow 11; 75 = 1x 64 + 11$$

deci în a 11-a linie de intrare a memoriei cache se plasează blocul 75 (16 bytes) al memoriei principale, unde se găsește conținutul byte-ului de adresă 1209.

1.3.3.2 Memoria cache complet asociativă

Memoria complet asociativă (full-asociativă) se situează la extrema opusă în raport cu memoria cu mapare directă. Dacă la memoria cu mapare directă un bloc din memoria principală poate fi mapat doar într-o singură intrare a memoriei cache la cea full-asociativă un bloc din memoria principală poate fi plasat (**asociat**) în oricare intrare din memoria cache. Pentru că blocul de memorie poate fi oriunde în cache, nu mai trebuie căutată intrarea în care se plasează, în cuvântul de adresă nu mai este nevoie de subcâmpul index ($wi = 0$ biți), există doar subcâmpurile selectare cuvânt (wc), selectare byte (wb) și câmpul tag. Tag-ul se aplică la toate intrările, iar operația de compararea simultană a tag-ului cu tag-urile înscrise în toate liniile memoriei cache se realizează în paralel, ceea ce implică existența unui număr de comparatoare egal cu cel al numărului liniilor din memoria cache (în raport cu un singur comparator de la memoria cache cu mapare directă). Un număr mare de comparatoare (egal cu numărul de intrări din memoria cache) duce la creșterea costului unei astfel de organizări, ceea ce face ca memoria cache full-asociativă să fie practică doar pentru un număr mic de blocuri/liniile. În general memoria full-asociativă este realizată pe bază de memorie adresabilă după conținut, **CAM** (Content Addressable Memory).



Memoria full-asociativă având un grad mare de asociativitate poate realiza un miss rate de valoare foarte mică, ceea ce este intuitiv. De exemplu, la memoria cu mapare directă dacă trebuie accesat un bloc care corespunde unei clase de resturi (modulo număr de intrări cache), iar acesta nu se află în memoria cache ci în memoria principală, atunci este transferat blocul din cache (care are aceeași clasă de resturi cu cel căutat) în memorie și blocul din memorie adus în cache; apoi dacă se adresează blocul din memorie care a fost înainte în cache, și care a fost trimis înapoi în memorie, atunci se face din nou swap între memorie și cache cu cele două blocuri (care au

aceeași clasă de resturi). Aceste transferuri, cu valori mari pentru miss penalty, sunt generate de faptul că toate blocurile de aceeași clasă de resturi concură la o singură intrare în cache, deci nu pot exista simultan în cache blocuri de aceeași clasă de resturi. La memoria full-asociativă pot coexista în memoria cache mai multe blocuri de memorie toate având aceeași clasă de resturi (plasate în diferite intrări/linii), deci numărul de situații de miss este mult mai mic!

1.3.3.3. Memoria cache set asociativă

Memoria set asociativă se situează într-o zonă intermediară în raport cu cea full asociativă și cea cu mapare directă. Memoria set asociativă prezintă un număr de seturi, fiecare dintre aceste seturi conține un număr n de intrări/linii care formează o memorie full asociativă cu n intrări, această organizare este referită ca memorie set asociativă cu n -căi (n -way set-associative). Pentru o memorie cache cu un număr total de m intrări, fiecare set având n căi, numărul total de seturi este egal cu raportul m/n . Fiecare set este selectat de subcâmpul index din cuvântul de adresă, la fel ca memoria cu mapare directă; numărul setului din memoria cache set asociativă în care va fi plasat un bloc din memoria principală se determină conform relației

(adresă bloc din memorie) modulo (numărul seturilor din cache)

dar acest bloc din memoria principală poate fi plasat în setul respectiv în oricare din cele n intrări/linii ale setului (mapare asociativă în cadrul setului și nu mapare directă). Un bloc din memorie este mapat direct (numai intr-un set (deci prin index), dar în cadrul setului plasat asociativ (în oricare intrare a setului)). Într-o intrare cache, a unui set, se înscrive tag-ul, blocul data și bitul valid (V). Deoarece în fiecare set cu n căi pot exista n blocuri de date cu aceeași clasă de resturi, modulo (numărul seturilor din cache), miss rate este destul de mic. Găsirea unui bloc dintr-un set se realizează prin intermediul tag-urilor.

Comparând o memorie cache set asociativă cu cele două tipuri anterioare de organizare rezultă că memoria cu mapare directă este o memorie set asociativă cu o singură cale ($n=1$), intr-un set, iar memoria full asociativă cu m intrări este o memorie set asociativă cu un singur set care are n căi, egal cu numărul total de intrări în cache, $n=m$.

În figura următoare este exemplificată maparea blocurilor (blocul este compus doar dintr-un singur cuvânt data) din memoria principală pentru intervalul de adrese 12-27 pe o memorie cache cu 8 intrări ($m=8$) în două variante. În prima variantă memoria cache set asociativă are două căi ($n=2$), deci rezultă patru seturi $m/n=4$ maparea se face conform relației:

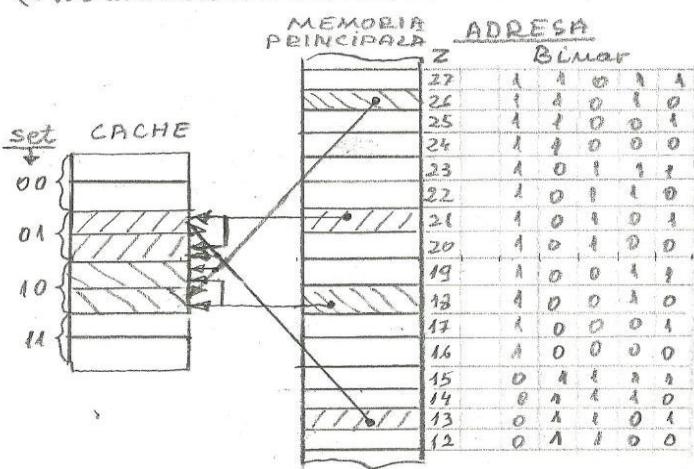
(adresă bloc din memorie) modulo 4

iar în a doua variantă memoria este organizată set asociativă cu patru căi ($n=4$), deci rezultă două seturi $m/n=2$, maparea se face conform relației

(adresă bloc din memorie) modulo 2

2-CĂI SET ASOCIAȚIVĂ

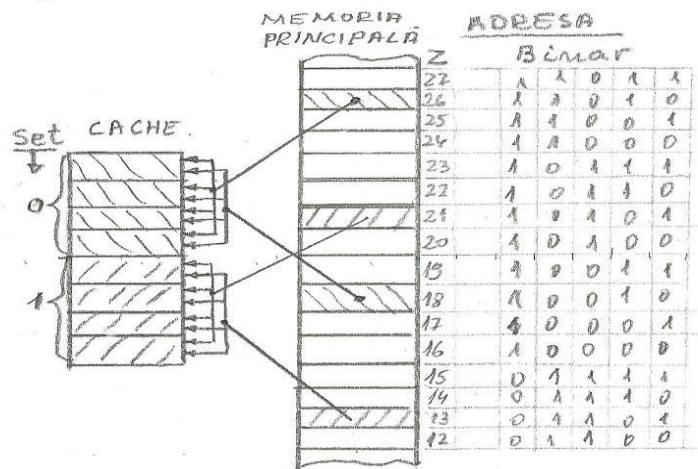
(Bloc adresa memorie) modulo 4



- modulo 4
- bloc = cuvânt date
- 2 comparații

4-CĂI SET ASOCIAȚIVĂ

(Bloc adresa memorie) modulo 2



- modulo 2
- bloc = cuvânt date
- 4 comparații

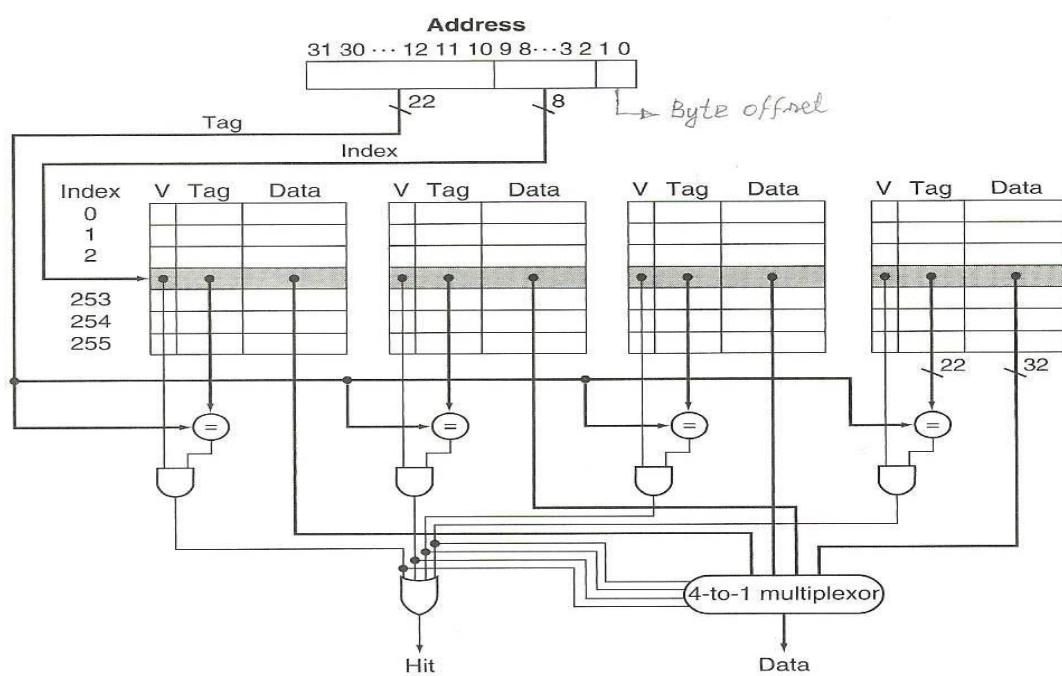


FIGURE 5.17 The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor. The comparators determine which element of the selected set (if any) matches the tag. The output of the comparators is used to select the data from one of the four blocks of the indexed set, using a multiplexor with a decoded select signal. In some implementations, the Output enable signals on the data portions of the cache RAMs can be used to select the entry in the set that drives the output. The Output enable signal comes from the comparators, causing the element that matches to drive the data outputs. This organization eliminates the need for the multiplexor.

Se pune întrebarea firească, cât se reduce miss rate prin creșterea asociativității. Din tabelul următor se poate deduce un astfel de răspuns. Se consideră o memorie cache cu capacitatea de 64KB, cu blocuri de 16 cuvinte, iar lungimea cuvânt este de 32 biți (4 bytes) rezultă numărul de blocuri /linii din memoria cache $64\text{KB}/(16 \times 4)\text{B} = 1\text{K}$ intrări. Prin trecerea de la maparea directă (set asociativ cu o singură cale) la organizarea set asociativă cu 2 căi, miss rate se reduce cu 16,5%, dar trecerea în contiunare la organizarea set asociativă cu 4 căi miss rate se reduce doar cu 3,4% (de la 8,6% la 8,3%). Aceste statistici sunt un argument de ce majoritatea organizărilor actuale de memorii cache sunt de tipul 2 căi set asociative

Associativity	Data miss rate
1	10.3%
2	8.6%
4	8.3%
8	8.1%

FIGURE 5.15 The data cache miss rates for an organization like the Intrinsic FastMATH processor for SPEC2000 benchmarks with associativity varying from one-way to eight-way. These results for 10 SPEC2000 programs are from Hennessy and Patterson [2003].

Alegerea între organizările maparea-directă, set asociativă și full asociativă, în oricare nivel de memorie cache, depinde de costul miss rate în raport cu costul asociativității atât din punct de vedere al timpului (timp de acces și miss penalty) cât și al (costului) implementării de hardware suplimentar.

1.3.3.4. Memoria cache multinivel

Pentru a micșora și mai mult gap-ul dintre frecvența de ceas foarte ridicată și timpul lung de acces la memoria SDRAM procesoarelor moderne au pe lângă memoria cache L1 încă un nivel de memorie cache L2, ambele integrate pe același chip cu μ P; capacitatea nivelului L2 este mai mare decât capacitatea nivelului L1 cam de un ordin de mărime (uzual L1: L2 = 50:1). Nivelul L2 este accesat totdeauna când se produce un miss pe L1. Dacă L2 conține date căutată atunci miss penalty este egal cu timpul de acces la L2, care este cu mult mai mic decât timpul de acces la memorie. Dar dacă date căutată nu se află nici în L1 nici în L2 atunci se accesează memoria rezultând o penalizare foarte mare. În proiectarea unei memorii cache pe două niveluri se caută realizarea nivelului L1 cu un timp de acces (hit time) cât mai mic, printr-o capacitate redusă și blocuri de dimensiuni reduse, iar pentru nivelul L2 o capacitate mai mare (pentru că timpul de acces este mai puțin critic decât la L1) și cu o mai mare asociativitate pentru a se micșora miss rate (în scopul reducerii numărului accesărilor la memorie, care introduc penalizări mari).

Ultimile procesoare (pentru desk-top, servere) au deja integrat pe chip și al treilea nivel de cache, L3, situat între L2 și memorie, cu o capacitate mai mare decât L2, uzual L3:L2 = 5:1

O atenție deosebită în optimizarea nivelurilor L2 și L3 trebuie îndreptată asupra algoritmului de înlocuire a unui bloc dintr-un nivel cache când se aduce un bloc nou din nivelul inferior; cel mai simplu și des utilizat este algoritmul referit LRU (Least-Recently Used), adică se substituie blocul care n-a mai fost accesat de cel mai lung timp (utilizat cel mai puțin în ultimul timp).

În tabelul următor sunt prezentate valori pentru parametrii nivelurilor de cache ale procesoarelor IBM Power P4(2001) și IBM Power P5(2004), acestea sunt procesoare multichip două procesoare identice pe același chip (dual-core).

Nivel cache	Capacitate	Asociativitate	Dimensiune Bloc/linie (byte)	Modul de înscriere	Latentă (în tacte)	
					P4(1,7GHz)	P5(1,9GHz)
L1 I-cache	64KB	Direct/2-căi	128	Instr. nu se înscriv	1	1
L1 D-cache	32KB	2-căi/4-căi	128	Write-through	1	1
L2 (D+I unificate)	1,5MB/2MB	8-căi/10-căi	128	Writeback	12	13
L3	32MB/36MB	8-căi/12-căi	512	Writeback	123	87
Memorie / referă valori pentru P4 respectiv P5					351	220

I-cache (memorie cache numai pentru stocarea de cuvinte instrucțiune)

D-cache (memorie cache numai pentru stocarea de cuvinte date)

Exemplul 1.4 Se consideră un sistem cu μ P cu frecvența de ceas de $f_{clk} = 4\text{GHz}$ ($T_{clk} = 0,25\text{ns}$) la care timpul de acces la nivelul L1 cache T_{ACL1} este un tact ($T_{ACL1} = 1 \text{ tact} = 0,25\text{ns}$) și timpul de acces la memorie SDRAM este $T_{ACm} = 100\text{ns}$ ($100\text{ns} / (0,25\text{ns/tact}) = 400 \text{ tacte}$). Cand toate accesările la memoria cache L1 sunt rezultate cu hit, (miss rate este 0) sistemul execută pe fiecare tact de ceas o instrucțiune, CPI=1 (Cycles Per Instruction)

1. Cât este valoarea pentru CPItotal, când se presupune că miss rate pentru nivelul unu de cache nu este zero ci 2%.

Soluție.

Numărul total de tacte pe instrucțiune , CPI total, consumat de procesor pentru o instrucțiune este egal cu CPI = 1 plus numărul de tacte de aşteptare pentru μ P când a rezultat miss pe nivelul L1 și se accesează memoria SDRAM(miss penalty)

$$\text{CPI total} = 1 + 2\% \times 400 = 9$$

2. Cu cât va crește viteza sistemului dacă se introduce și al doilea nivel (L2) de cache, care are timpul de acces de $T_{ACL2} = 5\text{ns}$ ($5\text{ns} / (0,25\text{ns/tact}) = 20 \text{ tacte}$) iar mis rate este de 0,5% ?

Soluție

- Dacă rezultă miss pe nivelul L1 atunci se accesează L2, iar pe acest nivel este numai hit la toate accesările atunci miss penalty este egal cu 20 de tacte

$$\begin{aligned}\text{CPI total} &= 1 + \text{numărul de tacte de aşteptare introduse de accesul la L2} = \\ &= 1 + 2\% \times 20 = 1 + 0,4 = 1,4\end{aligned}$$

- Dacă se produce miss pe nivelul L1, dacă se produce miss și pe nivelul L2 atunci se accesează SDRAM unde se găsește blocul căutat (cu o penalizare de 400tacte)

$$\text{CPI total} = 1 + (2\% - 0,5\%) 20 + 0,5\% \times (20 + 400) = 1 + 0,3 + 2,1 = 3,4$$

Procesorul care are o memorie cache pe două niveluri cu datele calculate anterior obține un spor de viteză de $9/3,4 = 2,6$.

1.4. PERIFERICELE (I/O)

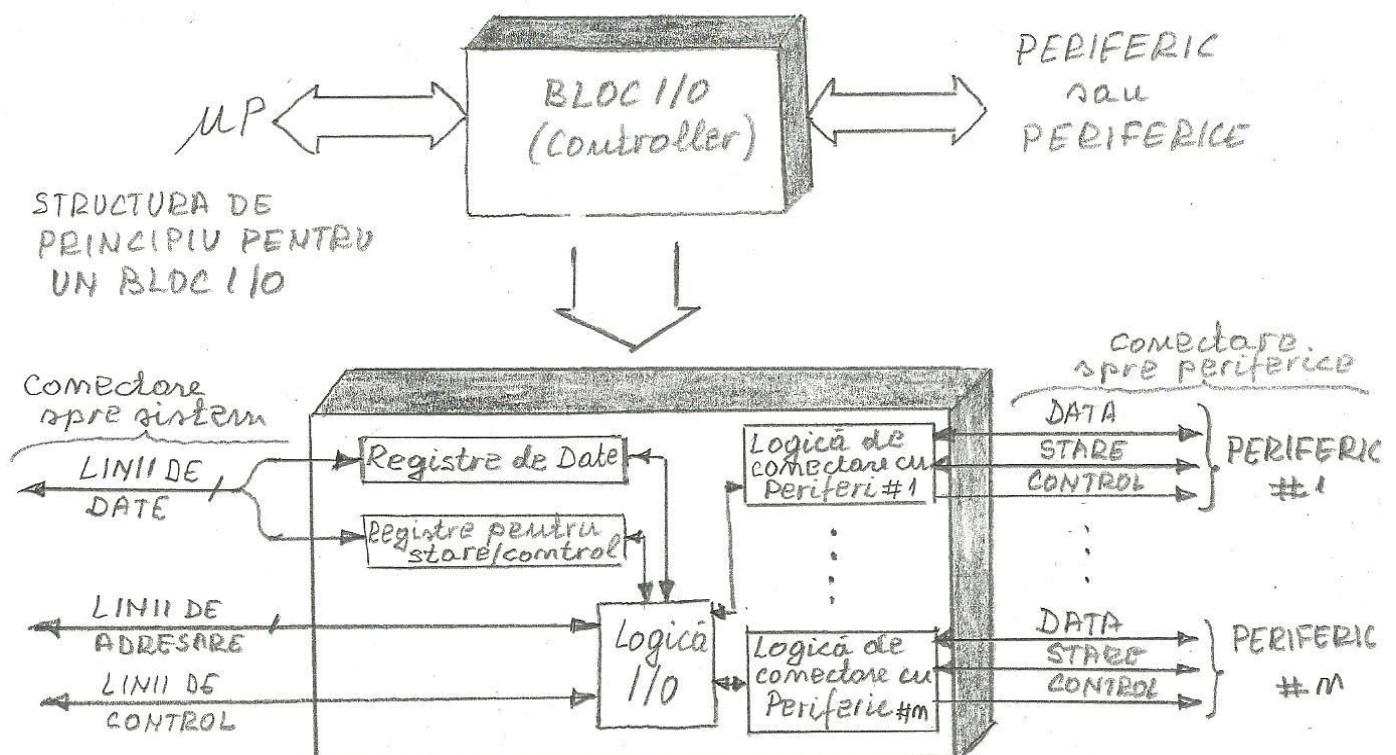
Elementele externe (periferice) care se pot conecta la μ P sunt foarte variate și prezintă o dispersie largă a caracteristicilor (tip de periferic, partener , rată de transfer (viteză/rată de transfer)), unele dintre acestea sunt prezentate în tabelul următor

Device	Behavior	Partner	Data rate (Mbit/sec)
Keyboard	Input	Human	0.0001
Mouse	Input	Human	0.0038
Voice input	Input	Human	0.2640
Sound input	Input	Machine	3.0000
Scanner	Input	Human	3.2000
Voice output	Output	Human	0.2640
Sound output	Output	Human	8.0000
Laser printer	Output	Human	3.2000
Graphics display	Output	Human	800.0000–8000.0000
Cable modem	Input or output	Machine	0.1280–6.0000
Network/LAN	Input or output	Machine	100.0000–10000.0000
Network/wireless LAN	Input or output	Machine	11.0000–54.0000
Optical disk	Storage	Machine	80.0000–220.0000
Magnetic tape	Storage	Machine	5.0000–120.0000
Flash memory	Storage	Machine	32.0000–200.0000
Magnetic disk	Storage	Machine	800.0000–3000.0000

FIGURE 6.2 The diversity of I/O devices. I/O devices can be distinguished by whether they serve as input, output, or storage devices; their communication partner (people or other computers); and their peak communication rates. The data rates span eight orders of magnitude. Note that a network can be an input or an output device, but cannot be used for storage. Transfer rates for devices are always quoted in base 10, so that 10 Mbit/sec = 10,000,000 bits/sec.

Raportul între rata de transfer cea mai ridicată 10^4 Mbit/s (network /LAN) și rata de transfer cea mai mică 1.10^{-4} Mbit/s (tastatură) este de 10^8

- Un periferic ca să poată fi conectat într-un sistem pe bază de μ P trebuie să fie adaptat (cerințe mecanice (cuplaje), mod de comunicare pe intrare și iesire (I/O), niveluri de semnale) la modul de lucru al procesorului, această adaptare se realizează printr-o circuistică de interfațare referită: bloc de I/O, I/O, controller I/O sau simplu controller. Controllerul, fizic, poate fi inclus în periferic sau poate fi inclus în sistemul cu μ P. Procesorul nu "vede" perifericul ci doar componenta I/O (controllerul). Fiecare bloc de I/O (adică periferic) are o adresă prin care este înregistrat în sistem și pe baza căreia este recunoscut de μ P. O structură bloc (generică) pentru I/O este schițată în figura următoare.

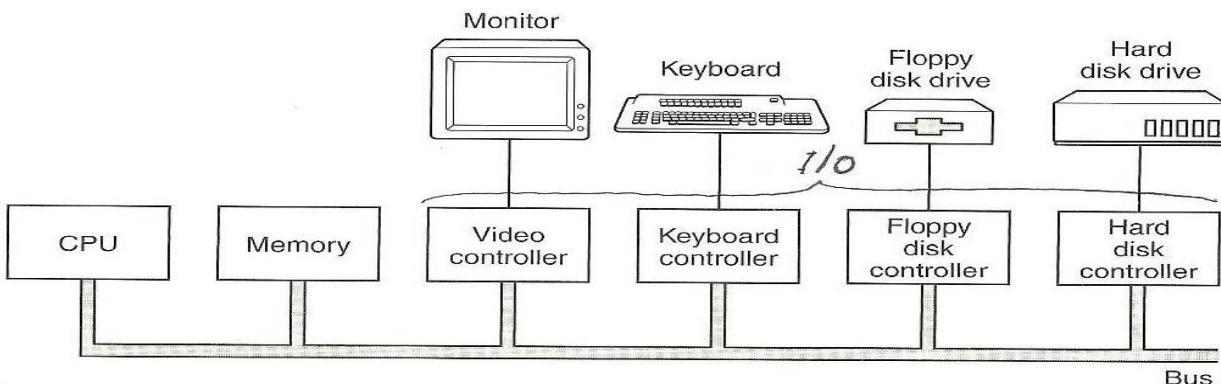
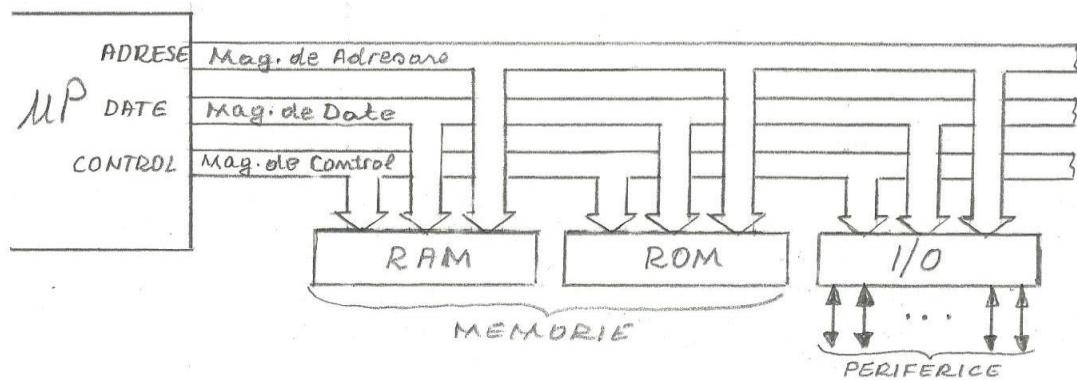


1.4 ORGANIZAREA UNUI SISTEM PE BAZĂ DE MAGIASTRALE

1.5.1 Sistemul cu trei magistrale

Magistrală – o cale de comunicație care utilizează un set de linii pentru conectarea diferitelor părți ale sistemului. Lățimea magistralei (w) este egală cu numărul de linii, în general numărul de linii sunt multiplu de byte (8 biți).

- Organizare de principiu



Logical structure of a simple personal computer.

1. Magistrale de date. Cuvintele DATA, în general, cu lungimea de multiplu de byte sunt transferate între μP, memorie și I/O pe magistrala de date. Cuvintele data pot fi instrucțiuni transferate între μP și memorie sau cuvinte data transferate între μP, memorie și I/O. Uneori pentru cuvintele lungi (mai mari dacă lățimea magistralei se fac două sau trei transferuri succesive pe magistrală). Magistrala de date este cu transfer bidirectional, adică μP poate fi atât sursă de date cât și destinație.

2. Magistrala de adresa. Este suportul pentru transmisia cuvintelor ADRESĂ. În general, această magistrală este unidirecțională, adică se transmit cuvinte adresă într-un singur sens, de la μP la memorie sau la I/O. Există situația când se transmit adrese și de la alte componente ale sistemului, de exemplu funcționare în DMA (Direct Memory Access). Pentru micșorarea numărului de pini μP și pentru creșterea numărul de operații în unitatea de timp (Throughput), magistrala de date este, uneori, multiplexată între transmiterea cuvintelor ADRESĂ și a cuvintelor DATA. Această multiplexare, Adresă/Date, este posibilă în timp deoarece intervalele de comunicație DATA și cele de comunicații ADRESA sunt disjuncte (nu se suprapun în timp).

3. Magistrala de control. Se înțelege prin magistrală toate liniile prin care sunt transmise semnalele pentru sincronizarea operațiilor în sistem, adică pentru a indica ce fel de informație există pe magistrala de date și ce operație se execută. Următoarele patru semnale de control generate de μP sunt cele mai uzuale:

- citire memorie, MEM_L
- înscriere memorie, MEMW_L
- înscriere I/O, I/OW_L
- citire I/O, I/OR_L

Există semnale de control generate și de alte elemente din sistem, de exemplu: TRANSFER ACK – confirmare (Acknowledgement) că o comandă a fost acceptată; WAIT- așteaptă până când memoria este citită; IRQ – cerere de intrerupere etc.

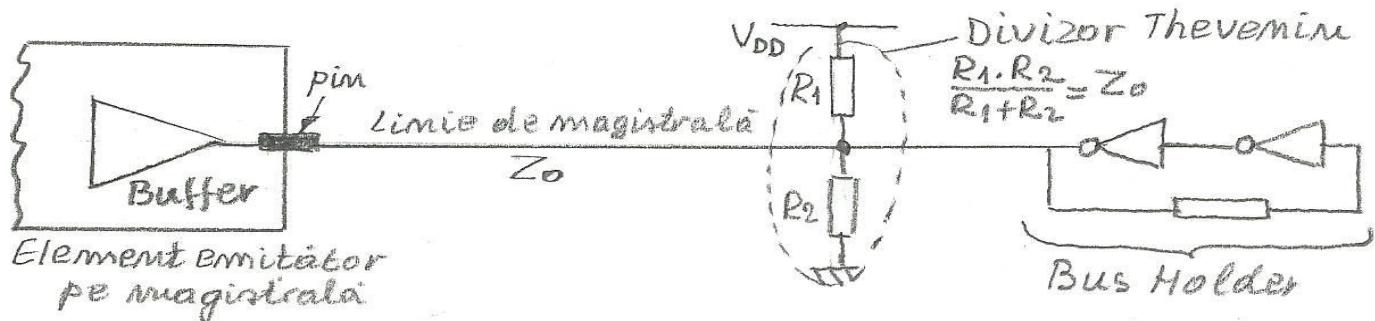
1.5.2. Caracteristicile magistralei

1. Latență – exprimă întârzierea de propagare pe magistrală, aceasta se reflectă în frecvență maximă de ceas, **f_{max}**, cu care se poate comanda transferul sau frecvența maximă a semnalului aplicat pe magistrală. Întârzierea (latență) pe o linie magistrală este determinată de:

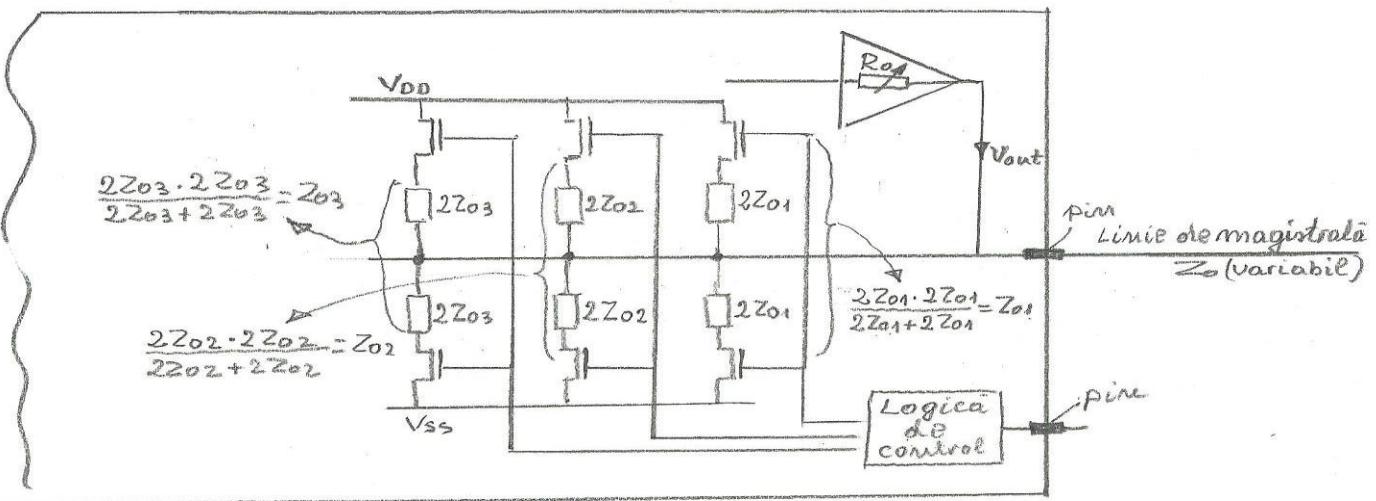
- constanta de timp, $T = RC$ (produsul dintre rezistență și capacitatea liniei);
- încărcarea liniei (numărul de sarcini comandate de elementul care generează semnalul pe linia de mag.);
- adaptarea liniei (atenuarea reflexiilor la capete).

Procesoarele actuale funcționează cu o frecvență interioară de ceas, f_{ckl}, și o altă frecvență exterioară pentru controlul sincronizării transferului pe magistrală, f_{cklmag}, care se obține prin divizarea frecvenței interioare. De exemplu, microprocesorul Pentium 4 are f_{ckl}= 2,66GHz și f_{cklmag}= 133MHz ($1,33 \times 20 = 2,66$).

O linie de magistrală nu se lasă niciodată în gol la capete, se adaptează printr-un divizor Thevenin, iar pentru ca atunci când linia nu este comandanță ca să nu se fixeze nivelul de tensiune la o valoare de potențial între ΔV_H și ΔV_L (care ar distruge porțile receptoare conectate la linie) se conecteză un bus holder.



Memoriile actuale, de tip DDR SDRAM sau cele QDR SRAM la care transferul pe magistrală se realizează la frecvențe de peste 500MHz, conectarea la linia de magistrală se realizează prin circuite care au posibilitatea de adaptare la valoarea impedanței caracteristice a liniei, Z_0 , atât la emisie cât și la recepție. În funcție de valoarea variabilă Z_0 a liniei, la elementul conectat la magistrală, se modifică rezistența de ieșire a bufferului de ieșire, R_o , încât să se obțină egalitatea $R_o = Z_0$ (pentru funcționarea ca generator pe magistrală) sau egalitatea $(2Z_{oi} \cdot 2Z_{oi}) / (2Z_{oi} + 2Z_{oi}) = Z_{oi} = Z_0$ (pentru funcționarea ca receptor pe magistrală), structura de principiu a unui astfel de circuit de conectare la magistrală este prezentată în figura următoare.



2. Lărgimea de bandă, B (Bandwidth). Se definește ca produsul dintre lățimea magistralei w (exprimată în bytes sau biți) și frecvența maximă admisă pentru comanda magistralei fmax (MHz, sau 1/s)

$$B = f_{\text{max}} \cdot w \quad [\text{MB/s, Mbit/s}]$$

Lărgimea de bandă exprimă transferul maxim pe magistrala respectivă. În general, pe o magistrală rata de transfer este mai mică decât B.

3.Throughput- exprimă numărul de operații în unitatea de timp; pentru magistrală se reduce la numărul de tranzacții în unitatea de timp, adică numărul de operații de citire/s sau înscriere/s.

Numărul de operații în unitatea de timp poate fi mărit printr-o multiplexarea a magistralei, adică realizând un transfer întrețesut pe magistrală. De exemplu, la accesarea pentru citire a unei memorii cu latență mare, după ce s-a aplicat adresa la memorie, pe intervalul de așteptare până la obținerea datelor (timpul de acces), magistrala poate fi utilizată pentru alte tranzacții.

Exemplul 1.5 Un procesor este conectat la o memorie printr-o magistrală multiplexată (date+adresa) de lățime w = 32 biți, comandată cu frecvență maximă de sincronizare $f_{\text{clkMag}} = 100\text{MHz}$ ($T_{\text{clkMag}} = 10\text{ns}$) . Adresarea memoriei pentru citire se face cu un cuvânt ADRESA de 32 biți (1w), cuvântul DATA este de 64 biți (2w), iar latența memoriei este de 50ns. Transferul cuvântului ADRESA pe magistrală spre memorie consumă un tact, iar transferul cuvântului DATA de la memorie spre procesor două tacte. Să se determine: lățimea de bandă, numărul de citiri în unitatea de timp/ throughput, rata de transfer (tranzacții) și gradul de utilizare al magistralei.

Soluție.

– Lărgimea de bandă, $B = w \cdot f_{\text{clkMag}} = 4\text{byte} \cdot 100\text{M} [1/\text{s}] = 400\text{MB/s}$

– Latența memoriei $50\text{ns}/(10\text{ns/tact}) = 5$ tacte

– Durata unei tranzacții (citire)

$$\begin{aligned} T_{\text{transz}} &= 1\text{tact (adresare)} + 5\text{tacte (acces memorie)} + 2\text{ tacte (transfer DATA)} = \\ &= 8\text{tacte} \rightarrow 80\text{ns/tranzactie(citire)} \end{aligned}$$

$$- \text{Troughput} = 1/(80\text{ns/tranzacție}) = 12,5 \cdot 10^6 \text{ citiri/s} = 12,5 \text{ Mcitiri/s}$$

$$- \text{Rata de transfer} = 12,5 \cdot 10^6 \text{ citiri/s} \times 64\text{biți/citire} = 100\text{MB/s}$$

$$- \text{Gradul de utilizare al magistralei } 3\text{tacte}/8\text{tacte} \cdot 100 = 37,5\% \quad (5 \text{ tacte se așteaptă citirea memoriei})$$

În cazul în care lungimea cuvântului DATA este de 128biți rezultă:

$$\begin{aligned} - T_{\text{transz}} &= 1\text{tact (adresare)} + 5\text{tacte (acces memorie)} + 4 \text{ tacte (transfer DATA)} = \\ &= 10\text{tacte} \rightarrow 100\text{ns/tranzactie} \end{aligned}$$

$$- \text{Troughput} = 1/(100\text{ns/tranzacție}) = 10 \cdot 10^6 \text{ citiri/s} = 10\text{Mcitiri/s}$$

$$- \text{Rata de transfer} = 10 \cdot 10^6 \text{ citiri/s} \times 128\text{biți/citire} = 160\text{MB/s}$$

$$- \text{Gradul de utilizare al magistralei } 5\text{tacte}/10\text{tacte} \cdot 100 = 50\%$$

iar dacă și lățimea magistralei devine $w = 128$ biți (cuvântul de date fiind tot 128 biți)

- Lărgimea de bandă, $B = w \cdot f_{clk} \cdot 100M [1/s] = 1600MB/s$
 - Ttranz = 1tact (adresare) + 5tacte (acces memorie) + 1 tacte (transfer DATA) =
= 7tacte → 70ns/tranzacție
 - Troughput = $1/(70ns/tranzacție) = 14,29 \cdot 10^6$ citiri/s = 14,29Mcitiri/s
 - Rata de transfer = $14,29 \cdot 10^6$ citiri/s x 128biți/citire = 228,64MB/s
 - Gradul de utilizare al magistralei 2tacte/7tacte · 100 = 28,5%
-

Magistralele paralele prezentate au avantajul că sunt structuri de transmiterea datelor realizabile la un preț scăzut și cu mare versatilitate în organizare, dar au și dezavantajele:

- pot apărea stangulări în comunicație (limitare din cauza lărgimii de bandă, rata de transfer necesară este mai mare decât B);
- latența limitează viteza de transfer;
- apariția defazajului de ceas (la lungimi mari ale linilor de magistrală)

în plus, dacă în sistem există mai multe elemente care pot comanda magistrala (masteri) atunci trebuie introdus un sistem de arbitrage pentru primirea dreptului de a comanda magistrala. Toate aceste dezavantaje a determinat tendința de utilizare a magistralelor seriale sau conexiunile punct-la-punct între elementele sistemului.

1.5.3 Magistrale sincrone

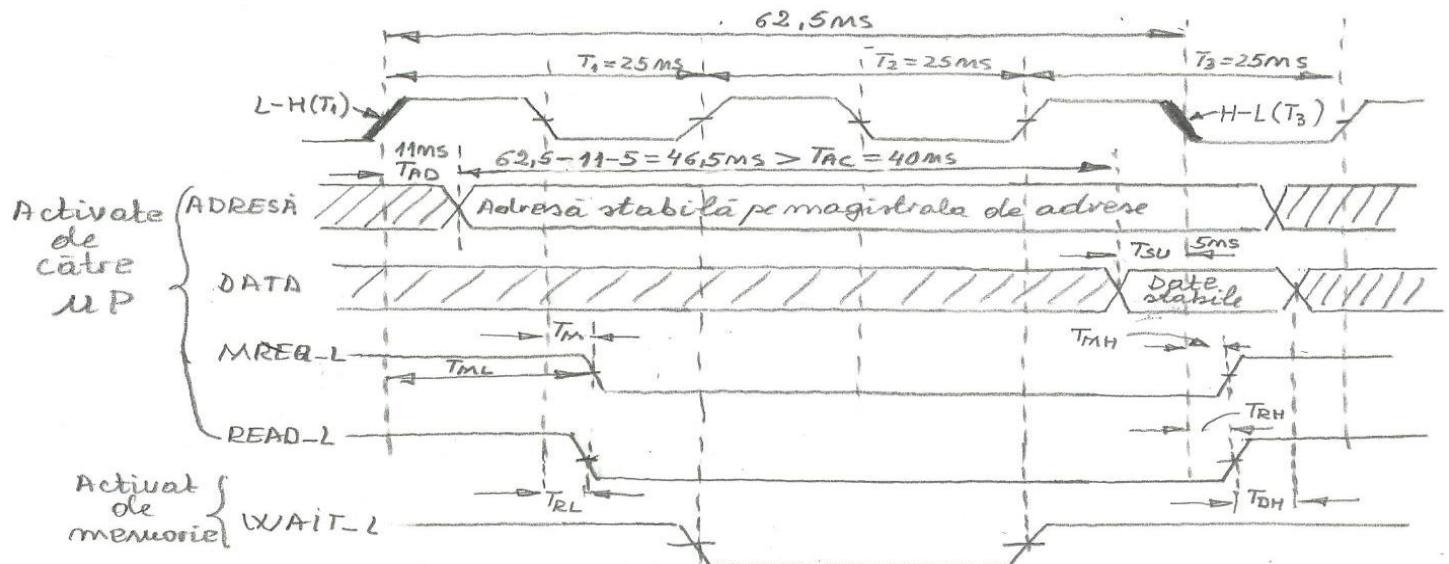
La o magistrală sincronă toate tranzacțiile se realizează sub controlul semnalului de ceas, adică sunt referite la fronturile H-L sau L-H ale acestui semnal. Această ”încadrare” a tranzacțiilor în cadența semnalului de ceas, care este dat, impune ca toate elementele care participă în tranzacția de pe magistrală să aibă parametrii de timp constanti. Astfel de condiții sunt îndeplinite de μP , memorie, display-uri grafice, ceea ce explică faptul că aceste elemente sunt conectate prin magistrale sincrone. Chiar dacă unele elemente sunt mai rapide decât perioada ceasului aceste elemente trebuie să se supună (să aștepte) fronturile perioadei semnalului de ceas. Toată succesiunea desfășurării tranzacției, cu un timing fixat, poate fi dirijată (producerea semnalelor de control) de către un ASM sau de către un μP .

Exemplul 1.6 Să se reprezinte succesiunea semnalelor necesare pentru operația de citire a unei memorii cu timpul de acces $T_{AC}=40$ ns. Frecvența ceasului de sincronizare pe magistrală $f_{clk}=40$ MHz ($T_{clk}=25$ ns); aplicarea adresei pe magistrala de adresare de către μP se face pe frontul L-H, înscrierea datelor de pe magistrala de date într-un registru al μP se face pe frontul H-L ale semnalului de ceas, iar restricțiile de temporizare pentru memorie (extrase din foia de catalog a memoriei) sunt cele date în tabelul următor.

Symbol	Parameter	Min	Max	Unit
T_{AD}	Address output delay		11	nsec
T_{ML}	Address stable prior to \overline{MREQ}	6		nsec
T_M	\overline{MREQ} delay from falling edge of Φ in T_1		8	nsec
T_{RL}	RD delay from falling edge of Φ in T_1		8	nsec
T_{TSU}	Data setup time prior to falling edge of Φ	5		nsec
T_{MH}	$MREQ$ delay from falling edge of Φ in T_3		8	nsec
T_{RH}	\overline{RD} delay from falling edge of Φ in T_3		8	nsec
T_{DH}	Data hold time from negation of \overline{RD}	0		nsec

Soluție.

Diagramele de temporizare la citirea memoriei, cu datele de timp din tabelul anterior, sunt prezentate în figura următoare.



Pe frontal L-H(T_1) se aplică ADRESA pe magistrala de adrese, iar citirea datelor stabilă DATA de pe magistrala de date se face pe un front H-L, dar intervalul între cele două fronturi nu poate fi mai mic decât T_{AD} (timpul de stabilire a cuvântului ADRESA + TAC(timpul de acces al memoriei) + T_{SU} (timpul de stabilizare a cuvântului DATA înainte de citire de către μP pe frontul H-L) = $11 + 40 + 5 = 56$ ns, deci citirea nu poate fi efectuată pe frontul H-L din perioada T_2 (56 ns $> 37,5$ ns = $25 + 12,5$) ci numai pe H-L(T_3), adică după $2 \times 25 + 12,5 = 62,5$ ns > 56 ns. Deoarece memoria nu poate produce DATA pe H-L(T_2) trebuie să informeze μP că este necesar să se consume un timp de aşteptare prin activarea (la începutul perioadei T_2) semnalul de WAIT, în cazul aceasta aşteptarea este de un tact (T_2) (în cazul general memoria ține activ semnalul WAIT atâtea tacte cât este necesar ca să genereze date valide care să fie citite de procesor).

Dacă memoria ar fi avut $T_{AC} = 50$ ns atunci $T_{AD} + T_{AC} + T_{SU} = 11 + 50 + 5 = 66$ ns $> 62,5$ ns ceea ce înseamnă că următorul front H-L pe care se pot citi date stabilă DATA de pe magistrala va fi H-L(T_4) la $87,5$ ns, deci semnalul WAIT trebuie activat pentru tactele $T_2 + T_3$.

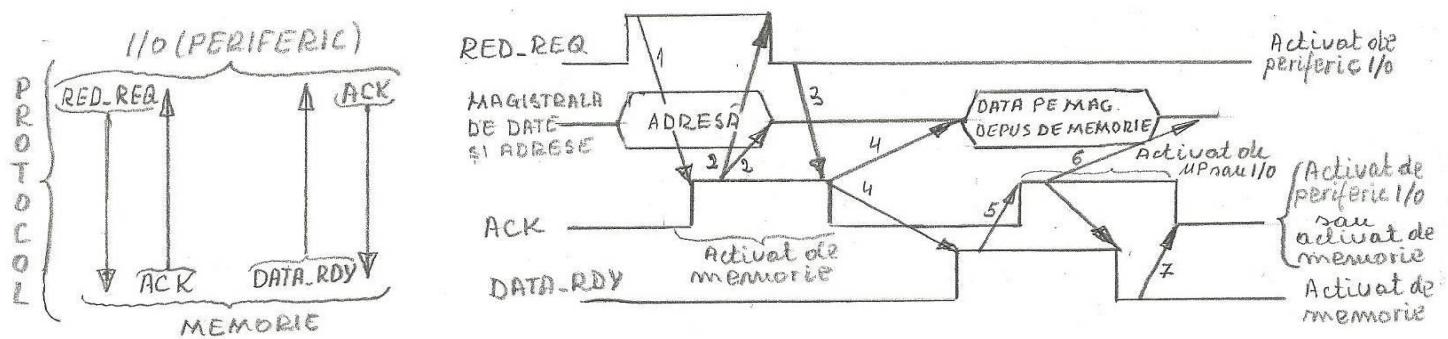
Succesiunea semnalelor de control (aplicare cuvânt ADRESA, MEMQ_L, READ_L) este realizată de către μP , dar poate fi generată și de către un ASM (controller) acesta este cazul când memoria este introdusă într-o aplicație fără μP .

1.5.4 Magistrale asincrone

Într-o magistrală cu funcționare asincronă sunt conectate elemente care prezintă caracteristici de viteză foarte diferite, mai mult valorile acestor caracteristici pot varia în timp (deci astfel de elemente nu pot fi reunite pe o magistrală sincronă unde timingul este foarte rigid, fixat/încadrat prin semnalul de ceas). În consecință, realizarea unei tranzacții pe o magistrală asincronă nu este "pilotată" de semnalul de ceas (lipsește ceasul), iar lungimea magistralei poate fi mult mai mare decât la una sincronă. Transferul pe magistrală se efectuează sub forma unui protocol, între cei doi parteneri implicați, care constă într-un proces de informare-confirmare, adică fiecare dintre parteneri determină (prin reacție) o cauză pentru celălat; trecerea la etapa următoare a transferului se realizează numai atunci când atât partenerul emițător cât și partenerul receptor agerează aceasta.

Exemplul 1.7. Se consideră că un dispozitiv I/O solicită citirea unui cuvânt din memorie. Pentru realizarea protocolului de citire din memorie sunt necesare următoarele trei semnale de control (acestea constituie magistrala de control):

1. Cerere citire, READ_REQ – prin activarea acestui semnal de către I/O se solicită memoriei o citire de un cuvânt DATA, simultan cu activarea acestui semnal dispozitivul I/O aplică și cuvântul ADRESA pe magistrala de adresare (aceeași magistrală este multiplexată între ADRESĂ și DATA). Elementul I/O permanet testeză starea semnalului DATA_RDY, activat de I/O.
2. Date disponibile, DATA_RDY – activarea acestui semnal indică faptul că pe magistrală a fost pus un cuvânt DATA; la o operație de citire DATA_RDY este activat de către memorie, iar la una de înscriere este activat de I/O. Memoria permanet testeză starea semnalului READ_REQ.
3. Confirmare ACK – la sesizarea activării semnalului READ_REQ (de către memorie) sau a semnalului DATA_RDY (de către I/O), celălalt partener implicat în protocol răspunde prin activarea semnalului ACK



Următorii şapte paşi ai protocolului încep după ce I/O a depus pe magistrala de date cuvântul ADRESA și informeză memoria prin activarea semnalului READ_REQ = 1 (informare).

1. Memoria testând permanent linia de semnal READ_REQ (comandată de I/O) constată că a fost activată, va citi cuvântul ADRESA și confirmă aceasta dispozitivului I/O prin activarea liniei de semnal ACK (confirmare).
2. I/O sesizând activarea semnalului de confirmare, ACK = 1, dezactivează semnalul READ_REQ și eliberează magistrala.
3. Memoria sesizând că READ_REQ = 0 (a fost dezactivat) anulează semnalul de confirmare, ACK = 0.
4. În continuare este o stare de aşteptare oricât de lungă (neîncadrată într-un număr de tacte de ceas ca la o tranzacție de pe o magistrală sincronă) până când memoria are cuvântul DATA disponibil, atunci îl pune pe

magistrala de date și informeză I/O prin activarea semnalul DATA_RDY (că se poate citi cuvântul DATA). Permanet I/O testează starea semnalului DATA_RDY, iar memoria testează starea semnalului READ_REQ.

5. I/O sesizând DATA_RDY= 1, citește cuvântul DATA și confirmă aceasta prin activarea semnalului ACK.
 6. Memoria primind confirmarea citirii, ACK =1, dezactivează semnalul DATA_RDY= 0 și elibereză magistrala .
 7. În final , I/O sesizând DATA_RDY= 0, anulează confirmarea, ACK=0, ceea ce indică faptul că tranzacția s-a încheiat. Poate începe o altă tranzacție pe magistrală.
-

1.5.5 Structurarea ierarhizată a magistralelor unui sistem

- Într-un sistem coexistă elemente care necesită viteza de transfer foarte ridicată (μ P, cache, componente grafice) cu elemente foarte lente (tastatură, mouse etc.). Pentru conectarea lor împreună soluția este să se realizeze magistrale cu valori diferite pentru bandwidth, în funcție de elementele pe care le conecteză, apoi aceste magistrale să fie conectate între ele prin adaptoare de magistrale (**bridge**). Deși o astfel de structurare apare, fizic, cu mai multe magistrale, totuși logic aceste magistrale formează doar o singură magistrală.

- Adaptorul de magistrale (bridge) jonctionează două magistrale realizând:
 - adaptarea și bufferarea nevelurilor de semnale;
 - adaptarea diferențelor de viteză;
 - compatibilizează protocolele.
- Fizic, magistralele pot fi încadrate în unul din următoarele tipuri

1. **Magistrală procesor-memorie**, caracterizată prin:

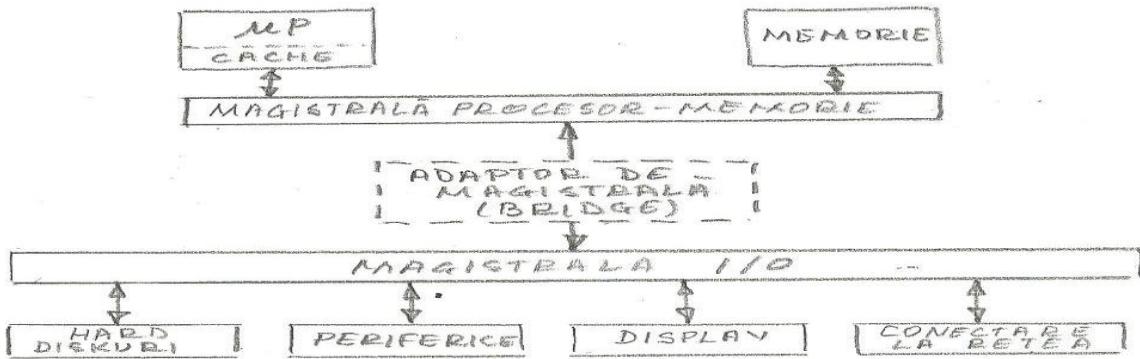
- uzual, este o magistrală proprietară realizată pentru o anumită platformă (cu un anumit microprocesor);
- bandwidth foarte ridicat, latență redusă, lungime foarte mică, realizată pentru un anumit μ P;
- conectare directă la μ P, optimizată pentru a maximiza transferul între procesor și memorie, funcționare sincronă;

2. **Magistrală I/O**:

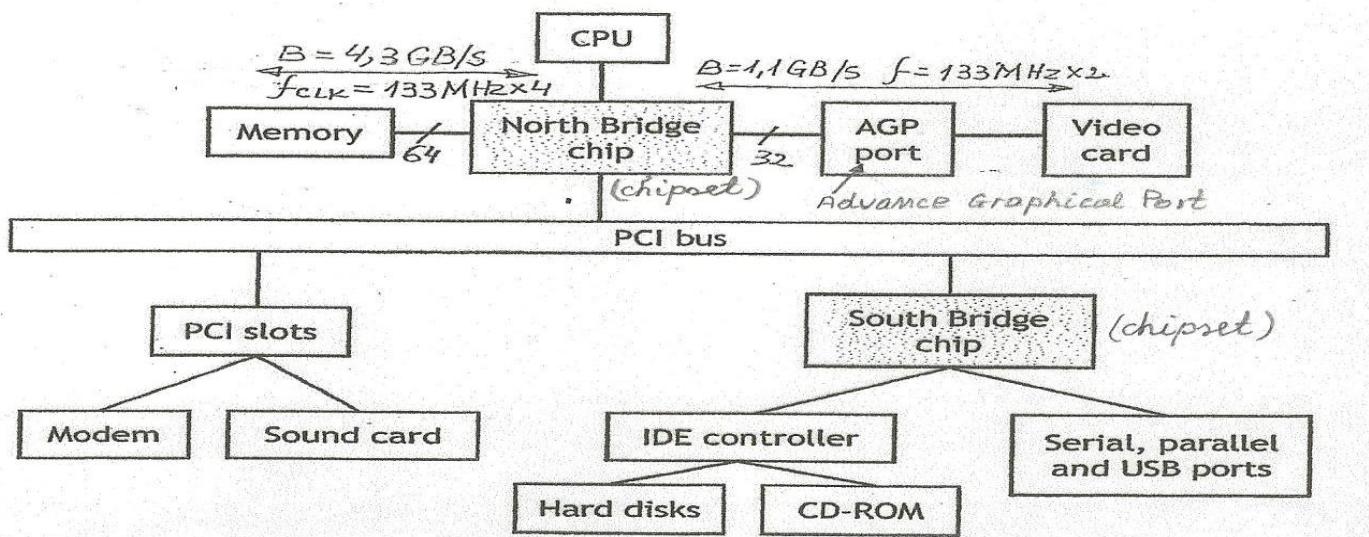
- uzual, este o magistrală standardizată realizată pentru a uni o varietate de dispozitive foarte diferite ca viteză;
- bandwidth scăzut, latență ridicată, lungime foarte mare (metrii);
- conectare prin adaptor la o magistrală fund-de-sertar sau la o magistrală procesor-memorie.

3. **Magistrala fund-de-sertar** (backplane bus):

- uzual, o magistrală standardizată, de lungime maximum 50 cm și plasată pe fundul sertarului în care este închis sistemul;
- este proiectată pentru a coexista pe o singură magistrală μ P, memoria și dispozitivele I/O; performanțele sale de comunicație balansează între cele ale magistralei procesor-memorie și cele ale magistralei I/O;
- conectarea în magistrală a componentelor I/O sub formă de carduri.
- Un exemplu de structurare a magistralelor într-un sistem este prezentat în figura următoare



- Fizic sunt două reagistral, una de viteză ridicată (procesor-memorie) cealaltă de viteză redusă (I/O)
- Logica sistemului are o singură reagistrală pe care componentele sale.
- Structurarea de principiu, pe bază de magistrale, a unui PC



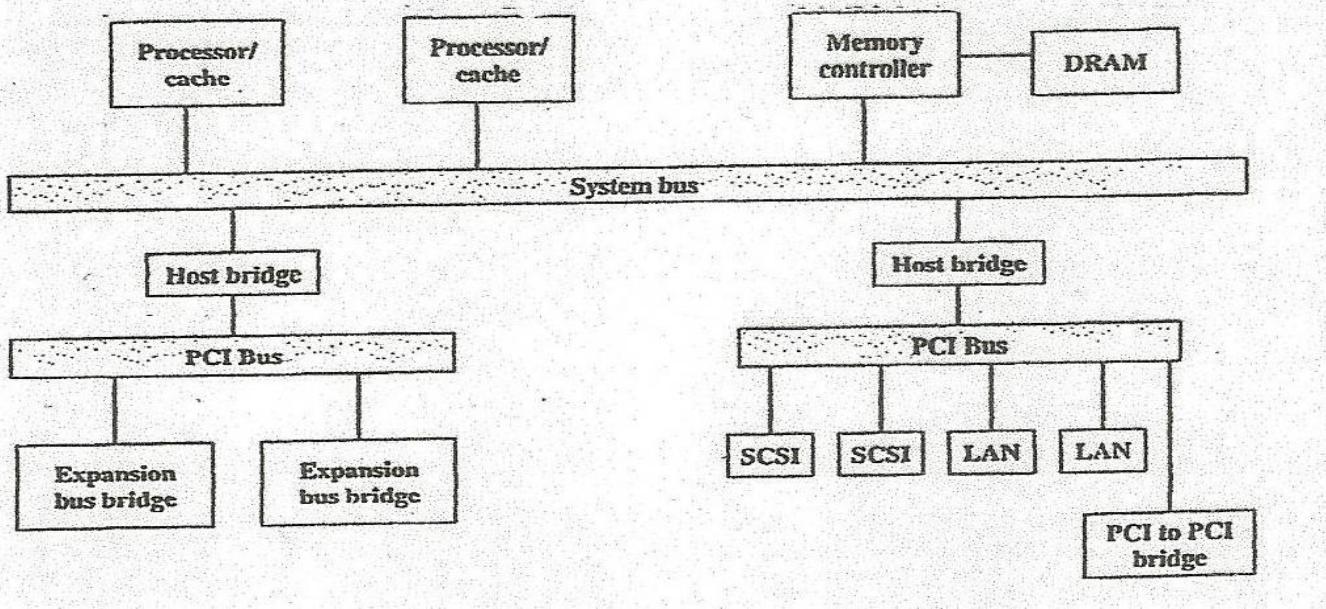
Se observă **North Bridge chipset** care conectează componente de viteză ridicată (μ P, memorie, componență video) la magistrala **PCI** (Peripheral Component Interconnect) care este o magistrală standardizată cu următoarele variante:

- PCI → $B = 32 \text{ biți} \times 33 \text{ MHz} = 133 \text{ MB/s}$
- PCI → $B = 64 \text{ biți} \times 66 \text{ MHz} = 528 \text{ MB/s}$
- PCI → $B = 64 \text{ biți} \times 133 \text{ MHz} = 1 \text{ GB/s}$

South Bridge chipset conectează componente de viteză mai redusă ale sistemului.

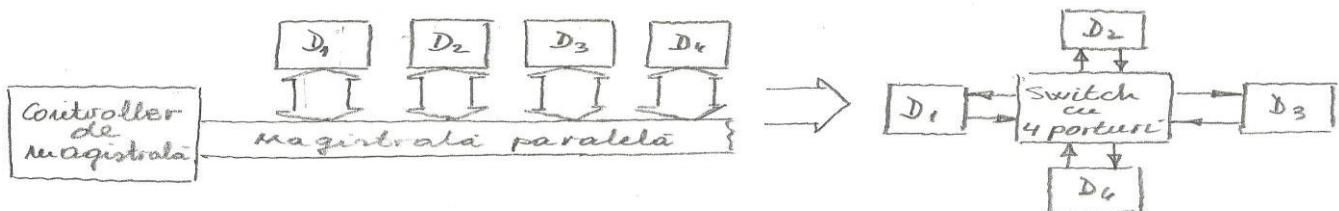
Referirea de chipset a rezultat din evoluția istorică a acestor circuite. Inițial, placa/plăcile de bază ale PC pe lângă μ P și memorie conțineau o mulțime de circuite (chip-uri) pentru realizarea sistemului. Cu progresele tehnologiei de integrare tot mai multe din aceste chip-uri au fost integrate în chip-uri de complexitate mai ridicată până când, în prezent, toate funcțiile necesare în sistem, în afară de cele proprii μ P și memoriei, au fost integrate în cele două chipuri (bridge-uri) de nord și de sud.

- Structurarea de principiu, pe bază de magistrale, a unui server

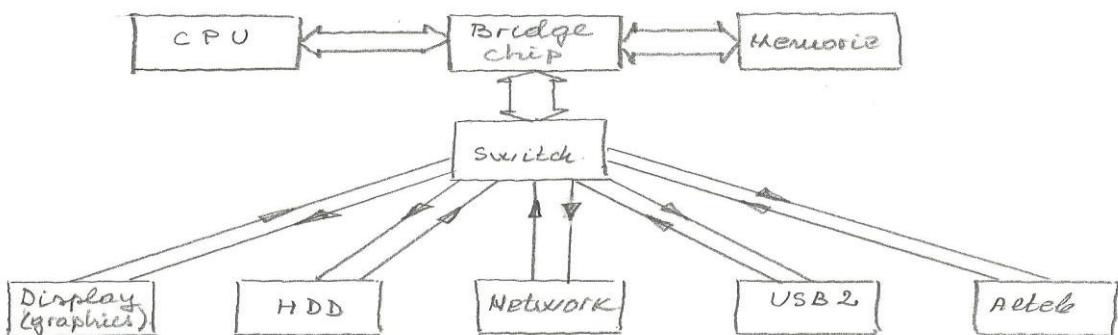


1.5.6. Prezent și tendințe în strucurarea pa bază de magistrale

- Magistrale paralele.
Avantaje: versatile, preț redus.
Desavantaje: strangularea transmisiilor (bottleneck), defazaj de ceas (clock-skew), necesită arbitraj.
- Deja în unele aplicații magistralele paralele au fost substituite de **magistrale punct-la-punct**.
Avantaje: conexiuni mai rapide (evitare clock-skew) deci performanțe mai ridicate, număr redus de pini, switch-ul realizează și arbitrarea



• Sistem PCI Express (PCIe) tipic



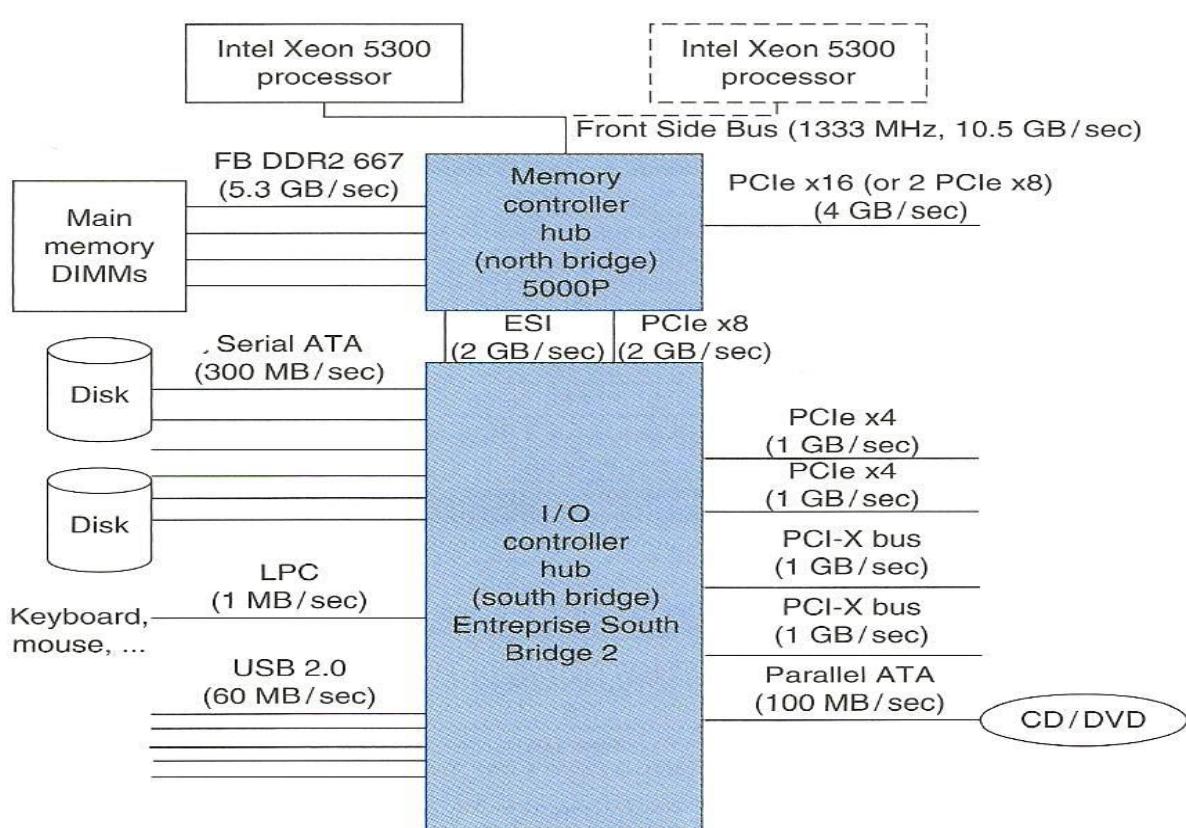
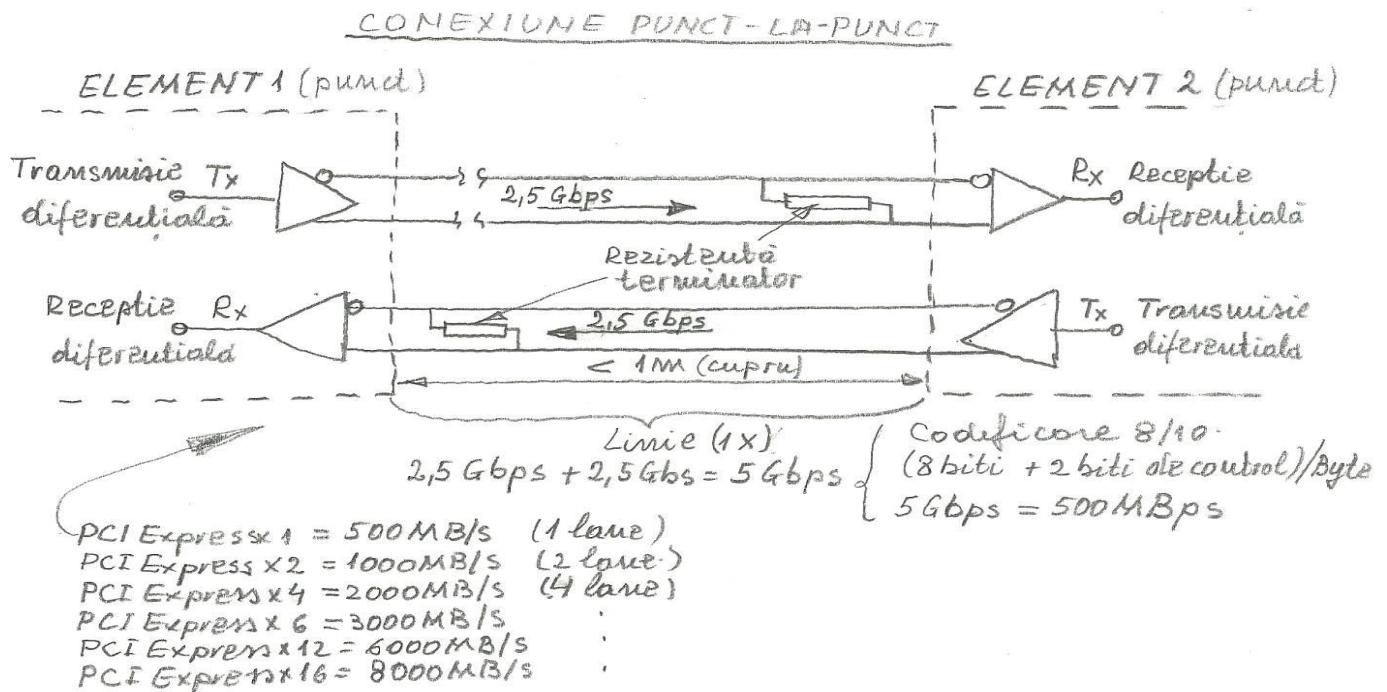


FIGURE 6.9 Organization of the I/O system on an Intel server using the Intel 5000P chip set. If you assume reads and writes are each half the traffic, you can double the bandwidth per link for PCIe.

ATA - Advance Technology Attachment, DIMM- Dual In-line Memory Module

	Intel 5000P chip set	Intel 975X chip set	AMD 580X CrossFire
Target segment	Server	Performance PC	Server/Performance PC
Front Side Bus (64 bit)	1066/1333 MHz	800/1066 MHz	—
Memory controller hub ("north bridge")			
Product name	Blackbird 5000P MCH	975X MCH	
Pins	1432	1202	
Memory type, speed	DDR2 FBDIMM 667/533	DDR2 800/667/533	
Memory buses, widths	4 x 72	1 x 72	
Number of DIMMs, DRAM/DIMM	16, 1 GB/2 GB/4 GB	4, 1 GB/2 GB	
Maximum memory capacity	64 GB	8 GB	
Memory error correction available?	Yes	No	
PCIe/External Graphics Interface	1 PCIe x16 or 2 PCIe x	1 PCIe x16 or 2 PCIe x8	
South bridge interface	PCIe x8, ESI	PCIe x8	
I/O controller hub ("south bridge")			
Product name	6321 ESB	ICH7	580X CrossFire
Package size, pins	1284	652	549
PCI-bus: width, speed	Two 64-bit, 133 MHz	32-bit, 33 MHz, 6 masters	—
PCI Express ports	Three PCIe x4		Two PCIe x16, Four PCI x1
Ethernet MAC controller, interface	—	1000/100/10 Mbit	—
USB 2.0 ports, controllers	6	8	10
ATA ports, speed	One 100	Two 100	One 133
Serial ATA ports	6	2	4
AC-97 audio controller, interface	—	Yes	Yes
I/O management	SMbus 2.0, GPIO	SMbus 2.0, GPIO	ASF 2.0, GPIO

FIGURE 6.10 Two I/O chip sets from Intel and one from AMD. Note that the north bridge functions are included on the AMD microprocessor, as they are on the more recent Intel Nehalem.

SMbus- System Management bus, GPIO- General Purpose I/O

1.6. LEGEA LUI AMDAHL

Legea lui Amdahl exprimă cantitativ cu cât se îmbunătățesc performanțele globale ale unui sistem când se îmbunătățesc doar performanțele unei componente a sistemului, componentă care este utilizată doar o fracțiune k din întregul timp de funcționarea al sistemului, ceea ce formal este redat prin relația

$$T_{\hat{imb}} = T_{nimb} \left[(1 - k) + \frac{k}{n} \right]$$

Respectiv creșterea de viteză (speed-up)

$$Speed-up = \frac{T_{nimb}}{T_{\hat{imb}}} = \frac{1}{(1 - k) + \frac{k}{n}}$$

în care:

T_{nimb} – timpul consumat de sistem pentru realizarea unei sarcini când componenta respectivă nu a fost îmbunătățită;

T_{imb} – timpul consumat de sistem pentru realizarea unei sarcini când componenta respectivă a fost îmbunătățită;

k – fracțiunea de timp, cât este utilizată componenta supusă îmbunătățirii, din timpul total necesar pentru realizarea unei sarcini;

n – de câte ori a crescut performanțele componentei îmbunătățite.

Exemplul 1.8. Un program de test (benchmark) este executat în 100s din care 80s sunt consumate de CPU (Central Processing Unit) iar restul de 20ns sunt consumate de I/O. Dacă timpul de execuție pe CPU este îmbunătățit în fiecare an cu 50%, dar nu și timpul pentru I/O, în cât timp va fi rulat acest program pe calculator după cinci ani?

Soluție.

$$T_{\text{imb}} = 100s; K = 80/100 = 0,8; n = 5 \times 1,5 = 7,5$$

$$T_{\text{imb}} = 100 \left[(1 - 0,8) + \frac{0,8}{7,5} \right] = 100(0,106 + 0,2) = 30,535s$$

și nu în 100s: $7,5 = 13,33s!$

$$\text{Speed-up} = \frac{100}{30,535} = 3,28$$

După t anii	CPU time [s]	I/O time [s]	Timpul total [s]	I/O time %
0	80	20	100	20
1	$80/1,5 = 53,33$	20	73,33	27,2
2	$53,33/1,5 = 35,55$	20	55,55	36,7
3	$35,55/1,5 = 23,70$	20	43,70	45,7
4	$23,70/1,5 = 15,80$	20	35,80	55,8
5	$15,80/1,5 = 10,53$	20	30,53	65,5

În performanță îmbunătățită a sistemului componenta îmbunătățită va avea o pondere cu atât mai mare cu cât coanta sa de timp (k) este mai mare și îmbunătățirea sa (n) crește mai mult. Pentru k = 1 creșterea performanțelor sistemului este egală cu îmbunătățirea (n) efectuată asupre componentei.

1.7. PROCESORUL – MAȘINĂ DE PROCESARE A PROGRAMELOR

1.7.1 Noțiuni: Alfabet, Sir, Limbaj

- Un **SIMBOL** este orice obiect. Exemple: #, 0, 1, a, b, begin, else, while.
- Un **ALFABET** (Σ) este orice mulțime nevidă de simboluri. Exemple:
 - Alfabetul binar $\Sigma = \{0,1\}$:
 - Alfabetul minuscul latin $\Sigma = \{a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,r,s,t,u,v,w,x,y,z\}$; 26 de simboluri (caractere).
 - Primele cinci majuscule din alfabetul latin $\Sigma = \{A,B,C,D,E\}$.
 - Simbolurile de pe tastatura unui calculator de buzunar $\Sigma = \{0,1,2,3,4,5,6,7,8,9,÷,=,-,+,\times,()\}$.
 - Câteva alfabete cu două sau trei simboluri: {T,F} True, False; {!, ?}; {a, b}; {–, ∩, ∪}; {0, 1 $^\diamond$ }.
 - Simbolurile email: (<http://www.unitbv.ro/iesc/Contact.aspx>)
- Un **ŞIR** este o secvență de simboluri pe un anumit alfabet Σ . Lungimea $|x|$ a unui sir x este egală cu numărul de simboluri din sir.

Exemple de siruri pe alfabetul {0,1}:

- 0; 1; 00; 11; 0000; 0111; 10101 respectiv cu lungimile: $|0|=1$; $|1|=1$; $|00|=2$; $|11|=2$; $|0000|=4$; $|0111|=4$; $|10101|=5$.
- Şirul vid Λ este un sir care nu conține nici un simbol $|\Lambda|=0$.

• Un **LIMBAJ** (L) peste un alfabet Σ este oricare mulțime finită sau infinită de siruri pe Σ . Altfel spus, elementele unui limbaj sunt siruri finite de simboluri din alfabetul Σ . Exemple de limaje pe alfabetul {0,1}:

- Mulțimea sirurilor x cu $|x| \leq 2$: $\{\Lambda, 0, 1, 00, 11, 10, 01\}$.
- Mulțimea sirurilor numai cu simbolul 0: $\{\Lambda, 0, 00, 000, 0000, \dots\}$.
- Mulțimea sirurilor numai cu 1 sau 0: $\{\Lambda, 0, 1, 00, 11, 000, 111, 0000, 1111, \dots\}$.
- Limbajul vid $\{\} = \emptyset$.
- Limbajul conținând sirul vid $\{\Lambda\}$; $\{\Lambda\} \neq \{\}$ deoarece $|\{\}\}| = 0$ pe când $|\{\Lambda\}| = 1$

Mulțimea tuturor limbajelor peste un alfabet Σ este notată cu Σ^*

Exemple:

1. Fie alfabetul $\Sigma = \{a\}$ pe care se definesc $L_1 = \{ \Lambda, a, aa, aaa, \dots \}$	\vdots	$L_i \subseteq \Sigma^*$	Orice limbaj este inclus sau egal cu Σ^*
	\vdots	$L_i = \{ \Lambda, a, aaaa, \dots \}$	
2. Fie alfabetul $\Sigma = \{0,1\}$ pe care se definesc $L_1 = \{ \Lambda, 0, 1, 00, 11, \dots \}$	\vdots	$L_i \subseteq \Sigma^*$	
	\vdots	$L_i = \{ \Lambda, 0, 01, 00, \dots \}$	

- Comenzile care se dau unui calculator de către operator pot fi exprimate ca şiruri de caractere din alfabetul latin. Exemple:

add – adună două numere; **sub** – scade două numere; **mult** – înmulțește două numere; **beq** – salt dacă două numere sunt egale; **bne** – salt dacă două numere nu sunt egale; **copy** – copiază un număr etc.

Un limbaj L_1 cu aceste şiruri poate fi:

$L_1 = \{ \text{add, sub, mult, beq, bne, copy} \}$ – de fapt, acesta este setul de instrucțiuni al limbajului L_1 (limbaj definit pe alfabetul latin, limbaj exprimat în această formă nu este "înțeles" de mașină, pentru a fi înțeles trebuie tradus/translatat în limbajul mașinii).

- Limbajul L_0 "înțeles" de mașină – **LIMBAJUL MAȘINĂ** – este definit pe alfabetul $\Sigma = \{0,1\}$.
- Utilizarea unui limbaj, definit pe alfabetul latin, pentru a lucra cu un calculator necesită ca fiecare instrucțiune a acestui limbaj (L_1) să fie **codificată/translatată** în limbajul mașină L_0 .

$\Sigma_1 = \{a, b, c, \dots\} \rightarrow L_1 = \{ \text{add, mult, sub, beq, copy} \}$ setul de instrucțiuni pe alfabetul latin

$\Sigma_1 = \{0, 1\} \rightarrow L_0 = \{ 0010, 0110, 1110, 0001, 1111 \}$ setul de instrucțiuni cod mașină

Translatarea anterioară a fiecărei instrucțiuni (setul de instrucțiuni pe alfabetul latin) în instrucțiuni cod mașină s-a făcut utilizând o codificare arbitrară. Dar se poate utiliza pentru codificare codul ASCII (American Standard Code for Information Interchange) în care fiecare caracter alfanumeric este codificat pe opt biți (pot fi codificate 256 de simboluri, $2^8 = 256$), de exemplu utilizând acest cod pentru instrucțiunea anterioară mult se obține cuvântul în binar (cuvânt cod mașină):

$\begin{matrix} \underline{\text{m}} & \underline{\text{u}} & \underline{\text{l}} & \underline{\text{t}} \\ \text{mult} \rightarrow 0x & 0110 & 1101 & 0111 & 0101 & 0110 & 1100 & 1111 & 0100 \end{matrix} = 6D756C74\text{hex}$ sau $6D756C74H$ (exprimări în hexazecimal)

6	D	7	5	6	C	7	4
---	---	---	---	---	---	---	---

Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char
20	(Space)	30	0	40	@	50	P	60	'	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	\$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	,	37	7	47	G	57	W	67	g	77	w
28	(38	8	48	H	58	X	68	h	78	x
29)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	
2D	-	3D	=	4D	M	5D]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Figure 2-41. The ASCII character set.

- **ASCII** cuprinde caractere latine, cifre arabe, simboluri, fiecare codificat pe opt biți (byte).
- **Universal Code (UNICODE)** cuprinde caracterele, simbolurile din aproape toate culturile, fiecare codificat pe 16 biți, deci în total 2^{16} combinații.
- Un LIMBAJ, în termeni foarte generali, este un sistem de simboluri/semne pentru comunicarea între mai multe părți.
 - Un calculator, chiar cu o interfață grafică, nu este altceva decât un manipulat de simboluri. Calculatorului i se aplică o secvență de simboluri, ca intrare, le procesează conform unui program și se obține, la ieșire, o succesiune de simboluri. Orice problemă care nu poate fi exprimată în simboluri acceptate de calculator nu poate fi procesată/rezolvată de calculator. Multitudinea problemelor rezolvate de calculator au în comun: **toate pot fi exprimate/aduse într-un limbaj înțeles de mașină**.

1.7.2 Structurarea pe niveluri a procesării pe calculator. Mașini virtuale.

- Structurarea pe niveluri a procesării pe calculator.

Nivelul cel mai de jos în procesare este cel al mașinii fizice (hardware), M_0 , pentru care există un limbaj mașină, L_0 , reprezentat simbolic prin perechea (M_0, L_0)

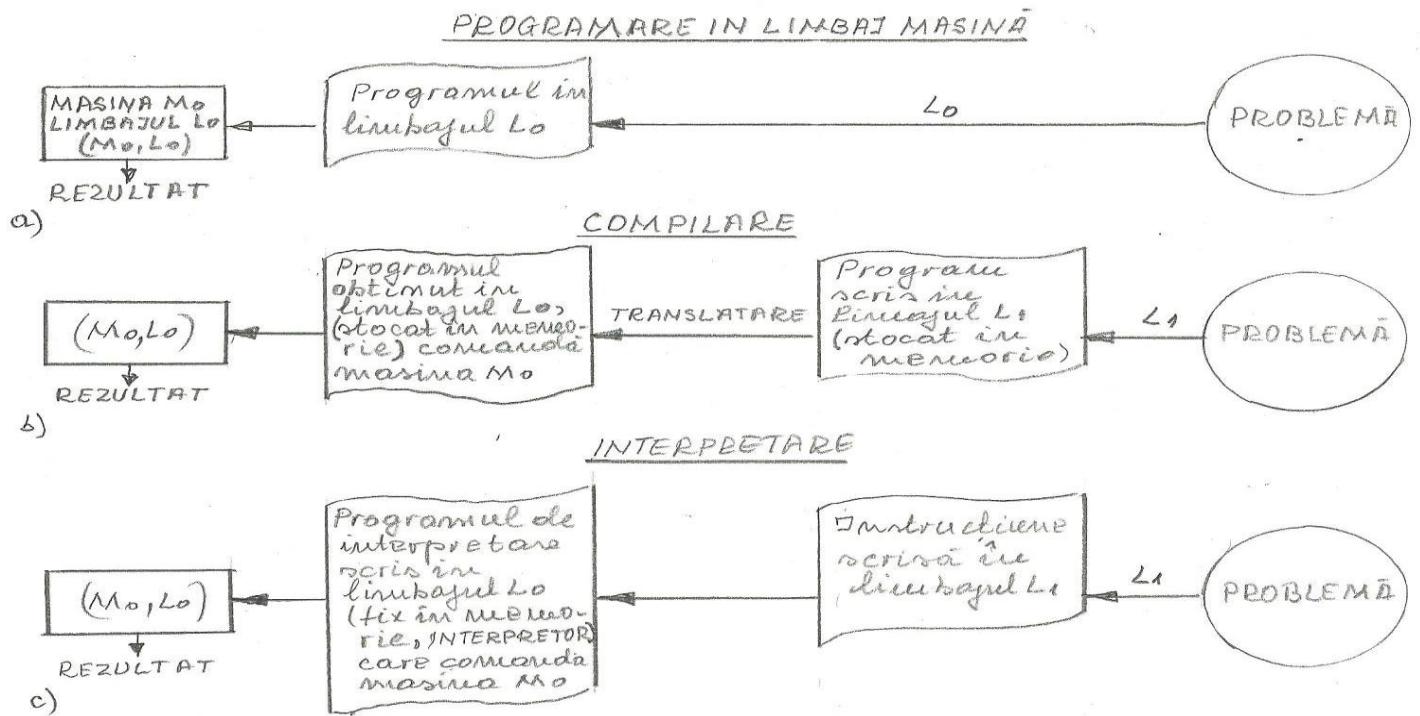


Figura a):

- problema descrisă printr-un program scris în limbajul L_0 (definit pe alfabetul $\{0,1\}$).
- programul în limbajul L_0 (cod binar, stocat în memorie) este executat de mașina M_0 (existentă fizic sub formă de circuite electronice); (M_0, L_0)

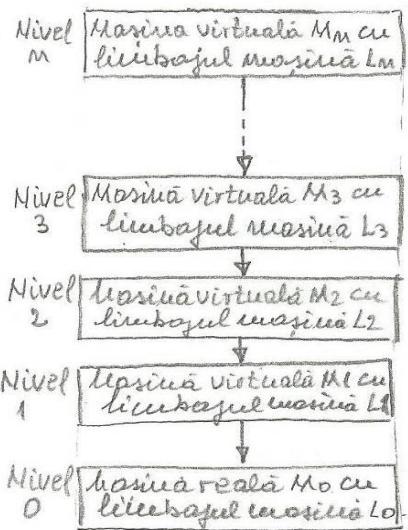
Figura b):

- problema descrisă printr-un program scris în limbajul L_1
- programul în limbajul L_1 este **compilat**/translatat într-un program, dar în limbajul L_0 și stocat în memoria mașinii M_0
- apoi programul în limbajul L_0 (stocat în memorie) este executat pe mașina M_0 .

Figura c):

- problema descrisă printr-un program scris în limbajul L_1
- fiecare instrucțiune a programului, în limbajul L_1 , este aplicată la intrarea **programului INTERPRETOR** (fix în memoria mașinii M_0) și **interpretată** ca o dată de intrare
- instrucțiunea este executată pe mașina M_0 .

- **Mașina virtuală** este doar o construcție logică și nu una fizică (nu există sub forma de circuistică-hardware). Pentru orice mașină se definește un limbaj, similar pentru orice limbaj se definește o mașină! Logic, se poate construi un limbaj oarecare L_i pentru care corespunde o mașină M_i , dar care nu are o implementare fizică – deci este o mașină virtuală (de fapt când există un limbaj există și o mașină virtuală! există perechea (L_i, M_i)).



Programul devenit în L_m este și translatat într-un program descris într-un limbaj al unei mașini de nivel mai scăzut decât m , și este interpretat de un INTERPRETOR al unei mașini de nivel mai joacă.

- Pentru construcția calculatorului (computer engineering) trebuie cunoscute toate aceste niveluri
- Pentru un programator pe un anumit nivel, cunoașterea acelui nivel, eventual și cel inferior, poate fi suficientă.

Programul în L_2 este și translatat într-un program în L_1 ori L_0 , și fiecare instrucțiune este interpretată de un INTERPRETOR pe mașina M_1 sau M_0 .

Programul în L_1 este și translatat într-un program în limbajul L_0 , și fiecare instrucțiune a programului este interpretată de un INTERPRETOR pe mașina M_0 .

Programul în limbajul L_0 definit pe alfabetul {0,1} poate fi executat direct de nivelul hardware al mașinii M_0 .

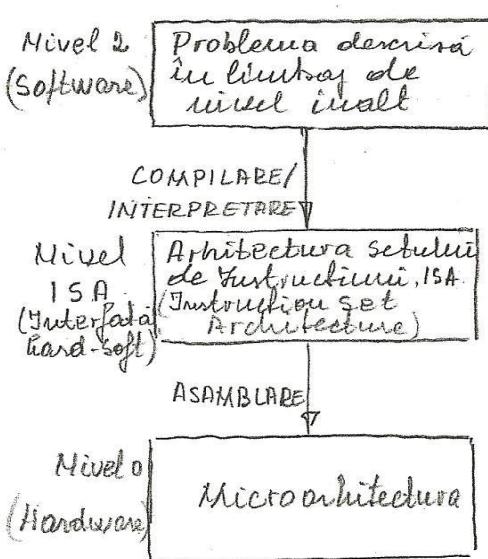
ORICE TIP DE PROBLEMA ESTE ABUSA IN FINAL INTR-UN PROGRAM IN LIMBAJUL L_0 , PE ALFABETUL {0,1}.

O problemă scrisă într-un limbaj de nivel înalt cu cât va trece prin mai multe niveluri succesive, până ajunge la mașina (M_0, L_0), va rula într-un timp din ce în ce mai lung. *Timpul cel mai scurt pentru rularea problemei se obține când programul este scris direct în limbajul L_0* , dar scrierea într-un astfel de limbaj este destul de dificilă și consumatoare de timp.

Exemplul 1.9. Mașina virtuală Java, JVM (Java Virtual Machine).

Obținerea unor programe direct executabile de pe Internet și rularea lor ca părți pentru pagina WEB ridică problema securității informației. Mai mult, aceste programe descărcate de pe internet trebuie să ruleze pe orice platformă (Pentium și Window, Sparc și Unix etc.), adică respectivele programe să fie independente de platformă. Pentru rezolvarea acestor probleme s-a inventat limbajul JAVA. Pentru limbajul Java s-a conceput mașina virtuală Java, JVM, deci perechea (limbaj Java, JVM). Un program scris în Java este compilat pentru această mașină și se obține o formă intermedieră de program – **Bytecode**, formă care este independentă de platformă, deci poate fi portat de la platformă la platformă. În prezent există aproape pe toate platformele un interpretor JVM. Deci, pentru rularea unui program scris în Java, obținut de pe Internet, sub formă de bytecode acesta se aplică la intrarea interpretorului Java, prezent pe platforma respectivă, care îl convertește într-un limbaj L_0 propriu mașinii M_0 (perechea L_0, M_0) al calculatorului pe care se lucrează.

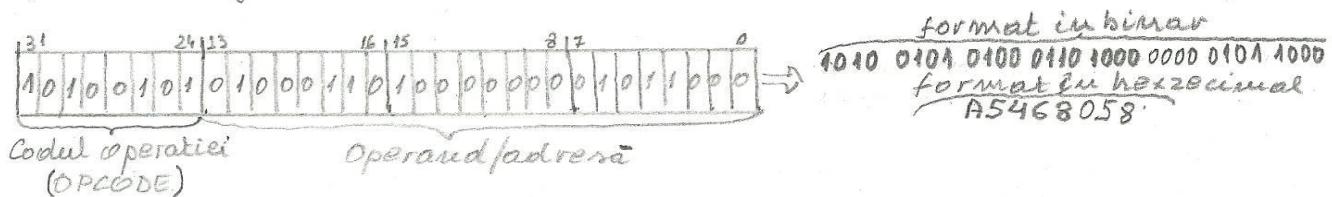
- Structurarea pe niveluri a calculatorului.



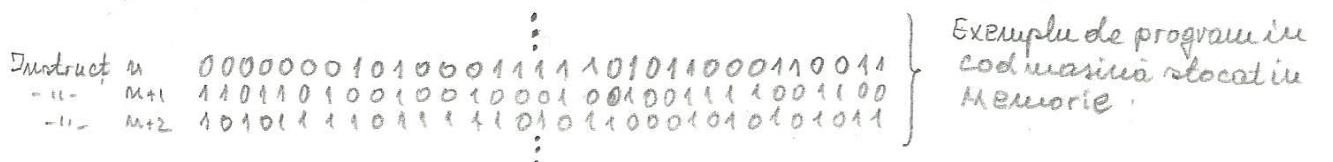
- Problema transpusă într-un program scris într-un limbaj de nivel înalt (C, C++, Java etc.)
- Mașina este privită d.p.d.v. al procesarii informației
- Componență din: CPU, Memorie, I/O
(Nu este figuraț sistemul de operare)
- Mașina este privită d.p.d.v. al capabilității de a efectua operații de transfer, aritmetică și logice
- Componență din: ALU-unitate ALU, unitatea de logica, registre, căi de transfer
- Limbajul de comunicare care comandă operații de transfer, logice și aritmice
- Mașina este privită d.p.d.v. al organelor și implementarilor hardware
- Componență din: transferoare, porti
- Limbajul mașinii este codul binar susținut de bici (cuvinte binare)

1.7.3. Organizarea și funcționarea de principiu a unui microprocesor

- Formatul general al unei instrucțiuni (în cod mașină/binar).



μ P recunoaște doar formatul binary de instrucțiune

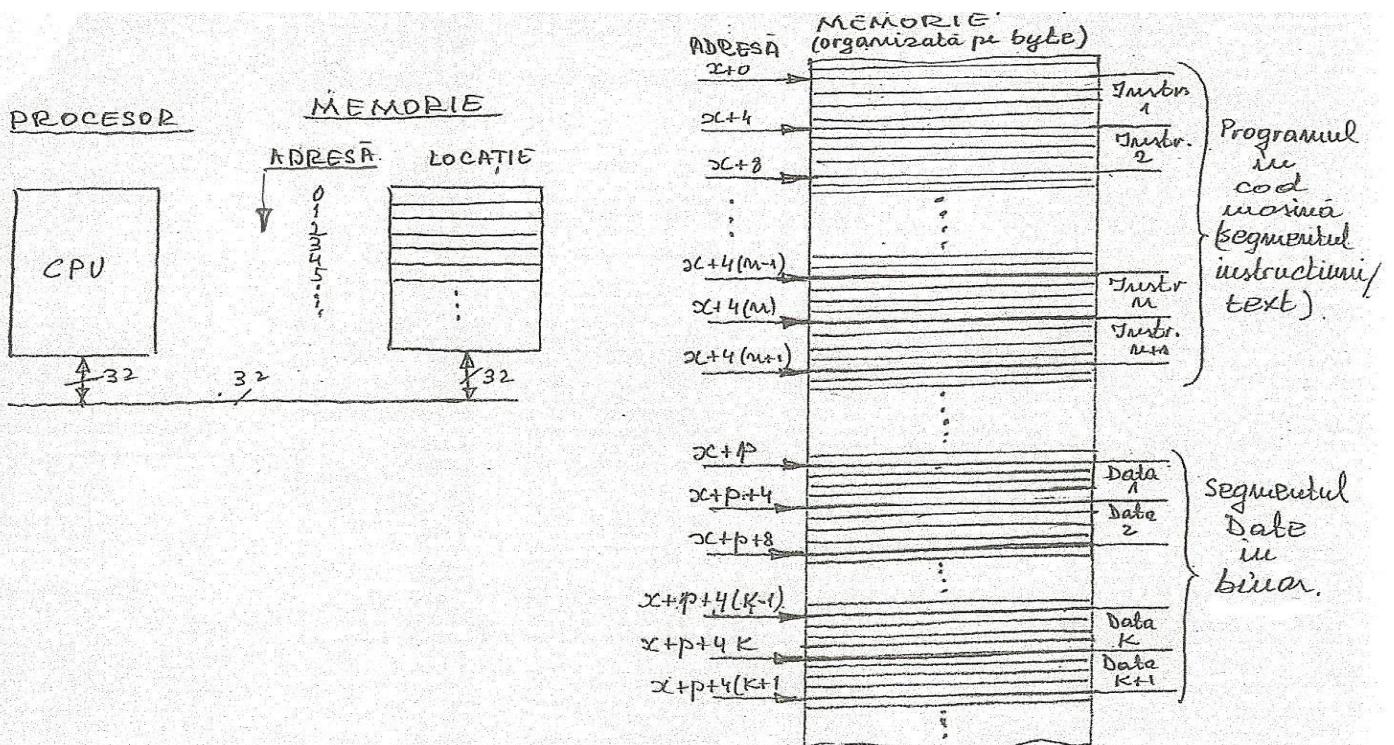


- Arhitectura de tip von Neumann.

Această arhitectură de mașină are o funcționare conform unui program stocat împreună cu datele într-o memorie comună; memoria este comună atât pentru date cât și pentru instrucțiuni. Dezavantajul acestei arhitecturi constă în faptul că procesorul fiind legat de memorie printr-o singură magistrală nu se poate accesa simultan INSTRUCȚIUNE și DATA.

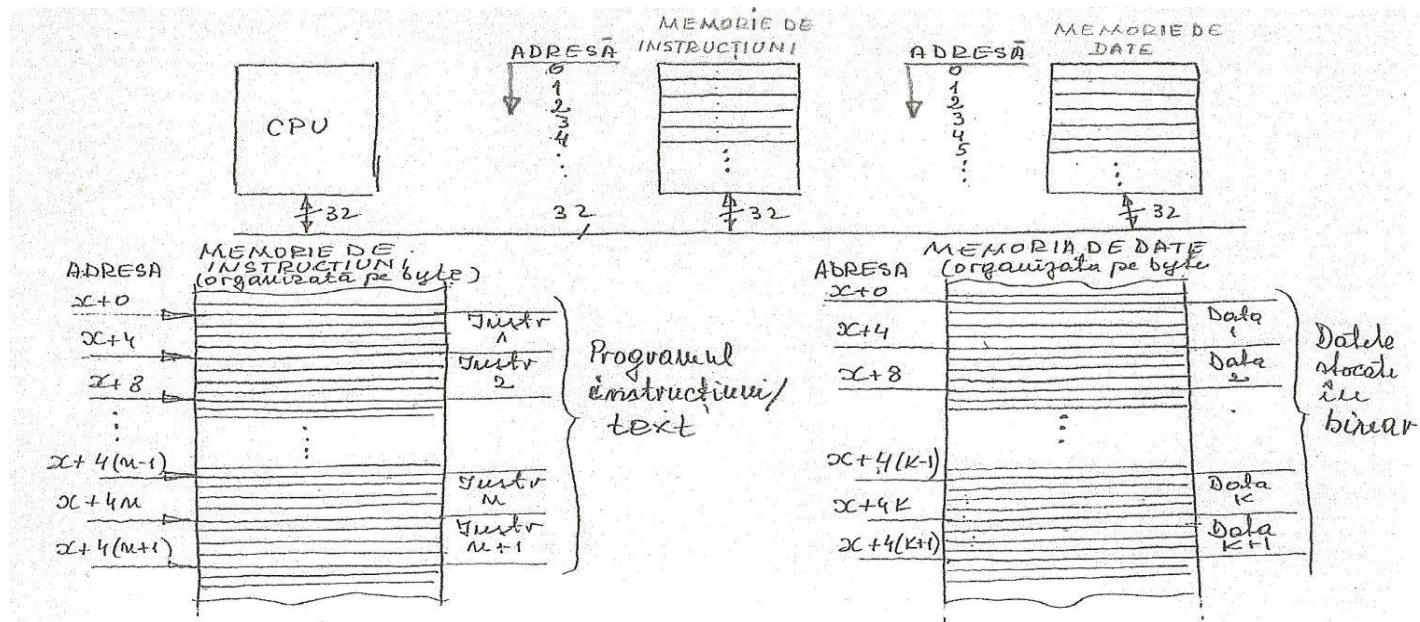
În figura de mai jos s-a considerat memoria organizată pe Byte, deci un cuvânt de 32 biți ocupă în memorie patru bytes, adică patru adrese consecutive. Relația de aliniere, de citire a unui cuvânt începând cu primul byte al cuvântului și nu bytes din două cuvinte alăturate, este:

(adresă cuvânt din memorie) modulo 4 =

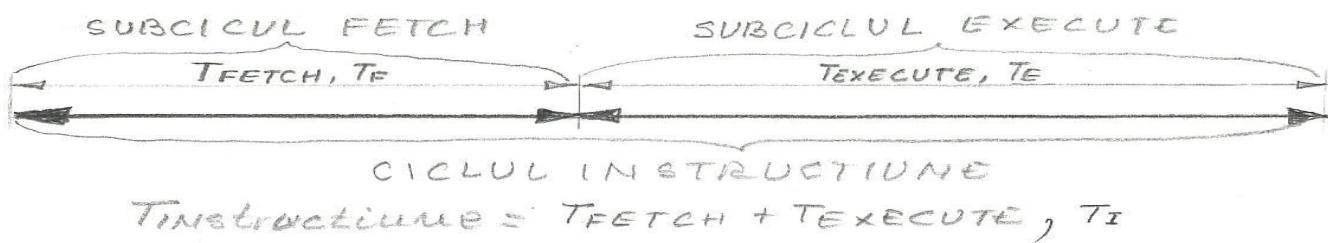


- Arhitectura de tip Harvard.

Memorie de date și memorie de instrucțiuni sunt separate, deci se pot accesa simultan datele și instrucțiunile.

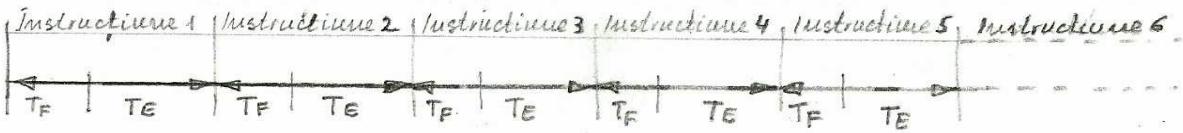


- Ciclul instrucțiune (timpul cât mașina este "ocupată" cu execuția unei instrucțiuni) cuprinde două subcicluri (frecvent, referite tot cicluri):
 - **FETCH**, cuprinde adresarea memoriei, citirea instrucțiunii, transferul pe magistrale și depunerea acesteia în un registru din μP;
 - **EXECUTE**, cuprinde realizarea de către μP a operației, codul operației (**OPCODE**), specificate în instrucțiunea adusă din memorie și depusă într-un registru din μP, apoi înscrierea rezultatului operației efectuate într-un registru al procesorului sau într-o locație de memorie.

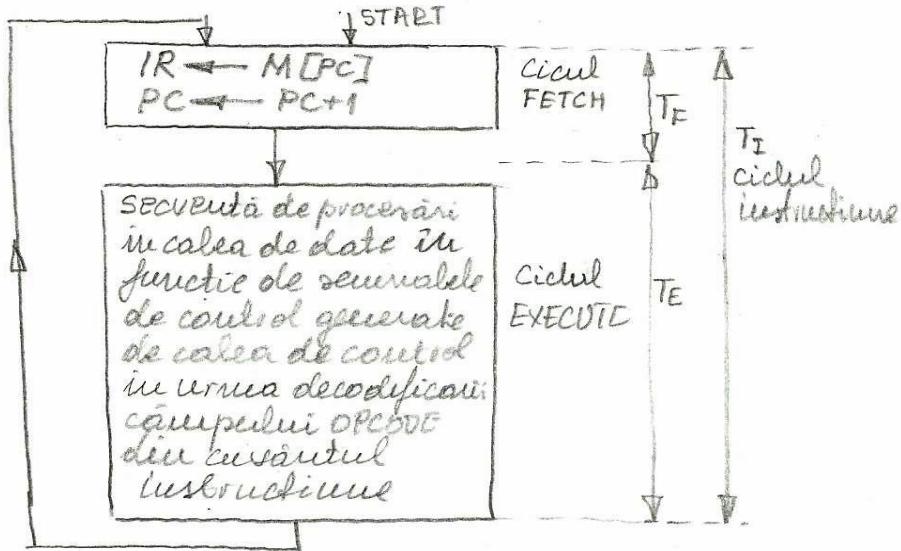


- Organograma ciclului instrucțiune

Instrucțiunile sunt procesate secvențial, sunt citite din memorie, aduse în μP și executate una după alta.



- Organizarea ciclului instrucțiunii



cicul FETCH este identic pentru toate instrucțiunile unei microprocesoruri.

cicul EXECUTE este specific pentru fiecare instrucție. Fiecare instrucție realizează o operatie care este specificata în cadrul OPCODE-ului instrucției.

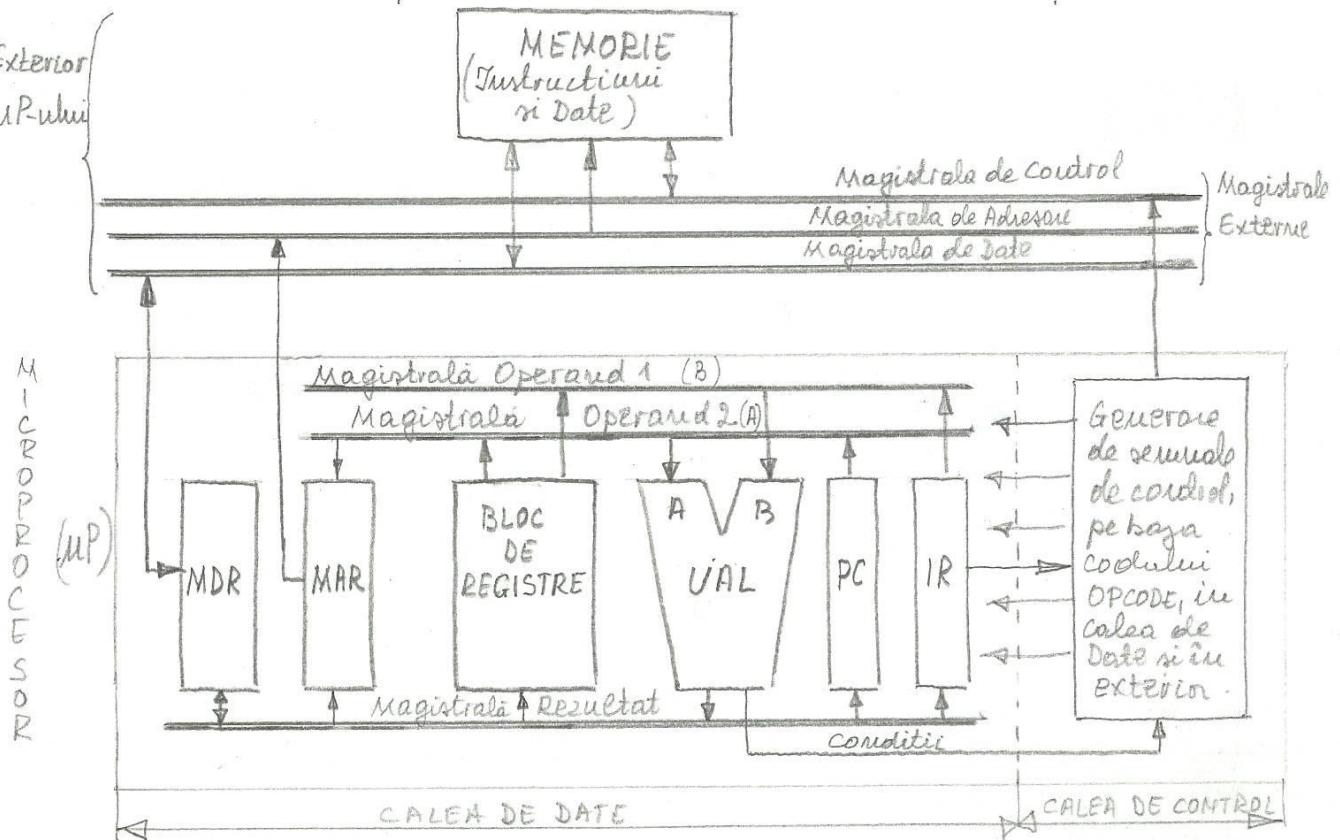
Timpul total consumat de CPU pentru execuția unui program cu lungime de N instrucțiuni, fiecare instrucție având ciclul (instrucție) este

$$T_{CPU} = N \times T_I$$

- Organizarea de principiu a unui microprocesor.

Organizarea în interiorul microprocesorului este realizată pe baza a trei magistrale: două magistrale pentru operanzi și una pentru rezultat (o operărie, în general are doi operanzi și generează un rezultat). și în exterior sistemul este organizat tot pe trei magistrale: de date, de adrese și de control; această organizare pe bază de trei magistrale este foarte uzuală. Se poate organiza procesorul și cu două sau chiar cu o singură magistrală, numai că pentru aceste organizări apar probleme de transferuri, deci administrarea accesurilor la magistrală.

Organizarea microprocesorului (atât din punct de vedere didactic cât și din punct de vedere al proiectantului) constă din două căi, **CALEA DE DATE** (cea care efectuează/execută operațiile) și **CALEA DE CONTROL** (cea care pe baza codului operației –OPCODE- instrucțiunii coordonează execuția instrucțiunii).



– **PC (Program Counter)**, numărătorul de adrese (pentru instrucțiuni). Adresa instrucțiunii (cuvântul conținut în PC) se aplică la memoria externă, $M[PC]$, de unde se citește instrucțiunea care se aduce (FETCH) pe magistrale și va fi depusă în procesor, μP , în registrul IR ($IR \leftarrow M[PC]$); programul counter se incrementeză automat ($PC \leftarrow PC + 1$) după citirea instrucțiunii din memorie, obținându-se adresa instrucțiunii următoare ($PC + 1$). Subciclul FETCH este identic pentru toate instrucțiunile.

– **IR (Instruction Register)**, Registrul de instrucțiuni. Instrucțiunea adusă în ciclul FETCH se depune în procesor în registrul IR și se păstrează în μP cât timp instrucțiunea se execută. OPCODE-ul respectivei instrucțiunii este trimis în Calea de Control, iar după decodificarea acestuia Calea de Control generează toate semnalele necesare în CALEA de DATE care vor comanda execuția instrucțiunii.

– **ALU (Arithmetic and Logic Unit)**. Unitatea Aritmetică și Logică efectuează operațiile, conform OPCODE-ului instrucțiunii, asupra operanzilor A și B pe durata subciclului EXECUTE. În urma operației execute se generează un rezultat (și dacă este cazul anumite condiții logice care se transmit căii de control); rezultatul se înscrise într-un registru intern sau în memoria externă.

– Bloc de registre, cuprinde în general un număr de registre egal cu puteri ale lui doi (8, 16, 32, 64, 128), dublu port pe ieșire, unde se păstrează datele (operanții) înainte de efectuarea operației și rezultatul după efectuarea operației. **Registrele sunt memoria internă a procesorului**.

– **MAR (Memory Address Register)** Acest regisztr nu are un rol logic în organizarea μP , ci doar o funcționalitate electrică, de fapt, este un buffer pentru semnalele din interiorul μP către exteriorul acestuia în calea de aplicare a adreselor la memoria externă.

– **MDR (Memory Data Register)** este un buffer pentru semnalele de pe traseul transferării datelor între procesor și exterior și invers.

1.7.4 Limbajul de asamblare

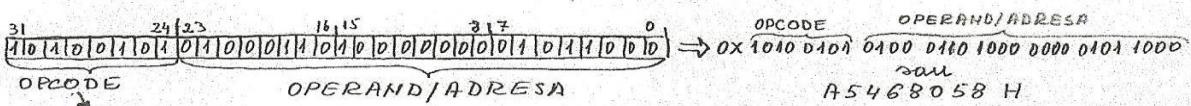
Limbajul de asamblare (assembly language) este un limbaj de nivel coborât care implementează o reprezentare simbolică a instrucțiunilor cod mașină și a datelor în scopul programării unui μP. Această reprezentare simbolică este definită de producătorul μP și se bazează pe mnemonice care specifică instrucțiuni, registre, locații de memorie și alte caracteristici ale limbajului; evident că această reprezentare (limbajul de asamblare) este propriu unui anumit procesor, ceea ce nu asigură portabilitatea unui program scris în limbaj de asamblare de la micropresosor la alt micropresosor (pentru că proesoarele nu au același limbaj mașină) spre deosebire de programele scrise într-un limbaj de nivel înalt, care sunt independente de procesor.

36

CAP. 1. Organizarea unui sistem pe bază de micropresosor

1.8.5 Limbajul de asamblare

• Instrucțiunea în cod mașină



$10100101_2 = A54 H \rightarrow$ codul operatiei Load (LD) din memorie a registrului R1

$0x468058 \rightarrow$ adresa în memorie de la care se încarcă registrul R1

• Instrucțiunea în limbaj de asamblare.

- Subcărusele din instrucțiunea în cod mașină sunt substituite cu mnemonice (silabe ușor memorabile, în general prezintări de la verbul acțiunii realizate de instrucțiune). EXEMPLU:
LD R1, (468058 H); se încarcă registrul R1 de la adresa 468058 H
- Formatarea generală de instrucțiuni

ETICHETA: CODUL OPERAȚIEI OPERAND; comentariu (nu se explică instrucțiunea INSTRUCȚIUNEA ci se specifică etapa algoritmica!!!)

EXEMPLU: $\begin{cases} a = b + c \\ d = d - e \end{cases}$ Algoritm: Se alocă variabilele la următoarele registre
 $a \rightarrow R1, b \rightarrow R2, c \rightarrow R3, d \rightarrow R4, e \rightarrow R5$

În limbaj (codul R1, R2, R3 ; suma a+b se obține în R1;
de aramblare sub R4, R4, R5 ; dif d-e se obține în R4)

- FIECARE MP ARE PROPRIUL SAV SET DE INSTRUCȚII !!!
- Tipurile de instrucțiuni ale la procesorul MIPS (Micropresosor without Interlocking în Pipeline System)

Registru R-type	31	26	25	21	20	16	15	11	10	shamt	6	5	funct	0
	OPCODE	rs	rt	rd										
add	\$ra, \$rs, \$rt													
and	\$rd, \$rs, \$rt													

slt \$rd, \$rs, \$rt; if(\$rs < \$rt) atunci \$rd $\leftarrow 1$ (set-less-than)

Immediate I-type	31	26	25	21	20	16	15	0	
	OPCODE	rs	rt	[I ₁₅]	Immediate [I ₀]				
addi	\$rt, \$rs, Imm								
lw	\$rt, Imm(\$rs)								

beq \$rs, \$rt, Imm; if(\$rs = \$rt) atunci PC $\leftarrow (PC+4) + (Imm \times 4)$, salt conditional

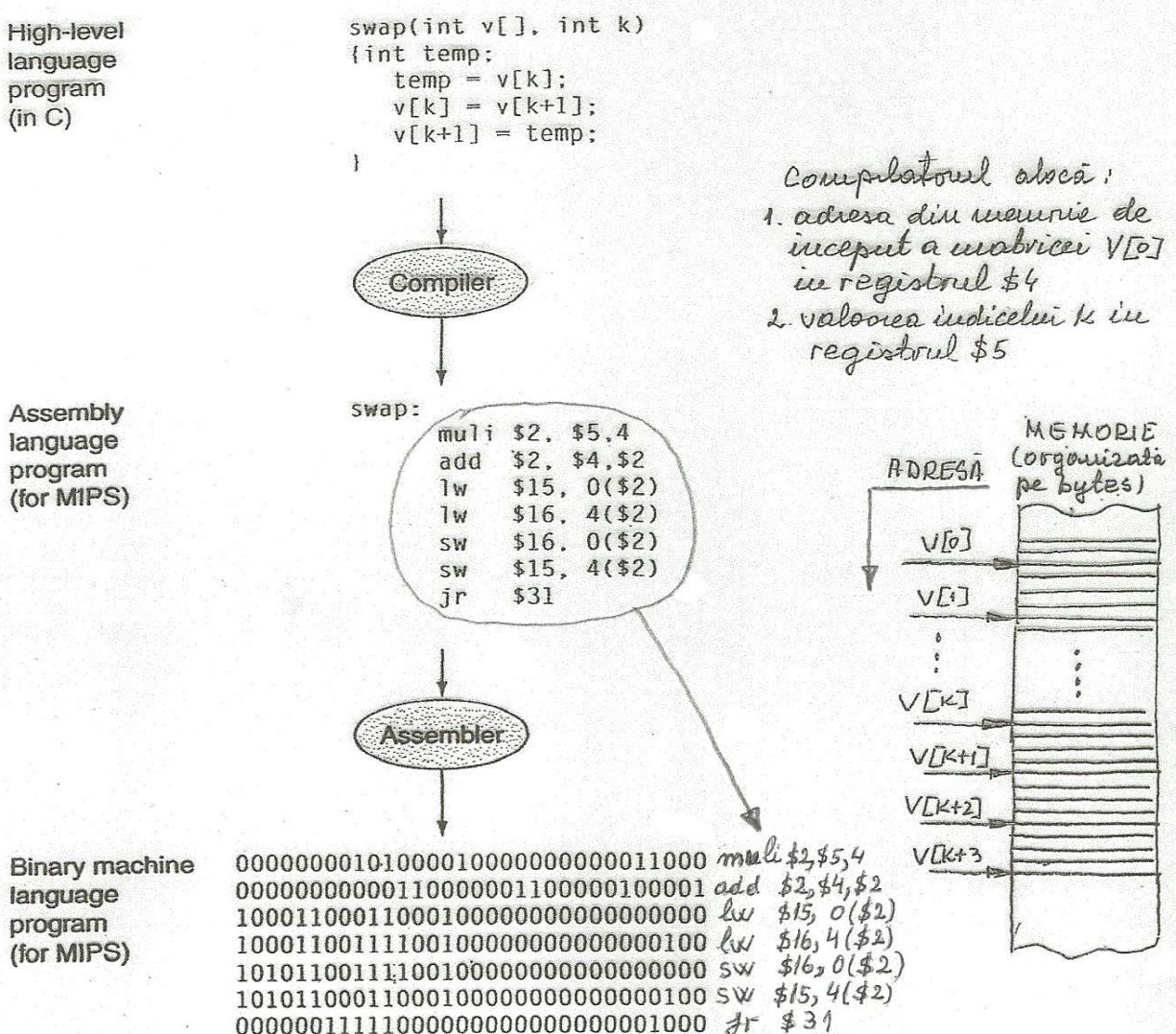
Jump J-type	31	26	25	Address (26 bici)	
	OPCODE	[I ₂₅]	Address (26 bici)		[I ₀]
J	address				

J address; $PC \leftarrow (PC_{31} \div PC_{28}) \times \dots \times (I_{25} \div I_0) \times 2^2$ salt unconditional la address

Problema de rezolvat poate fi exprimată fie într-un limbaj de nivel înalt fie în limbaj de asamblare. Dacă programul sursă respectiv este într-un limbaj de asamblare, se translatează în program limbaj mașină (program obiect) utilizând un **program** denumit **ASAMBLOR**. Asamblarea este o operație simplă deoarece se face o mapare/aplicație unu-la-unu între o instrucțiune din limbajul de asamblare și aceeași instrucțiune în cod mașină. Deoarece această aplicație este bijectivă există și **program DEZASAMBLOR** care realizează aplicația inversă, de pe mulțimea codurilor binare (instrucțiunilor cod mașină) pe mulțimea setului de instrucțiuni, adică de la o instrucțiune în binar la o instrucțiune în limbaj de asamblare.

Dacă programul sursă este scris în limbaj de nivel înalt atunci se face o compilare care produce un program în limbaj de asamblare (cum este în figura următoare), apoi programul din limbaj de asamblare este asamblat și se obține programul obiect, compilarea nu este o operație bijectivă, de aceea nu există decompliator. Fiecare instrucțiune de nivel înalt, prin compilare generează mai multe instrucțiuni de asamblare. Există compilatoare care produc direct cod obiect fără fază intermedieră de program în limbaj de asamblare.

EXEMPLU: subroutines swap, două elemente vecine ale unei matrice, $V[K]$ și $V[K+1]$ sunt schimbată între ele $V[K] \leftrightarrow V[K+1]$

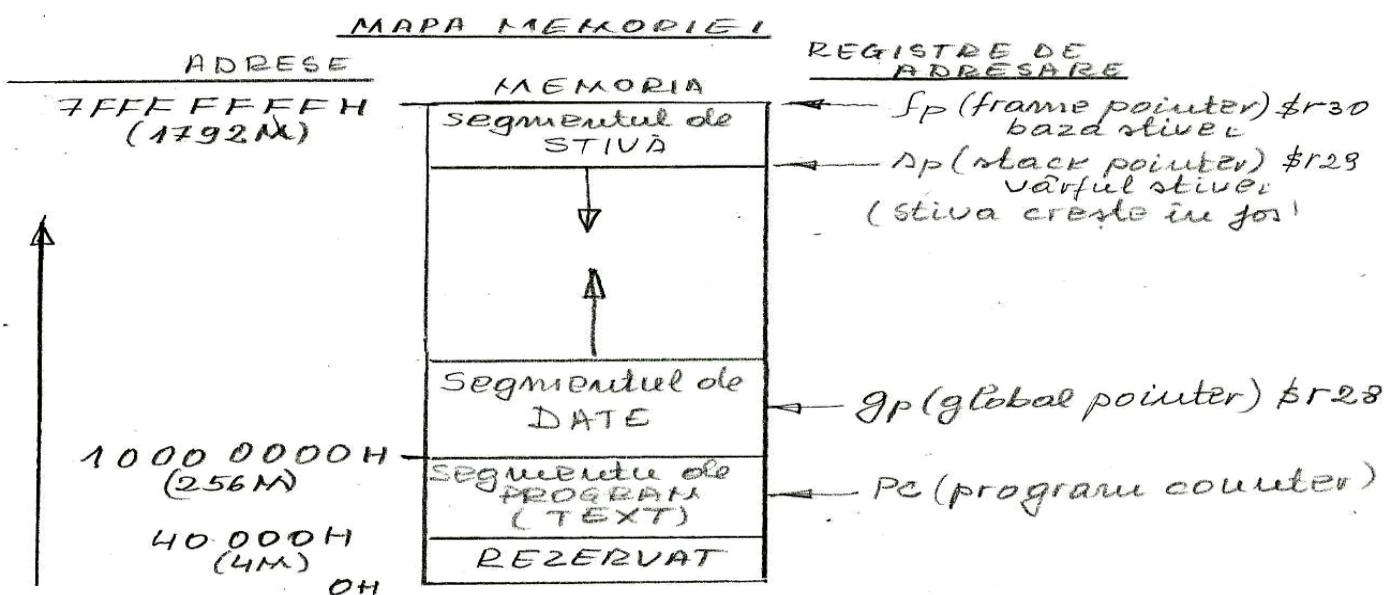


C program compiled into assembly language and then assembled into binary machine language. Although the translation from high-level language to binary machine language is shown in two steps, some compilers cut out the middleman and produce binary machine language directly.

1.7.5 Setul (partial) de instrucțiuni al procesorului MIPS (R2000)

- Prin arhitectura unui microprocessor se înțelege ceea ce “vede” un programator (utilizator) pentru scrierea programelor în limbajul de asamblare al microprocesorului sau un scriitor de compilator pentru translatarea programelor de nivel înalt în limbajul de asamblare al microprocesorului: elementele care definesc arhitectura sunt:
 - setul de instrucțiuni de asamblare, **ISA** (Instruction Set Architecture);
 - tipizarea datelor utilizate;
 - banca de register interne ale procesorului;
 - structurare logică a memoriei.

- Structurarea logică a memoriei pentru MIPS este dată în figura următoare.



- Segmentul de memorie de adrese 0- 4M este rezervat sistemului de operare.
- Segmentul de memorie de adrese 4- 256M, rezervat pentru programele în binar (obiect) care se rulează pe μ P, este referit ca segment TEXT; acest segment este adresat cu registrul **program counter**, PC.
- Segmentul de memorie de adrese 1000 000H → este alocat pentru datele stocate și generate în program, referit ca segment DATE; adresa din acest segment se construiește pe baza registrului \$r28, referit ca **global pointer**.
- STIVA (soft, realizată în memorie) a programului este plasată la partea superioară a adreselor de memorie. Baza stivei este fixată prin registrul \$r30, denumit **frame pointer**, iar vîrful stivei se adresează cu registrul \$r29, referit ca **stack pointer**. Deoarece stiva crește în jos iar datele cresc în sus, atenție ca stiva să nu intre în zona de date (sau zona dinamică de date) iar datele să nu intre în stivă (această verificare permanentă este sarcina programatorului).

Convențiile pentru asamblor în alocarea registrelor sunt:

- \$r0 (\$zero) este cablat la zero, nu poate fi înscris de programator ci doar pentru citirea constantei zero.
- \$r1 (\$at) este rezervat pentru operațiile efectuate de programul asamblor, de exemplu pentru realizarea de pseudoinstrucțiuni.
- \$r2 , \$r3 (\$v0,\$v1) sunt rezervate pentru stocarea valorilor rezultatelor unei subrute și apoi transmiterea lor programului principal (apelant).

- \$r4,\$r5,\$r6,\$r7 (\$a0,\$a1,\$a2,\$a3) sunt utilizate pentru transmiterea parametrilor/argumentelor din programul apelant (Caller) la programul apelat (Callee), de exemplu subrutină. Dacă se transmit mai mult de patru parametri atunci pe lângă cele patru registre în plus se utilizează și transmiterea prin stivă.
- \$r8,\$r9,\$r10,\$r11,\$r12,\$r13,\$r14,\$r15,\$r24,\$r25 (\$t0,\$t1,\$t2,\$t3,\$t4,\$t5,\$t6,\$t7,\$t8,\$t9), zece registre temporare. Aceste registre pot fi utilizate de programul apelat fără nici o restricție, pentru că conținutul lor a fost salvat de programul apelant (Caller saved registers) înainte ca procesorul să ruleze programul apelat.
- \$r16,\$r17,\$r18,\$r19,\$r20,\$r21,\$r22,\$r23 (\$s0,\$s1,\$s2,\$s3,\$s4,\$s5,\$s6,\$s7) opt registre temporar salvate. În aceste registre programul apelant are valori care sunt necesare și după reîntoarcerea din programul apelat, deci trebuie protejate de către apelant. Dacă programul apelat necesită și utilizarea acestor registre, trebuie întâi să le salveze (Callee saved registers), iar după utilizare să refacă conținutul lor încât la reîntoarcere în programul apelant acesta să le poată utiliza cu valoarile de dinainte de apelare nemodificate.
- \$r26 (\$k0), registru rezervat pentru utilizare numai de către nucleul Sistemului de Operare.
- \$r27 (\$k1), registru rezervat pentru utilizare numai de către nucleul Sistemului de Operare.
- \$r28 (\$gp), pointer global la segmentul DATA; se pot adresa date construind adresa prin adunare la \$gp (ca registru de bază) un deplasament.
- \$r29 (\$sp), stack pointer, indică vârful stivei alocate programului care rulează (stiva este construită în soft, deci trebuie permanent verificat ca stiva care crește în jos să nu intre în segmentul dinamic DATA)
- \$r30 (\$fp), frame pointer, fixeză baza segmentului de adrese alocate pentru stivă.
- \$r31 (\$ra), stochează adresa de reîntoarcere dintr-o subrutină. Exemplu: la apelarea unei subrute prin instrucțiunea jal sub, plasată la adresa A se salvează în \$r31 adresa următoare (de reîntoarcere, A+4); se efectuează microoperațiile PC ← sub; \$ra ← (A+4). După terminarea subrutei prin instrucțiunea jr \$ra se realizează reîntoarcere la adresa din programul apelant, A+4 (PC ← \$ra).

Convențiile fixate sunt necesare pentru a realiza programe în limbaj de asamblare portabile, când sunt scrise de diferiți programatori sau pentru a scrie părți de programe compatibile în cadrul unei echipe de programatori

Register name	Number	Usage
\$zero	0	constant 0
\$at	1	reserved for assembler
\$v0	2	expression evaluation and results of a function
\$v1	3	expression evaluation and results of a function
\$a0	4	argument 1
\$a1	5	argument 2
\$a2	6	argument 3
\$a3	7	argument 4
\$t0	8	temporary (not preserved across call)
\$t1	9	temporary (not preserved across call)
\$t2	10	temporary (not preserved across call)
\$t3	11	temporary (not preserved across call)
\$t4	12	temporary (not preserved across call)
\$t5	13	temporary (not preserved across call)
\$t6	14	temporary (not preserved across call)
\$t7	15	temporary (not preserved across call)
\$s0	16	saved temporary (preserved across call)
\$s1	17	saved temporary (preserved across call)
\$s2	18	saved temporary (preserved across call)
\$s3	19	saved temporary (preserved across call)
\$s4	20	saved temporary (preserved across call)
\$s5	21	saved temporary (preserved across call)
\$s6	22	saved temporary (preserved across call)
\$s7	23	saved temporary (preserved across call)
\$t8	24	temporary (not preserved across call)
\$t9	25	temporary (not preserved across call)
\$k0	26	reserved for OS kernel
\$k1	27	reserved for OS kernel
\$gp	28	pointer to global area
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address (used by function call)

FIGURE B.6.1 MIPS registers and usage convention.

Setul (redus) de instrucțiuni ale microprocesorului MIPS (R2000)

MIPS operands

Name	Example	Comments
32 registers	\$s0-\$s7, \$t0-\$t9, \$gp, \$fp, \$zero, \$sp, \$ra, \$at, Hi, Lo	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register \$zero always equals 0. Register \$at is reserved for the assembler to handle large constants. Hi and Lo contain the results of multiply and divide.
2^{30} memory words	Memory[0], Mémory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses; sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; overflow detected
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; overflow detected
	add immediate	addi \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constant; overflow detected
	add unsigned	addu \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; overflow undetected
	subtract unsigned	subu \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; overflow undetected
	add immediate unsigned	addiu \$s1,\$s2,100	$\$s1 = \$s2 + \$s3$	+ constant; overflow undetected
	move from coprocessor register	mfc0 \$s1,\$epc	$\$s1 = \epc	Used to copy Exception PC plus other special registers
	multiply	mult \$s2,\$s3	Hi, Lo = $\$s2 \times \$s3$	64-bit signed product in Hi, Lo
	multiply unsigned	multu \$s2,\$s3	Hi, Lo = $\$s2 \times \$s3$	64-bit unsigned product in Hi, Lo
	divide	div \$s2,\$s3	Lo = $\$s2 / \$s3$, Hi = $\$s2 \bmod \$s3$	Lo = quotient, Hi = remainder
Logical	divide unsigned	divu \$s2,\$s3	Lo = $\$s2 / \$s3$, Hi = $\$s2 \bmod \$s3$	Unsigned quotient and remainder
	move from Hi	mfhi \$s1	$\$s1 = \text{Hi}$	Used to get copy of Hi
	move from Lo	mflo \$s1	$\$s1 = \text{Lo}$	Used to get copy of Lo
	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Three reg. operands; logical AND
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \$s3$	Three reg. operands; logical OR
	and immediate	andi \$s1,\$s2,100	$\$s1 = \$s2 \& 100$	Logical AND reg, constant
Data transfer	or immediate	ori \$s1,\$s2,100	$\$s1 = \$s2 100$	Logical OR reg, constant
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 << 10$	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 >> 10$	Shift right by constant
	load word	lw \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Word from memory to register
	store word	sw \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Word from register to memory
Conditional branch	load byte unsigned	lbu \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Byte from memory to register
	store byte	sb \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Byte from register to memory
	load upper immediate	lui \$s1,100	$\$s1 = 100 * 2^{16}$	Loads constant in upper 16 bits
	branch on equal	beq \$s1,\$s2,25	if ($\$s1 == \$s2$) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if ($\$s1 != \$s2$) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than; two's complement
Unconditional jump	set less than immediate	slti \$s1,\$s2,100	if ($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$	Compare < constant; two's complement
	set less than unsigned	sltu \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than; natural numbers
	set less than immediate unsigned	sltiu \$s1,\$s2,100	if ($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$	Compare < constant; natural numbers
	jump	j 2500	go to 10000	Jump to target address
jump	jump register	jr \$ra	go to $\$ra$	For switch, procedure return
	jump and link	jal 2500	$\$ra = \text{PC} + 4$; go to 10000	For procedure call

Main MIPS assembly language instruction set. The floating-point instructions are shown in Figure 4.47 on page 291. Appendix A gives the full MIPS assembly language instruction set.

Formatul unor instrucțiuni din setul microprocesorului MIPS (R2000)

Name	Format	Example							Comments
		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits		
add	R	0	2	3	1	0	32	add	\$1,\$2,\$3
sub	R	0	2	3	1	0	34	sub	\$1,\$2,\$3
addi	I	8	2	1		100		addi	\$1,\$2,100
addu	R	0	2	3	1	0	33	addu	\$1,\$2,\$3
subu	R	0	2	3	1	0	35	subu	\$1,\$2,\$3
addiu	I	9	2	1		100		addiu	\$1,\$2,100
mfc0	R	16	0	1	14	0	0	mfc0	\$1,\$epc
mult	R	0	2	3	0	0	24	mult	\$2,\$3
multu	R	0	2	3	0	0	25	multu	\$2,\$3
div	R	0	2	3	0	0	26	div	\$2,\$3
divu	R	0	2	3	0	0	27	divu	\$2,\$3
mfhi	R	0	0	0	1	0	16	mfhi	\$1
mflo	R	0	0	0	1	0	18	mflo	\$1
and	R	0	2	3	1	0	36	and	\$1,\$2,\$3
or	R	0	2	3	1	0	37	or	\$1,\$2,\$3
andi	I	12	2	1		100		andi	\$1,\$2,100
ori	I	13	2	1		100		ori	\$1,\$2,100
sll	R	0	0	2	1	10	0	sll	\$1,\$2,10
srl	R	0	0	2	1	10	2	srl	\$1,\$2,10
lw	I	35	2	1		100		lw	\$1,100(\$2)
sw	I	43	2	1		100		sw	\$1,100(\$2)
lui	I	15	0	1		100		lui	\$1,100
beq	I	4	1	2		25		beq	\$1,\$2,100
bne	I	5	1	2		25		bne	\$1,\$2,100
slt	R	0	2	3	1	0	42	slt	\$1,\$2,\$3
slti	I	10	2	1		100		slti	\$1,\$2,100
sltu	R	0	2	3	1	0	43	sltu	\$1,\$2,\$3
sltiu	I	11	2	1		100		sltiu	\$1,\$2,100
j	J	2			2500			j	10000
jr	R	0	31	0	0	0	8	jr	\$31
jal	J	3			2500			jal	10000

MIPS instruction formats

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

Main MIPS machine language. Formats and examples are shown, with values in each field: op and funct fields form the opcode (each 6 bits), rs field gives a source register (5 bits), rt is also normally a source register (5 bits), rd is the destination register (5 bits), and shamt supplies the shift amount (5 bits). The field values are all in decimal. Floating-point machine language instructions are shown in Figure 4.47 on page 291. Appendix A gives the full MIPS machine language.

În continuare sunt prezentate opt exemple de programe scrise în limbajul de asamblare pentru microprocesorul MIPS.

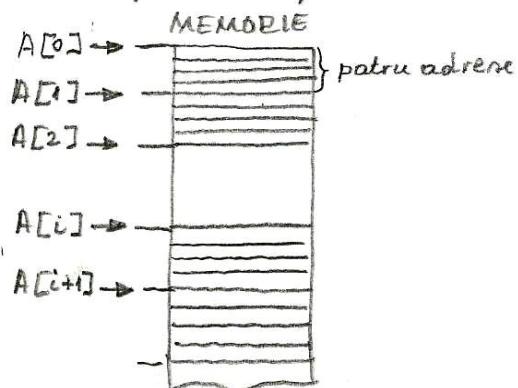
- EXEMPLUL 4.** Pentru instrucțiunea `add` din C va se genera cod masivă (la un element $V[i]$ al unei matrice V se adună scalarul h)

$$V[i] = V[i] + h$$

Se consideră că adresa $A[0]$ a primului element al matricei V . Se află în registrul $\$A_2$, valoarea indexului i este ieserită în registrul $\$A_1$, iar scalarul h se află în registrul $\$A_3$. Memoria este organizată pe byte deci un cuvânt de 32 biti ocupa patru byte de memorie, reglând că la creșterea indexului i cu o unitate adresa cadranelor successive din memorie se incrementează cu 4 (dintr-un patru în patru).

compilatorul aloca:

$A[0]$	i	h
\downarrow	\downarrow	\downarrow
$\$A_2$	$\$A_1$	$\$A_3$
($\$18$)	($\$17$)	($\$19$)



<code>add \$t_0, \$t_1, \$t_2</code>	; Reg \$t_1 conține $2 \times i$
<code>add \$t_0, \$t_0, \$t_0</code>	; $\$t_0 \leftarrow 2(2 \times i) = 4i$, indexul multiplicat cu patru
<code>add \$t_0, \$A_2, \$t_0</code>	; $\$t_0 \leftarrow A[0] + 4i$, adresa $A[i]$ în memorie a elementului $V[i]$
<code>lw \$t_1, 0(\$t_0)</code>	; $\$t_1 \leftarrow V[i]$, elementul de memorie $M[A[0]+4i]$ se descrie în $\$t_1$
<code>add \$t_1, \$A_3, \$t_1</code>	; $\$t_1 \leftarrow V[i] + h$
<code>sw \$t_1, 0(\$t_0)</code>	; la adresa $A[0]+4i$ se inscrie componenta vectorului $V[i]$ sumată cu scalarul h .

Utilizând conexiunea de utilizare a registrului și codul în binar al instrucțiunilor procesorului în forma arhitecturii se obține următorul cod masivă

cod masivă (cod obiect)
02314020 H
01084020 H
02484020 H
80090000 H
02694820 H
A0D90000 H

→

<code>add \$8, \$17, \$17</code>	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>17</td><td>17</td><td>8</td><td>0</td><td>32</td></tr></table>	0	17	17	8	0	32	<code>02314020 H</code>
0	17	17	8	0	32			
<code>add \$8, \$8, \$8</code>	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>18</td><td>8</td><td>8</td><td>0</td><td>32</td></tr></table>	0	18	8	8	0	32	<code>01084020 H</code>
0	18	8	8	0	32			
<code>add \$8, \$18, \$8</code>	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>18</td><td>8</td><td>8</td><td>0</td><td>32</td></tr></table>	0	18	8	8	0	32	<code>02484020 H</code>
0	18	8	8	0	32			
<code>lw \$9, 0(\$8)</code>	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>35</td><td>8</td><td>9</td><td></td><td>0</td><td></td></tr></table>	35	8	9		0		<code>80090000 H</code>
35	8	9		0				
<code>add \$9, \$19, \$9</code>	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>19</td><td>9</td><td>9</td><td>0</td><td>32</td></tr></table>	0	19	9	9	0	32	<code>02694820 H</code>
0	19	9	9	0	32			
<code>sw \$9, 0(\$8)</code>	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>43</td><td>8</td><td>9</td><td></td><td>0</td><td></td></tr></table>	43	8	9		0		<code>A0D90000 H</code>
43	8	9		0				

Traducatoarea dăe limbaj de nivel înalt (cod sură) în cod reprezentă (cod obiect) și-a realizat încă din compilatoare și a obținut programul în limbaj de asamblare, apoi prin disasembler s-a obținut programul în cod masivă (obiect).

42

CAP. 1. Organizarea unui sistem pe bază de microprocesor

- EXEMPLUL 2.** Obținerea unei constante cu lungimea mai mare de 16 biti / Iată instrucțiunile MIPS admise constante-immediate-două de 16 biti, deci o valoare în intervalul $0 \div (2^{16}-1)$.
Instrucțiunea lui (load upper immediate) să plaseze o constantă cu lungimea de 16 biti, plasată în poziția 15-0 din cadrul instrucțiunii, în poziția 31-16 dintr-un register al procesorului, deci o multiplicare cu 2^{16} .

Lui \$t0, 255 ; \$t0 este registerul \$8 conform convenției de alocare.

După execuție
conținutul reg \$t0 va fi:

Să se încarce \$t0 cu următoarea valoare:

0000 0000 0011 1101	0000 1001 0000 0000
61 ₁₀	2304 ₁₀

lui \$t0, 61 ; conținutul registerului \$t0 va fi

0000 0000 0011 1101	0000 0000 0000 0000
61 ₁₀	0

add \$t0, \$t0, 2304 ; 0000 0000 0011 1101 0000 1001 0000 0000 ; \$t0

- EXEMPLUL 3.** Sa elementele $V[i]$ ale unei matrice vector V să se sumeze la rezultatul R (este o extensie de la EXEMPLUL 1, adunându-se rezultatul R cu la un singur element al matricii și totalele elementelor). Numărul elementelor matricii este dim, iar adresa primului element $A[0]$

Programul în C

Loop: $V[i] = V[i] + R$

$i = i + j$

if ($i \neq l$) go to Loop

corespunzătorul adresei registrile

i	$A[0]$	h	$j (=1)$	$dim (=100) = \text{limite}$
↓	↓	↓	↓	↓
\$t1	\$t2	\$t3	\$t4	\$t5

add \$t1, \$zero, \$zero ; se initializează valoarea contorului ($i=0$)
 Loop: add \$t0, \$t1, \$t1 ; se adună valoarea componentă
 add \$t0, \$t0, \$t0 ; se obține val. contorului multiplicată cu 4
 add \$t0, \$t2, \$t0 ; $\$t0 \leftarrow A[0] + i \times 4$, se calculează adresa elementului $V[i]$
 lwr \$t1, 0(\$t0) ; $\$t1 \leftarrow M[A[0] + 4 \times i]$, se citește elementul $V[i]$
 add \$t1, \$t3, \$t1 ; $\$t1 \leftarrow V[i] + R$
 swr \$t1, 0(\$t0) ; $M[A[0] + 4 \times i] \leftarrow V[i] + R$ în memorie se șterge
 valoarea componentă rezervată cu R
 add \$t1, \$t4, \$t4 ; se incrementează indexul $i = i + 1$
 bne \$t1, \$t5, Loop ; Dacă $i \leq dim - 1$ atunci se reia sumarea
 pentru următorul element al matricii

Acest program este o buclă do-while.

Gh. TOACSE – Arhitectura și Organizarea Microprocesoarelor

- EXEMPLUL 4. PSEUDOINSTRUCTIUNILE sunt instrucțiuni pe care numai **asemblerul le recunoaște**, acestea nu sunt codate în procesor, adică OPCODE-UL lor nu este recunoscut de procesor. Când în programul scris, în liniile de **asemblare**, se întâlnește o pseudoinstrucție, **asemblerul o substituie cu ce altă instrucție** ce există corect recunoscut de procesor (este **simbolizată**).

- Pseudoinstrucția move

move \$rdest, \$rsursă ; \$rdest \leftarrow \$rsursă
este substituia cu

add \$rdest, \$zero, \$rsursă ; \$rdest \leftarrow \$zero + \$rsursă

În ISA pentru MIPS există următoarele două instrucțiuni de salt bne \$ra, \$rb, ACOL0 ; Dacă \$ra \neq \$rb atunci salt la adresa ACOL0
beq \$ra, \$rb, ACOL0 ; Dacă \$ra = \$rb atunci salt la adresa ACOL0

Pentru următoarele patru relații de ordine $a < b$, $a \geq b$, $a > b$, $a \leq b$
nu există instrucțiuni de salt dar se poate realiza salt și
pentru aceste relații de ordine definiind următoarele
pseudoinstrucții (deci în total se poate realiza salt cu 6 relații de ordine)

- Pseudoinstrucția branch-less-than, blt ($a < b$)

blt ; \$ra, \$rb, ACOL0 ; Dacă $a < b$ atunci salt la adresa ACOL0
care este substituia de asembler ce următoarele 2 instrucții.
slt \$t0, \$ra, \$rb ; dacă $a < b$ atunci $\$t0 \leftarrow 1$, altfel ($a \geq b$) $\$t0 \leftarrow 0$
bne \$zero, \$t0, ACOL0 ; dacă $\$t0 \neq \$zero$ salt la ACOL0 (deci $a < b$)

- Pseudoinstrucția branch-greater-or-equal, bge ($a \geq b$)

bge \$ra, \$rb, ACOL0 ; Dacă $a \geq b$ atunci salt la adresa ACOL0
care este substituia de asembler ce următoarele 2 instrucții
slt \$t0, \$ra, \$rb ; dacă $a < b$ atunci $\$t0 \leftarrow 1$, altfel ($a \geq b$) $\$t0 \leftarrow 0$
beq \$zero, \$t0, ACOL0 ; dacă $\$t0 = \$zero$ salt la ACOL0 (deci $a \geq b$)

- Similar se pot defini pseudoinstrucțiuni

branch-greater-than, bgt ($a > b$)

branch-less-or-equal ble ($a \leq b$)

- { beq \$ra, \$rb, ACOL0 ; salt la adresa ACOL0 dacă $\$ra = \rb

L2: următoarea instrucție din program
Dacă saltul părăsește adresa ACOL0 este mai lung decât se
poate exprima printr-o miecă biti (imediatul din formă
instrucției de salt) atunci asemblerul substituie cu:

{ bne \$ra, \$rb, L2 ; salt la adresa L2 dacă $\$ra \neq \rb

 ; salt recondiționat la adresa ACOL0 dacă $\$ra = \rb
 ; Immediatul ACOL0 este exprimat cu 26 de biti

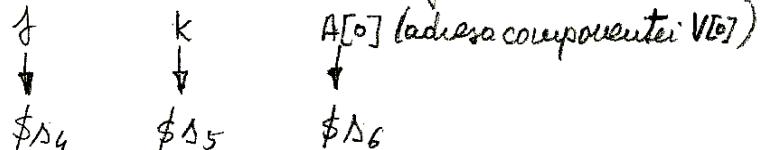
L2: următoarea instrucție din program după } ACOL0

- EXEMPLUL 5. În exemplul 3 se parcurgea succesiv toate elementele unei matrice coloană (vector) până la atingerea valori maxime a indexului (bucătă de until). În acest exemplu se parcurge un număr neșpecificat de elemente ale vectorului, abăt căt este îndeplinită o anumită condiție ($V[i] == k$), buclă do while. Forma clasică de buclă do while, pentru acest exemplu, în limbajul C se scrie:

`while (V[i]==k)`

`i = i + j`

Compilatorul aloca registrele:



După compilare se obține programul în limbaj de arhitectură

```

add    $t1, $zero, $zero ; indexul se inițializează (i=0)
Loop: add   $t0, $t1, $t1 ; $t0 ← 2 × i
      add   $t0, $t0, $t0 ; $t0 ← 4 × i
      add   $t0, $t0, $A6 ; $t0 ← A[0] + 4 × i se calculează adresa elementului V[i]
      lw    $t0, 0($t0) ; $t0 ← M[A[0] + 4 × i] se citește elementul V[i]
      bne  $t0, $A5, IESIRE ; Dacă V[i] ≠ k atunci salt la adresa IESIRE și se continuă la următorul element (salut la loop)
      add   $t1, $t1, $A4 ; se incrementează indexul i = i + j.
      J     Loop          ; se repetă bucla
  
```

IESIRE:

Asa cum să se generează următorul program în cod rusesc. Se consideră că adresa etichetei Loop este 800001₁₀.

ADRESE

79996	0 0 0 9 0 32
Loop: 80000	0 9 9 8 0 32
80004	0 8 1 8 8 0 32
80008	0 8 1 22 8 0 32
80012	35 1 8 1 8 0
80016	5 1 8 21 2
80020	0 1 9 20 9 0 32
80024	0 1 20000

IESIRE: 80028

```

add $9, $0, $0
add $8, $9, $9
add $8, $8, $8
add $8, $8, $22
lw $8, 0($8)
bne $8, $21, 8 (2 × 4 = 8)
add $9, $8, $20
J 80000 (4 × 20000 = 80000 se obține printr-o deplasare la stânga cu elorūa poziții a cursorului 20000).
  
```

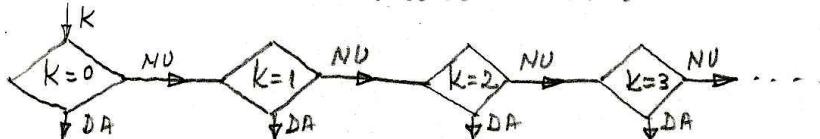
Înstrucția nr. 6-a este un salt la adresa IESIRE(80028) care se obține prin adunarea la continutul PC+4 a valorii 8. Continutul în PC este $PC+4 = 80016+4 = 80020$, deoarece după aducerea unei instrucții din memoria lui acesta (instrucția de adresa 80016) PC indică adresa următoare, adică PC+4. De asemenea înregistrul din corpul instr. de salt include de adunare se deplasează cu 2 pozitii la stânga, deci rezultă că $4(2 \times 4 + 20020 = 20028 = IESIRE)$ la sfîrșitul instr 8-a.

45

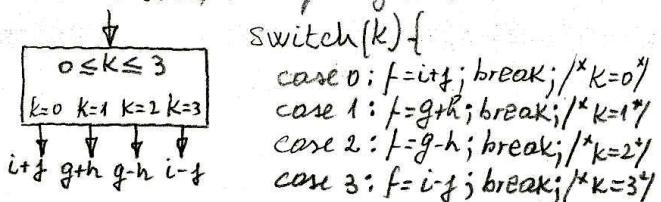
CAP 1. Organizarea unui sistem pe bază de microprocesor

- EXEMPLUL 6. Instrucțiunea SWITCH din limbajul C, uneori referită ca CASE, să fie compilată în limbajul de asamblare MIPS.

Se pune deosebire de decizia binară, unde sunt două căi de ieșire, decizia prin SWITCH nu realizează nici multe căi de ieșire în funcție de valoarea unei variabile k. Implementarea unei decizii cu mai multe căi de ieșire se poate realiza și printr-un lanț de decizii IF-THEN-ELSE în modul următor:



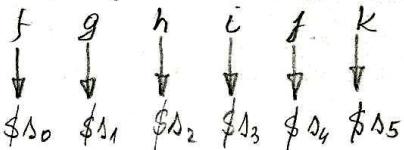
O altă variantă de implementare a instrucțiunii SWITCH, în limbaj de asamblare, este ce opțional unui TABEL CU ADRESE DE SALT, TAS. În funcție de valoarea lui k se determină o adresă ATAS[k] în TAS. Într-o locație adiacentă din TAS este iperscriea o instrucție de salt care efectuează un salt la o adresă unde se află programul ce realizează procesarea pe calea de ieșire din instrucțiunea SWITCH corespunzătoare valorii variabilei k. Versiunea în C a instrucțiunii C a instrucțiunii SWITCH cu patru căi de ieșire, corespunzătoare lui k=0, 1, 2, 3 este:



switch(k){

case 0 : $f = i + j$; break; /* $k=0$ */
 case 1 : $f = g + h$; break; /* $k=1$ */
 case 2 : $f = g - h$; break; /* $k=2$ */
 case 3 : $f = i - j$; break; /* $k=3$ */

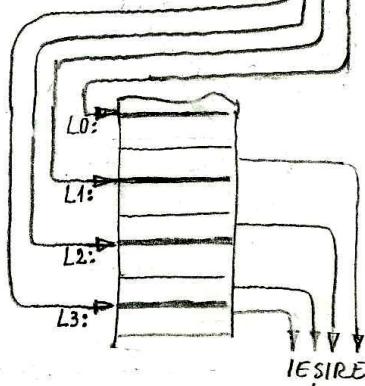
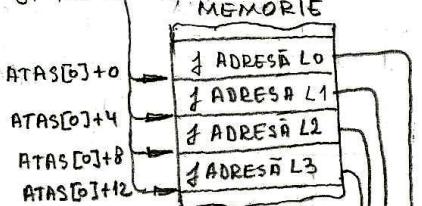
Complilatorul aloca:



Adresa de început a tabelului ATAS[0] se află în \$t₄, iar valoarea k este în \$t₂. În urma compilării se obține:

```

slt $t3, $t0, $zero ; Test dacă k<0
bne $t3, $zero, Iesire ; Dacă k<0, terminat SWITCH
slt $t3, $t5, $t2 ; Test dacă k<4
beq $t3, $zero, Iesire ; Dacă k>3, terminat SWITCH
add $t1, $t0, $t5; $t1 ← 2xk
add $t1, $t1, $t1; $t1 ← 4xk
add $t0, $t1, $t4; $t1 ← ATAS[0] + 4k
  
```

Jr \$t₀ ; TABELUL CU ADRESE DE SALT

```

jr $t0 ; salt la una din adresele L0, L1, L2, L3
L0: add $t0, $t3, $t4; f = i + j
     f Iesire ; terminat SWITCH
L1: add $t0, $t1, $t2; f = g + h
     f Iesire ; terminat SWITCH
L2: sub $t0, $t1, $t2; f = g - h
     f Iesire ; terminat SWITCH
L3: sub $t0, $t3, $t4; f = i - j
     f Iesire ; terminat SWITCH
  
```

- EXEMPLUL 7 Se să se inițieze cu valoarea zero toate elementele unei matrice coloana array[i] care are dimensiunea, dim.
Pentru accesarea elementelor matricii se va utiliza comparativ adăugând indexul căt și pointerul.

Programul în C

Utilizarea indexului, i:

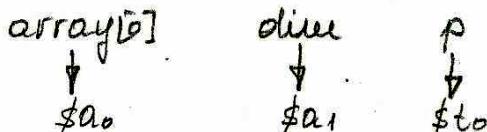
Procedura sterge1

sterge1 (int array[], int dim)

```
int i
for(i=0, i < dim; i=i+1)
    array[i]=0
```



Compilatorul aloca:



Programul în limbaj de asamblare

```
move $t0,$zero ; i=0
Loop1: sll $t1,$t0,2 ; i:$t1 = i×4
        add $t2,$a0,$t1 ; $t2 = adresa array[i]
        sw $zero,0($t2) ; array[i] ← 0
        addi $t0,$t0,1 ; i = i+1
        slt $t3,$t0,$a1 ; $t3 = 1(i < dim)
        bne $t3,$zero,Loop1; dacă(i < dim) atunci salt la Loop1
```

Deoarece în fiecare iteratie se calculează adresa urmării elementelor matricei, cu toate că acesta nu se redoublează, se poate reduce numărul de instrucțiuni din corpul buclei în felul următoare.

```
move $t0,$a0 ; p = &array[0]
sll $t1,$a1,2 ; $t1 = dim × 4
add $t2,$a0,$t1 ; $t2 = &array[dim]
sw $zero,0($t2); Memorie[p] ← 0
addi $t0,$t0,4 ; p = p+4
slt $t3,$t0,$t2 ; $t3 = (p < &array[dim])
bne $t3,$zero,Loop2; dacă(p < &array[dim]) atunci salt la Loop2
```

Procedura sterge1 trebuie să multiplice și să adună în cadrul buclei pentru ca indexul să fie incrementat și fiecare adresă să se recalculeze în fiecare iteratie. În schimb la procedura sterge2 pointerul p se încrementează direct, ceea ce duce la micșorarea numărului de instrucțiuni din corpul buclei de la sare la patru (aleși un factor mai scurt de procesor).

utilizarea pointerului p

&var - adresa variabilei var
*p - obiectul indicat de pointerul p

Procedura sterge2

sterge2 (int *array, int dim)

```
int *p
for(p=&array[0]; p < &array[dim]; p=p+1)
    *p=0
```

```
move $t0,$a0 ; p = adresa lui array[0]
Loop2: sw $zero,0($t0) ; Memorie[p] ← 0
        add $t1,$t0,4 ; p = p+4
        addi $t0,$t0,1 ; $t0 = 2 × dim
        add $t2,$t1,$t1 ; $t2 = 4 × dim
        slt $t3,$t0,$t2 ; $t3 = 1(p < &array[dim])
        bne $t3,$zero,Loop2; dacă(p < &array[dim]) atunci salt la Loop2
```

move \$t0,\$a0 ; p = &array[0]
sll \$t1,\$a1,2 ; \$t1 = dim × 4
add \$t2,\$a0,\$t1 ; \$t2 = &array[dim]
sw \$zero,0(\$t2); Memorie[p] ← 0

```
addi $t0,$t0,4 ; p = p+4
slt $t3,$t0,$t2 ; $t3 = (p < &array[dim])
bne $t3,$zero,Loop2; dacă(p < &array[dim]) atunci salt la Loop2
```

- EXEMPLUL 8. Sirurile alfa numerice, codificate ASCII, sunt stocate in memorie începând de la o adresa S(0), fiecare caracter ocupănd un byte; caracterele sunt în cod ASCII, iar ultimul caracter din sir, sfârșitul sirului este caracterul nul (Null terminate, 00H). Pentru astfel de siruri să se scrie o subrutină NCS (număr - caracter în sir) care adună cădăt este apelata determinată lungimea sirului. Subrutina NCS găsește adresa de început S(0) în registrul \$t0, iar returnarea numărului de caractere se face prin registrul \$V0 (instrucția lbu încarcă din memorie, într-un registru un curenț de un byte).

- programul principal (apelant)

(PC) move \$t0, \$t0 ; \$t0 ← S(0), în \$t0 se calculează adresa de început ale sir
 (PC) | NCS ; PC ← NCS, \$31 ← (PC+4)
 (PC+4) lstr. următoare ; Reîntoarcerea din subrutină se face la adresa (PC+4)

- subrutina NCS

NCS: move \$V0, \$zero ; \$V0 ← 0, contorul numărului de caractere se initializează
 Loop: lbu \$t1, 0(\$t0) ; \$t1 ← S(i) se încarcă un caracter din sir în \$t1
 beq \$t1, \$zero, IESIRE ; Dacă este caracterul null adună salt la IESIRE
 addi \$V0, \$V0, 1 ; incrementarea contorului
 addi \$t0, \$t0, 1 ; incrementarea adresei
 J Loop ; următorul caracter (următoarea iteratie)
 IESIRE: J \$t0 ; PC ← \$t0 (= PC+4), reîntoarcere în programul principal.

- EXEMPLUL 9. Dintre registrele \$t0 și \$t2 să se realizeze transferurile



sll \$t2, \$t2, 8 ;	<table border="1"> <tr> <td>B2</td> <td>B1</td> <td>B0</td> <td>0</td> <td>\$t2</td> </tr> </table>	B2	B1	B0	0	\$t2
B2	B1	B0	0	\$t2		
srl \$t2, \$t2, 16 ;	<table border="1"> <tr> <td>0</td> <td>0</td> <td>B2</td> <td>B1</td> <td>\$t2</td> </tr> </table>	0	0	B2	B1	\$t2
0	0	B2	B1	\$t2		
sll \$t2, \$t2, 8 ;	<table border="1"> <tr> <td>0</td> <td>B2</td> <td>B1</td> <td>0</td> <td>\$t2</td> </tr> </table>	0	B2	B1	0	\$t2
0	B2	B1	0	\$t2		
lui \$t1, 65280 ;	<table border="1"> <tr> <td>11111111000000000000000000000000</td> <td>\$t1</td> </tr> </table>	11111111000000000000000000000000	\$t1			
11111111000000000000000000000000	\$t1					
addi \$t1, \$t1, 255 ;	<table border="1"> <tr> <td>11111111000000000000000011111111</td> <td>\$t1</td> </tr> </table>	11111111000000000000000011111111	\$t1			
11111111000000000000000011111111	\$t1					
and \$t1, \$t1, \$t0 ;	<table border="1"> <tr> <td>1A200000000000000000000000000000</td> <td>\$t1</td> </tr> </table>	1A200000000000000000000000000000	\$t1			
1A200000000000000000000000000000	\$t1					
or \$t2, \$t2, \$t1 ;	<table border="1"> <tr> <td>A2B2B1A0</td> <td>\$t2</td> </tr> </table>	A2B2B1A0	\$t2			
A2B2B1A0	\$t2					

1.8 ETAPELE ÎN REALIZAREA UNUI PROGRAM EXECUTABIL

48

CAP. 1. Organizarea unui sistem pe bază de micropresor

1.8.7 Etapele în procesarea unui program.

- ASAMBLORUL - este programul calculator care translatăaza programul scris în limbaj de asamblare (program sursă sau fișier sursă) în program cod masină (program obiect). Aceasta translatoare este o măpare 1:1 (o instrucțiune în limbaj de asamblare nu este altceva decât o instrucțiune cod masină dor scrisă cu numeronice, cife și alte simboluri) Deoarece măparea este 1:1, deci o aplicatie bjectivă, există operația de asamblare căt ncea de dezasamblare (DEZASAMBLOR). Majoritatea asamblărilor produc cod obiect (cod masină) fără dorință treceri

CONTOR / ETICHETA, PROGRAM SURSA
ADRESA
(în decimal)

0 0 0 0	add \$t ₂ , \$t ₃ , \$t ₄
0 0 0 4	sw \$t ₂ , 20(%p)
0 0 0 8	bne \$t ₂ , \$t ₂ ACOLO
0 0 1 2	;
0 0 5 6	AICI ; move \$0, \$0
0 0 8 8	ACOLO) bne \$t ₂ , \$t ₃ AICI

TABELUL DE SIMBOLURI

Prima treccere		A doua treccere	
ETICHETA	ADRESA	ETICHETA	ADRESA
ACOLO	?	ACOLO	0088
AICI	0056	AICI	0056

Fiecare modul de program sursă (în limbaj de asamblare) se asamblă separat. Se utilizează un CONTOR care se incrementează cu 4 la fiecare instrucțiune care se translatăază. Contorul se inițializează cu 0000 la prima instrucțiune care este convertită în cod masină utilizând un tabel cu codurile în binar al tuturor instrucțiunilor din limbajul de asamblare al ap. De exemplu pentru prima instrucțiune de mai sus se obține:

add \$t₂, \$t₃, \$t₄ →

OPCODE	\$t ₃	\$t ₄	\$t ₂	shmt	funct
0	11	12	18	0	32

↓

000000	01011	01100	10010	00000	100000
--------	-------	-------	-------	-------	--------

 ⇒ 016C9020H

În urmă trece, la instrucțiunea a treia cu adresa 0008 (contor), pentru eticheta ACOLO (care este o adreșă) nu se poate substitui cu valoarea adresei (0088) deoarece programul nu a ajuns la iestr de eticheta ACOLO încă contorul să fi fost incrementat la val 0088. Această etichetă este trecută în tabelul de simboluri ca nezerovalo. În urmă sală la eticheta AICI din iestr de la adresa 0088 se poate rezolva deoarece astăzi cănd s-a ajuns la iestr de adresa 0056 eticheta AICI este substituită în tabel cu valoarea 0056. Eticheta ACOLO este zerovalo. La a doua treccere cănd se curioaste valoare adresei etichetei ACOLO, asamblatorul rezolvă doar etichetele interne nu și cele externe (din alte module)!

Gh. TOACSE – Arhitectura și Organizarea Microprocesoarelor

CAP 1. Organizarea unui sistem pe bază de microprocesor

49

Assembler Syntax

Comments in assembler files begin with a sharp sign (#). Everything from the sharp sign to the end of the line is ignored.

Identifiers are a sequence of alphanumeric characters, underscores (_), and dots (.) that do not begin with a number. Instruction opcodes are reserved words that *cannot* be used as identifiers. Labels are declared by putting them at the beginning of a line followed by a colon, for example:

```
.data
item: .word 1
      .text
      .globl main. # Must be global
main: lw    $t0, item
```

Numbers are base 10 by default. If they are preceded by 0x, they are interpreted as hexadecimal. Hence, 256 and 0x100 denote the same value.

Strings are enclosed in double quotes (""). Special characters in strings follow the C convention:

- newline \n
- tab \t
- quote \"

SPIM supports a subset of the MIPS assembler directives:

```
.align n   Align the next datum on a 2n byte boundary. For example,
           .align 2 aligns the next value on a word boundary.
           .align 0 turns off automatic alignment of .half, .word,
           .float, and .double directives until the next .data or
           .kdata directive.

.ascii str  Store the string str in memory, but do not null-terminate it.
.asciiz str Store the string str in memory and null-terminate it.
.byte bl.... bn  Store the n values in successive bytes of memory.
.data <addr> Subsequent items are stored in the data segment. If the
           optional argument addr is present, subsequent items are
           stored starting at address addr.
.double dl.... dn  Store the n floating-point double precision numbers
           in successive memory locations.
.extern sym size Declare that the datum stored at sym is size bytes
           large and is a global label. This directive enables the
           assembler to store the datum in a portion of the data segment
           that is efficiently accessed via register $gp.
.float fl.... fn  Store the n floating-point single precision numbers
           in successive memory locations.
.globl sym  Declare that label sym is global and can be referenced from
           other files.
.half hl.... hn  Store the n 16-bit quantities in successive memory
           halfwords.
.kdata <addr> Subsequent data items are stored in the kernel data segment.
           If the optional argument addr is present, subsequent items are
           stored starting at address addr.
.ktext <addr> Subsequent items are put in the kernel text segment. In
           SPIM, these items may only be instructions or words (see
           the .word directive below). If the optional argument addr
           is present, subsequent items are stored starting at address
           addr.
.set noat and .set at  The first directive prevents SPIM from
           complaining about subsequent instructions that use register
           $at. The second directive reenables the warning. Since
           pseudoinstructions expand into code that uses register
           $at, programmers must be very careful about leaving values
           in this register.
.space n   Allocate n bytes of space in the current segment (which
           must be the data segment in SPIM).
.text <addr> Subsequent items are put in the user text segment. In
           SPIM, these items may only be instructions or words (see
           the .word directive below). If the optional argument addr
           is present, subsequent items are stored starting at address
           addr.
.word wl.... wn  Store the n 32-bit quantities in successive memory
           words.
```

SPIM does not distinguish various parts of the data segment (.data, .rdata, and .sdata).

Identifier = Directiva

Etichetă: Directiva ; comandă în
(Directivele sunt instrucțiunile /
comenzi care pentru programul
asamblor nu și pentru AP.
Directivele nu sunt transilate
de asamblor în program
obiect pentru microprocesor)

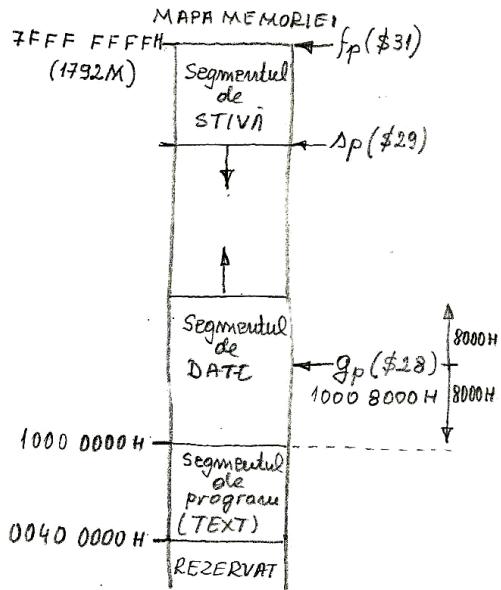
String Directive

Define the sequence of bytes produced by this directive:

```
.ascii "The quick brown fox jumps over the lazy dog"
```

```
.byte 84, 104, 101, 32, 113, 117, 105, 99
.byte 107, 32, 98, 114, 111, 119, 110, 32
.byte 102, 111, 120, 32, 106, 117, 109, 112
.byte 115, 32, 111, 118, 101, 114, 32, 116
.byte 104, 101, 32, 108, 97, 122, 121, 32
.byte 100, 111, 103, 0
```

$$T=84(\text{ASCII}); h=104(\text{ASCII}); e=101(\text{ASCII})$$



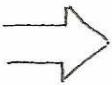
EXEMPLU: Subroutine pentru calculul sumei $\sum_{i=0}^{100} i^2$

PROGRAM SURSA
(in limbajul C)

```
#include <stdio.h>
int main (int argc, char *argv[])
{
    int i;
    int sum = 0;

    for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

COMPILEARE



```
.text
.align 2
.globl main
main:
    subu    $sp, $sp, 32
    sw     $ra, 20($sp)
    sd     $a0, 32($sp)
    sw     $0, 24($sp)
    sw     $0, 28($sp)

loop:
    lw     $t6, 28($sp)
    mul   $t7, $t6, $t6
    lw     $t8, 24($sp)
    addu  $t9, $t8, $t7
    sw     $t9, 24($sp)
    addu  $t0, $t6, 1
    sw     $t0, 28($sp)
    ble   $t0, 100, loop
    la     $a0, str
    lw     $a1, 24($sp)
    jal   printf
    move  $v0, $0
    lw     $ra, 20($sp)
    addu  $sp, $sp, 32
    j     $ra

.data
.align 0
str:
.ascii "The sum from 0 .. 100 is %d\n"
```

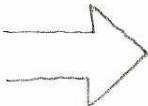
FIGURE A.4 The same routine written in assembly language with labels, but no comments. The commands that start with periods are assembler directives (see pages A-51–A-53). .text indicates that succeeding lines contain instructions. .data indicates that they contain data. .align n indicates that the items on the succeeding lines should be aligned on a 2^n byte boundary. Hence, .align 2 means the next item should be on a word boundary. .globl main declares that main is a global symbol that should be visible to code stored in other files. Finally, .asciiz stores a null-terminated string in memory.



⇒ Asamblarea este o operatie bijectiva

```
addiu $29, $29, -32
sw   $31, 20($29)
sw   $4, 32($29)
sw   $5, 36($29)
sw   $0, 24($29)
sw   $0, 28($29)
lw   $14, 28($29)
lw   $24, 24($29)
multu $14, $14
addiu $8, $14, 1
slli $1, $8, 101
sw   $8, 28($29)
mflo $15
addu $25, $24, $15
bne $1, $0, -9
sw   $25, 24($29)
lui $4, 4096
lw   $5, 24($29)
jal 1048 812
addiu $4, $4, 1072
lw   $31, 20($29)
addiu $29, $29, 32
jr   $31
move $2, $0
```

ASAMBLARE



```
0010011101111011111111111100000
10101111011111000000000000010100
10101111010010000000000000100000
1010111101001010000000000000100100
1010111101000000000000000011000
1010111101000000000000000011100
100011110101110000000000000011100
100011110101110000000000000011100
00000001110000000000000011000
0010010111001000000000000000001
0010010000000010000000001100101
1010111101010000000000000011100
0000000000000000111100000010010
00000011000011111100100000100001
001010000100000111111111110111
101011110111100100000000000011000
0011110000001000001000000000000
100011110101001010000000000011000
000011000010000000000000001110100
001001001000010000000010000110000
100011110111110000000000000010100
001001110111101000000000000010000
00000011111000000000000000000010000
00000000000000000000000000000010000
```

The same routine written in assembly language. However, the code for the routine does not label registers or memory locations nor include comments.

PROGRAM OBIECT

53

CAP. 1. Organizarea unui sistem pe bază de microprocesor

- DEZASAMBLORUL. Se poate realiza un program calculator dezasamblorul dezvoltat asamblarea este o mapeare 1:1 (aplicatie bijectivă) între multimea instrucțiunilor în limbaj de dezvoltare și multimea instrucțiunilor din cod masiv al microprocesorului. Dezasamblarea este operată în sensul opus dezvoltării. Nu aceeași proprietate există la compilarare, decodificatorul se poate reține foarte greu deoarece operația de compilarare nu este bijectivă.

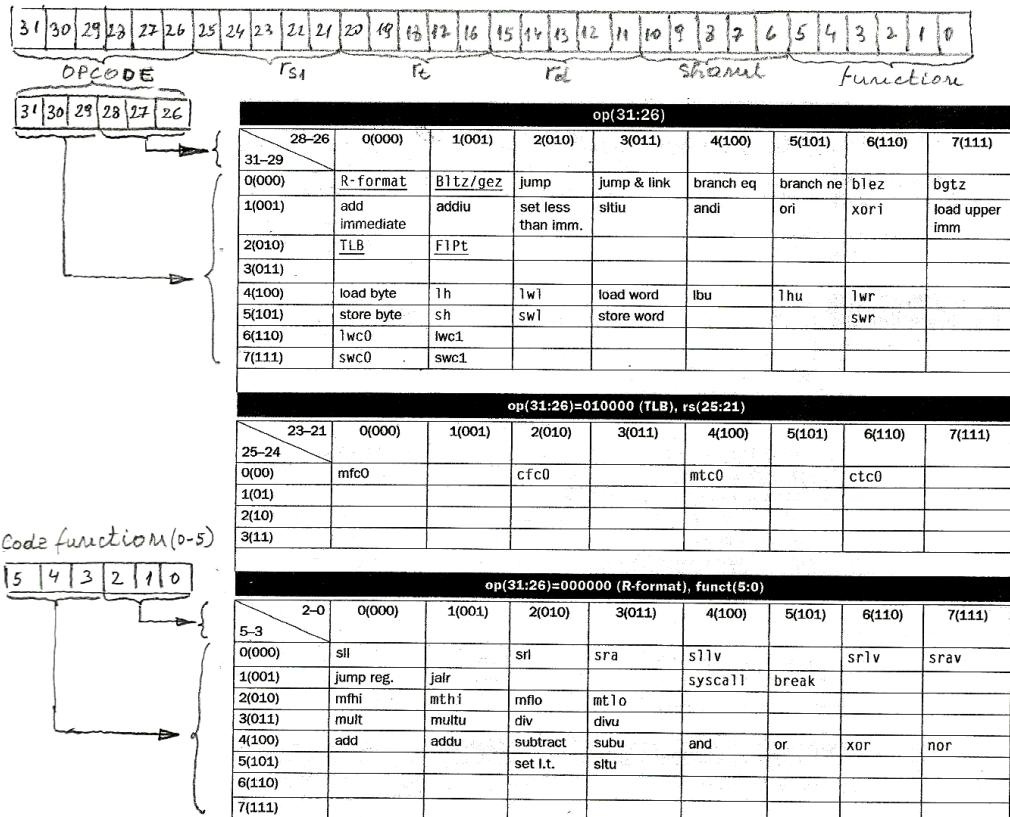


FIGURE 3.18 MIPS Instruction encoding. This notation gives the value of a field by row and by column. For example, in the top portion of the figure load word is found in row number 4 (100_{two} for bits 31–29 of the instruction) and column number 3 (011_{two} for bits 28–26 of the instruction), so the corresponding value of the op field (bits 31–26) is 100011_{two}. Underscore means the field is used elsewhere. For example, R-format in row 0 and column 0 (op = 000000_{two}) is defined in the bottom part of the figure. Hence subtract in row 4 and column 2 of the bottom section means that the funct field (bits 5–0) of the instruction is 100010_{two} and the op field (bits 31–26) is 000000_{two}. The F1PT value in row 2, column 1 is defined in Figure 4.48 on page 292 in Chapter 4. Bltz/gez is the opcode for four instructions found in Appendix A: bltz, bgez, bltzal, and bgezal. Instructions given in full name using color are described in Chapter 3, while instructions given in mnemonics using color are described in Chapter 4. Appendix A covers all instructions.

EXEMPLU. Pentru codul masiv (în hexazecimal) 00AF8020H să se determine instrucția care l-a generat.

1. Se converteste în binar și se scrie instrucția în format binar

OPCODE	R _{S1}	R _R	R _D	Shamt	funcție
0000000 = 0 ₁₀ ; 00101 = 5 ₁₀ ; 01111 = 15 ₁₀ ; 10000 = 16 ₁₀ ; 00000 = 0 ₁₀ ; 1000000 = 32 ₁₀					

2. Se separă cămpurile OPCODE, funcție, R_D, R_{S1}, R_R și shamt

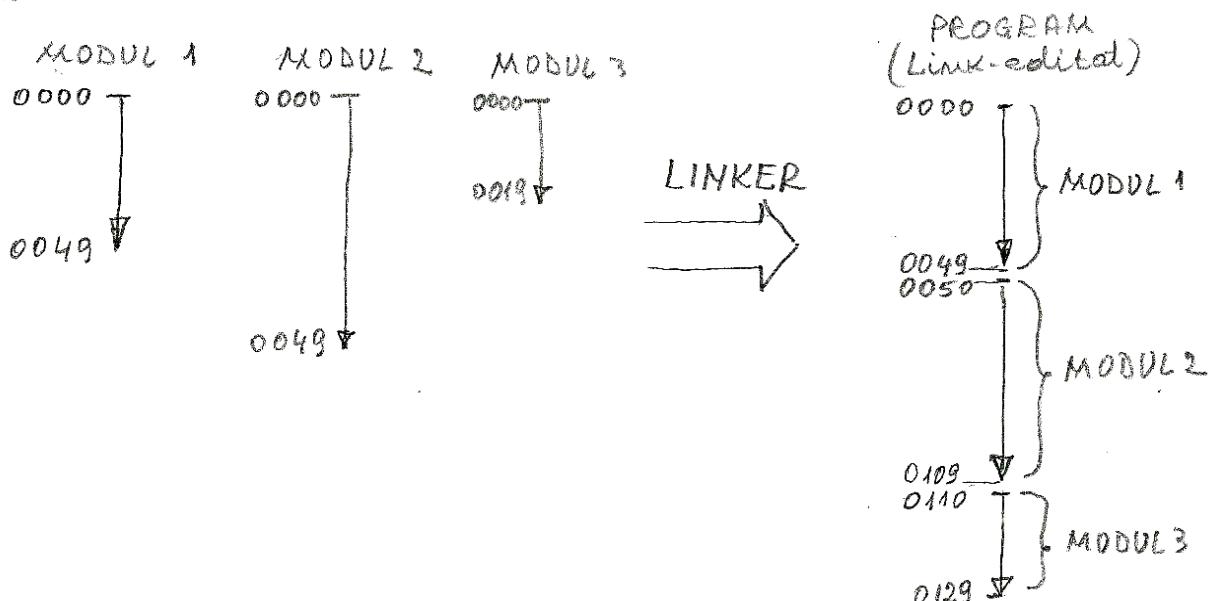
$$0000000 = 0_{10}; 00101 = 5_{10}; 01111 = 15_{10}; 10000 = 16_{10}; 00000 = 0_{10}; 1000000 = 32_{10}$$

3. Din tabelul de mai sus și din codul numeric al registrilor corelat cu convenția de alocare a registrelor rezultă instrucția add \$10, \$11, \$12.

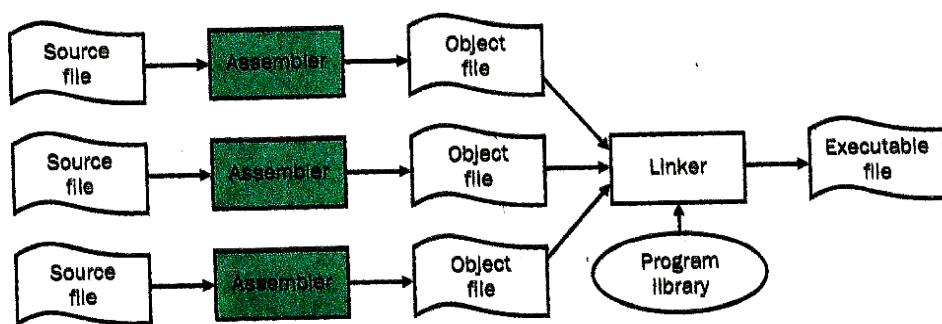
Gh. TOACSE – Arhitectura și Organizarea Microprocesoarelor

- **LINK-EDITORUL (LINKER)** - este programul calculator care funcționează modulele de cod obiect într-un singur program, program care este executabil pe masină (COD EXECUTABIL).

EXEMPLU



Programul LINKER realoca adresele în module, rezolvă toate etichetele externe, recalculă valoarea adreselor de sală, funcționează modulele din biblioteca de programe (necesare ie programul respectiv)



The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file..

- **Binary Translation** - Obținerea unui cod executabil pentru un procesor dintr-un cod executabil al unei alt procesor.

- INCARCĂTORUL (LOADER) - este programul calculator care preia codul executabil și îl încarcă în memoria calculatorului. Deoarece memoria operativă a calculatorului este partajată între mai multe programe care rulează pe calculator, încărcațorul găsește acele segmente de memorie unde poate fi încărcat programul (TEXT), DATELE necesare și segmentul de STIVĂ alocat.

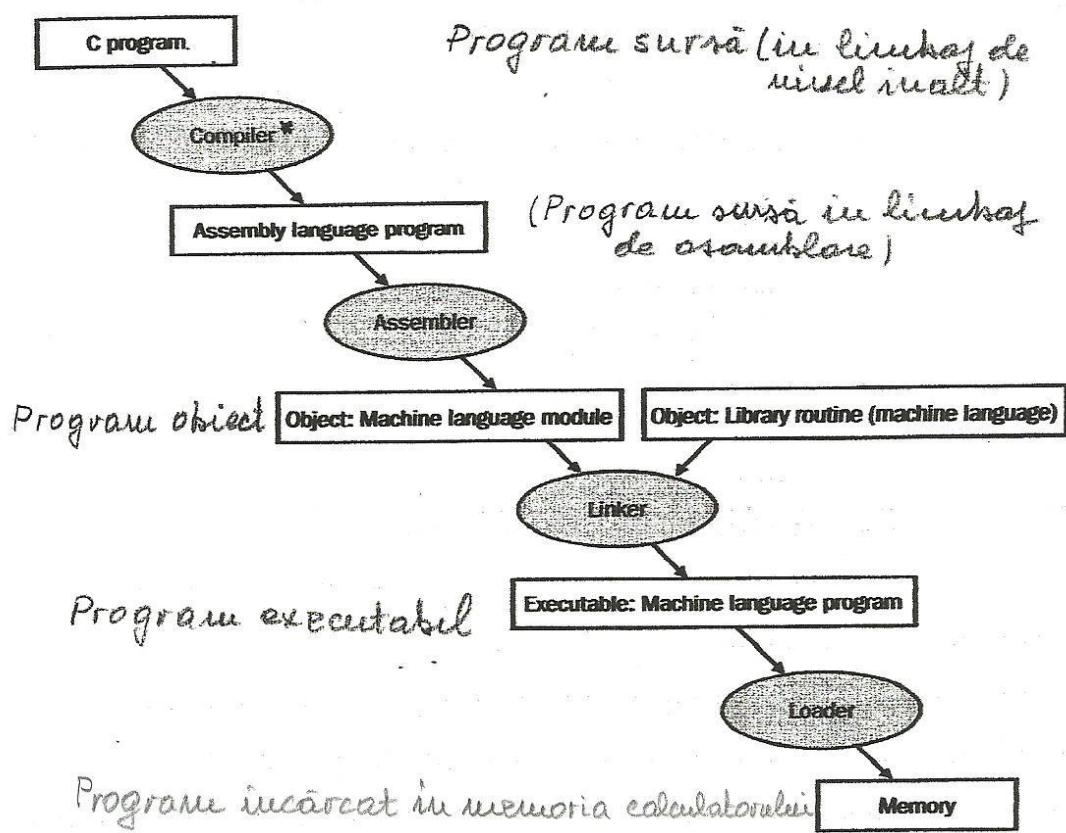
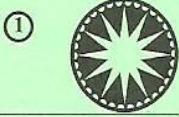


FIGURE 3.21 A translation hierarchy. A high-level-language program is first compiled into an assembly language program and then assembled into an object module in machine language. The linker combines multiple modules with library routines to resolve all references. The loader then places the machine code into the proper memory locations for execution by the processor. To speed up the translation process, some steps are skipped or combined together. Some compilers produce object modules directly, and some systems use linking loaders that perform the last two steps. To identify the type of file, Unix follows a suffix convention for files: C source files are named `x.c`, assembly files are `x.s`, object files are named `x.o`, and an executable file by default is called `a.out`. MS-DOS uses the suffixes `.TXT`, `.ASM`, `.OBJ`, and `.EXE` to the same effect.

* Compilatoarele actuale generează direcodul surșă (scris în limbaj de nivel înalt) programul obiect (cod masină), adică cuprinde și etapa de asamblare (asemblerul).

MIPS Reference Data



CORE INSTRUCTION SET

	NAME, MNEMONIC	MAT	FOR- OPERATION (in Verilog) (Hex)	OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1) 0 / 20 _{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2) 8 _{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2) 9 _{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]	0 / 21 _{hex}
And	and	R	R[rd] = R[rs] & R[rt]	0 / 24 _{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3) c _{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4) 4 _{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4) 5 _{hex}
Jump	j	J	PC=JumpAddr	(5) 2 _{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5) 3 _{hex}
Jump Register	jr	R	PC=R[rs]	0 / 08 _{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2) 24 _{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2) 25 _{hex}
Load Linked	ll	I	R[rt]=M[R[rs]+SignExtImm]	(2,7) 30 _{hex}
Load Upper Imm.	lui	I	R[rt]={imm, 16'b0}	f _{hex}
Load Word	lw	I	R[rt]=M[R[rs]+SignExtImm]	(2) 23 _{hex}
Nor	nor	R	R[rd] = ~ (R[rs] R[rt])	0 / 27 _{hex}
Or	or	R	R[rd] = R[rs] R[rt]	0 / 25 _{hex}
Or Immediate	ori	I	R[rt] = R[rs] ZeroExtImm	(3) d _{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	0 / 2a _{hex}
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0 (2)	a _{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0 (2,6)	b _{hex}
Set Less Than Unsigned	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6) 0 / 2b _{hex}
Shift Left Logical	sll	R	R[rd] = R[rt] << shampt	0 / 00 _{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shampt	0 / 02 _{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0) (2)	28 _{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (atomic) ? 1 : 0 (2,7)	38 _{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0) (2)	29 _{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt] (2)	2b _{hex}
Subtract	sub	R	R[rd] = R[rs] - R[rt] (1)	0 / 22 _{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt] (1)	0 / 23 _{hex}
			(1) May cause overflow exception	
			(2) SignExtImm = { 16{immediate[15]}, immediate }	
			(3) ZeroExtImm = { 16{1b'0}, immediate }	
			(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }	
			(5) JumpAddr = { PC+4[31:28], address, 2'b0 }	
			(6) Operands considered unsigned numbers (vs. 2's comp.)	
			(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic	

BASIC INSTRUCTION FORMATS

R	opcode	rs	rt	rd	shamt	funct		
	31 26 25	21 20	16 15	11 10	6 5	0		
I	opcode	rs	rt	immediate				
	31 26 25	21 20	16 15				0	
J	opcode	address						
	31 26 25							0

Copyright 2009 by Elsevier, Inc., All rights reserved. From Patterson and Hennessy, Computer Organization and Design, 4th ed.

ARITHMETIC CORE INSTRUCTION SET

(1) OPCODE
/ FMT / FT
/ FUNCT
(Hex)

NAME, MNEMONIC	MAT	FOR- OPERATION	OPCODE / FUNCT (Hex)
Branch On FP True	bc1t	FI if(FPcond)PC=PC+4+BranchAddr (4)	11/8/1--
Branch On FP False	bc1f	FI if(!FPcond)PC=PC+4+BranchAddr(4)	11/8/0--
Divide	div	R Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	0/-/-/1a
Divide Unsigned	divu	R Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt] (6)	0/-/-/1b
FP Add Single	add.s	FR F[fd] = F[fs] + F[ft]	11/10/-/0
FP Add	add.d	FR {F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}	11/11/-/0
Double	c.x.s*	FR FPcond = (F[fs] op F[ft]) ? 1 : 0	11/10/-/y
FP Compare Single	c.x.d*	FR FPcond = ({F[fs],F[fs+1]} op {F[ft],F[ft+1]}) ? 1 : 0	11/11/-/y
Double	*	(x is eq, lt, or le) (op is ==, <, or <=) (y is 32, 3c, or 3e)	
FP Divide Single	div.s	FR F[fd] = F[fs] / F[ft]	11/10/-/3
FP Divide	div.d	FR {F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}	11/11/-/3
FP Multiply Single	mul.s	FR F[fd] = F[fs] * F[ft]	11/10/-/2
FP Multiply	mul.d	FR {F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}	11/11/-/2
Double	sub.s	FR F[fd]=F[fs] - F[ft]	11/10/-/1
FP Subtract	sub.d	FR {F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}	11/11/-/1
Load FP Single	lwcl	I F[rt]=M[R[rs]+SignExtImm] (2)	31/-/-/-
Load FP	ldcl	I F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	35/-/-/-
Double			
Move From Hi	mfhi	R R[rd] = Hi	0/-/-/10
Move From Lo	mflo	R R[rd] = Lo	0/-/-/12
Move From Control	mfc0	R R[rd] = CR[rs]	10/0/-/0
Multiply	mult	R {Hi,Lo} = R[rs] * R[rt]	0/-/-/18
Multiply Unsigned	multu	R {Hi,Lo} = R[rs] * R[rt] (6)	0/-/-/19
Shift Right Arith.	sra	R R[rd] = R[rt] >> shampt	0/-/-/3
Store FP Single	swcl	I M[R[rs]+SignExtImm] = F[rt] (2)	39/-/-/-
Store FP	sdcl	I M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1] (2)	3d/-/-/-
Double			

FLOATING-POINT INSTRUCTION FORMATS

FR	opcode	fmt	ft	fs	fd	funct
	31 26 25	21 20	16 15	11 10	6 5	0
FI	opcode	fmt	ft	immediate		
	31 26 25	21 20	16 15			

PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	bit	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
\$zero	0	The Constant Value 0	N.A.
\$at	1	Assembler Temporary	No
\$v0-\$v1	2-3	Values for Function Results and Expression Evaluation	No
\$a0-\$a3	4-7	Arguments	No
\$t0-\$t7	8-15	Temporaries	No
\$s0-\$s7	16-23	Saved Temporaries	Yes
\$t8-\$t9	24-25	Temporaries	No
\$k0-\$k1	26-27	Reserved for OS Kernel	No
\$gp	28	Global Pointer	Yes
\$sp	29	Stack Pointer	Yes
\$fp	30	Frame Pointer	Yes
\$ra	31	Return Address	Yes

OPCODES, BASE CONVERSION, ASCII SYMBOLS

(3)

MIPS (1) MIPS	(2) MIPS	Binary	Deci-	Hexa-	ASCII	Deci-	Hexa-	ASCII
opcode	funct		mal	mal	Char-	mal	mal	acter
(31:26)	(5:0)				acter			acter
(1)	sll	add.f	00 0000	0 0	NUL	64 40	@	
		sub.f	00 0001	1 1	SOH	65 41	A	
j	srl	mul.f	00 0010	2 2	STX	66 42	B	
jal	sra	div.f	00 0011	3 3	ETX	67 43	C	
beq	slv	sqrt.f	00 0100	4 4	EOT	68 44	D	
bne		abs.f	00 0101	5 5	ENQ	69 45	E	
blez	srlv	mov.f	00 0110	6 6	ACK	70 46	F	
bgtz	srav	neg.f	00 0111	7 7	BEL	71 47	G	
addi	jr		00 1000	8 8	BS	72 48	H	
addiu	jalr		00 1001	9 9	HT	73 49	I	
slti	movz		00 1010	10 a	LF	74 4a	J	
sltiu	movn		00 1011	11 b	VT	75 4b	K	
andi	syscall	round.w.f	00 1100	12 c	FF	76 4c	L	
ori	break	trunc.w.f	00 1101	13 d	CR	77 4d	M	
xori		ceil.w.f	00 1110	14 e	SO	78 4e	N	
lui	sync	floor.w.f	00 1111	15 f	SI	79 4f	O	
(2)	mfhi		01 0000	16 10	DLE	80 50	P	
mthi			01 0001	17 11	DC1	81 51	Q	
mflo	movzf		01 0010	18 12	DC2	82 52	R	
mtlo	movnf		01 0011	19 13	DC3	83 53	S	
			01 0100	20 14	DC4	84 54	T	
			01 0101	21 15	NAK	85 55	U	
			01 0110	22 16	SYN	86 56	V	
			01 0111	23 17	ETB	87 57	W	
mult			01 1000	24 18	CAN	88 58	X	
multu			01 1001	25 19	EM	89 59	Y	
div			01 1010	26 1a	SUB	90 5a	Z	
divu			01 1011	27 1b	ESC	91 5b	[
			01 1100	28 1c	FS	92 5c	\	
			01 1101	29 1d	GS	93 5d]	
			01 1110	30 1e	RS	94 5e	^	
			01 1111	31 1f	US	95 5f	=	
lb	add	cvt.s.f	10 0000	32 20	Space	96 60	-	
lh	addu	cvt.d.f	10 0001	33 21	!	97 61	a	
lw	sub		10 0010	34 22	"	98 62	b	
lw	subu		10 0011	35 23	#	99 63	c	
lbu	and	cvt.w.f	10 0100	36 24	\$	100 64	d	
lhu	or		10 0101	37 25	%	101 65	e	
lwr	xor		10 0110	38 26	&	102 66	f	
	nor		10 0111	39 27	,	103 67	g	
sb			10 1000	40 28	(104 68	h	
sh			10 1001	41 29)	105 69	i	
swl	slt		10 1010	42 2a	*	106 6a	j	
sw	sltu		10 1011	43 2b	+	107 6b	k	
			10 1100	44 2c	,	108 6c	l	
			10 1101	45 2d	-	109 6d	m	
swr			10 1110	46 2e	.	110 6e	n	
cache			10 1111	47 2f	/	111 6f	o	
ll	tge	c.f.f	11 0000	48 30	0	112 70	p	
lwc1	tgeu	c.unf	11 0001	49 31	1	113 71	q	
lwc2	tlc	c.eqf	11 0010	50 32	2	114 72	r	
pref	tltu	c.ueqf	11 0011	51 33	3	115 73	s	
	teq	c.oltf	11 0100	52 34	4	116 74	t	
ldc1		c.ultf	11 0101	53 35	5	117 75	u	
ldc2	tne	c.olef	11 0110	54 36	6	118 76	v	
		c.ulef	11 0111	55 37	7	119 77	w	
sc		c.sff	11 1000	56 38	8	120 78	x	
swc1		c.nglef	11 1001	57 39	9	121 79	y	
swc2		c.seqf	11 1010	58 3a	:	122 7a	z	
		c.nglf	11 1011	59 3b	;	123 7b	{	
		c.lt,f	11 1100	60 3c	<	124 7c		
sdc1		c.ngef	11 1101	61 3d	=	125 7d	}	
sdc2		c.le,f	11 1110	62 3e	>	126 7e	~	
		c.ngtf	11 1111	63 3f	?	127 7f	DEL	

(1) opcode(31:26) == 0

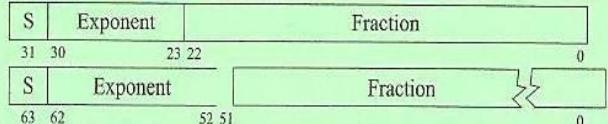
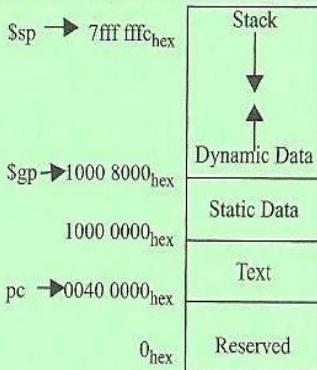
(2) opcode(31:26) == 17₁₀ (11_{hex}); if fmt(25:21) == 16₁₀ (10_{hex}) f = s (single); if fmt(25:21) == 17₁₀ (11_{hex}) f = d (double)
IEEE 754 FLOATING-POINT STANDARD

(4)

IEEE 754 Symbols

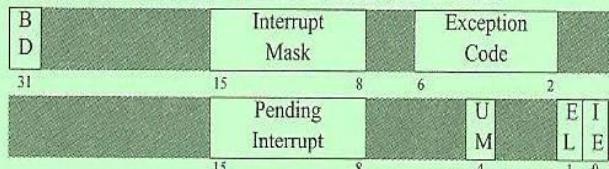
$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

where Single Precision Bias = 127,
Double Precision Bias = 1023.

IEEE Single Precision and Double Precision Formats:

MEMORY ALLOCATION

DATA ALIGNMENT

Double Word							
Word				Word			
Halfword		Halfword		Halfword		Halfword	
Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte
0	1	2	3	4	5	6	7

Value of three least significant bits of byte address (Big Endian)

EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS


BD = Branch Delay, UM = User Mode, EL = Exception Level, IE = Interrupt Enable

EXCEPTION CODES

Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented
6	IBE	Bus Error on Instruction Fetch	12	Ov	Arithmetic Overflow Exception
7	DBE	Bus Error on Load or Store	13	Tr	Trap
8	Sys	Syscall Exception	15	FPE	Floating Point Exception

SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

PRE-SIZE	FIX	PRE-SIZE	FIX	PRE-SIZE	FIX	PRE-SIZE	FIX
10 ³ , 2 ¹⁰	Kilo-	10 ¹⁵ , 2 ⁵⁰	Peta-	10 ⁻³	milli-	10 ⁻¹⁵	femto-
10 ⁶ , 2 ²⁰	Mega-	10 ¹⁸ , 2 ⁶⁰	Exa-	10 ⁻⁶	micro-	10 ⁻¹⁸	atto-
10 ⁹ , 2 ³⁰	Giga-	10 ²¹ , 2 ⁷⁰	Zetta-	10 ⁻⁹	nano-	10 ⁻²¹	zepto-
10 ¹² , 2 ⁴⁰	Tera-	10 ²⁴ , 2 ⁸⁰	Yotta-	10 ⁻¹²	pico-	10 ⁻²⁴	yocto-

The symbol for each prefix is just its first letter, except µ is used for micro.

CAP 2.

ARHITECTURA SETULUI DE INSTRUCȚIUNI, ISA (INSTRUCTION SET ARCHITECTURE)

2.1 COMPOENELE (ARHITECTURALE ALE) MICROPROCESORULUI

ARHITECTURA – referă acele atribute ale µP care sunt vizibile („imaginea”) de către programator (în limbaj de asamblare) sau de către scriitorul de compilator; adică acele atribute care au un impact direct asupra execuției logice a programului. Atributele arhitecturale sunt:

- setul de instrucții și contextul de utilizare a acestora;
- modurile de adresare și organizarea registrelor;
- structurarea datelor (pe cuvinte) și structurarea cuvintelor folosite de calculator;
- logica de accesare a memoriei;
- modalități de accesare I/O.

ISA este interfața între soft și hard!

- Definiții ale limbajului:
 1. Un sistem de simboluri /semne de comunicare între părți (foarte general).
 2. O mulțime finită sau infinită de siruri, definite pe un alfabet (Σ), (matematic).
 3. Un sistem de reguli, simboluri și cuvinte speciale utilizate pentru construcția unui program (informatic, limbaj de programare).

(Un limbaj, ca orice sistem de semne este dotat cu trei niveluri: sintactic, semantic și pragmatic)

- Instrucționea – descrierea unei acțiuni a procesorului cu ajutorul unui limbaj de programare cu următoarea sintaxă

$$i : OP \ D, \ S_1, \ S_2 \quad ; \quad D \leftarrow [S_1] \ OP \ [S_2]$$

unde:

i – eticheta instrucționii

OP – codul operației (OPCODE) efectuate de instrucție

D – argumentul/operandul destinație (data, registru, adresă)

S₁, S₂ – argumentele/operanții sursă (data, registru, adresă)

Instrucționă este ”celula” pe care o procesază un µP, care sub forma unui cuvânt binar are următoarea structură/câpuri:

Câmpul Operatiei (OPCODE)	Câmpul OPERANZI/ADRESE	Câmpul pentru specificarea adresei următoare (secvențialitatea)
------------------------------	------------------------	--

- OPCODE, este câmpul obligatoriu în orice instrucționă
- OPERANZI/ADRESE, în acest câmp se specifică operații sursă și operandul rezultat; în locul operanților se poate specifica locul, ADRESA) unde sunt plasări aceștia (acest câmp nu este obligatoriu în formatul instrucționii, există instrucții care nu au operanți, de exemplu NOP- No Operation, nici o operație).
- ADRESA INSTRUCȚIUNII URMĂTOARE, este câmpul care conține informația despre modul cum se realizează secvențialitate, adică unde se află instrucționă următoare. Informația constă în adresa instrucționii următoare, adresă care se înscrisează în PC, PC \leftarrow PC + k ; unde k este incrementul, în număr de adrese de memorie, de exemplu 1, 4, 8, la instrucționă următoare sau PC + k este o adresă fixată (sau

calculată) la o altă instrucțiune din program sub forma unui salt necondiționat (jump) sau condiționat (branch). Uneori, acest câmp poate lipsi din formatul unor instrucțiunii, deoarece adresa instrucțiunii următoare este implicită se obține din adresa instrucțiunii prezente plus un increment fix (+1, +4, +8). Regula de adresare corectă este: adresa în memorie **modulo k = 0**.

- Concepte. µP fiind o mașină matematică, arhitectura sa se bazează pe concepte din matematică în matematică transpusă în arhitectura µP

$$\begin{array}{lll} 1. \text{ Completitudinea} & \rightarrow & \text{Completitudinea} \\ 2. \text{ Regularitatea} & \rightarrow & \text{Ortogonalitatea} \end{array}$$

- **COMPLETITUDINEA.** Alegerea setului de instrucțiuni pentru un µP trebuie realizată încât să satisfacă toate aplicațiile care vor rula pe procesor și să genereze un cod optim. Această asemenea ar duce la un număr foarte mare de instrucțiuni/operatori în setul de instrucțiuni (ISA); deci se impune eliminarea uneia și reținerea altora ținând cont de următoarele trei criterii de alegere:

1. frecvența de utilizare a respectivei instrucțiuni în viitoarele aplicații;
2. eficiența instrucțiunii în utilizare (în scrierea programelor);
3. costul implementării instrucțiunii în ISA (suprafața de Si consumată pentru realizarea circuistică necesare instrucțiunii respective).

Prin simulare pe programe de test (benchmark) se poate determina frecvența medie de utilizare a fiecărei instrucțiuni din ISA, această frecvență de utilizare, de exemplu, este redată în tabelele următoare

Core MIPS	Name	Integer	Floating point	Arithmetic core + MIPS-32	Name	Integer	Floating point
add	add	0%	0%	FP add double	add.d	0%	8%
add immediate	addi	0%	0%	FP subtract double	sub.d	0%	3%
add unsigned	addu	7%	21%	FP multiply double	mul.d	0%	8%
add immediate unsigned	addiu	12%	2%	FP divide double	div.d	0%	0%
subtract unsigned	subu	3%	2%	load word to FP double	l.d	0%	15%
and	and	1%	0%	store word to FP double	s.d	0%	7%
and immediate	andi	3%	0%	shift right arithmetic	sra	1%	0%
or	or	7%	2%	load half	lh	1%	0%
or immediate	ori	2%	0%	branch less than zero	bltz	1%	0%
nor	nor	3%	1%	branch greater or equal zero	bgez	1%	0%
shift left logical	sll	1%	1%	branch less or equal zero	blez	0%	1%
shift right logical	srl	0%	0%	multiply	mul	0%	1%
load upper immediate	lui	2%	5%				
load word	lw	24%	15%				
store word	sw	9%	2%				
load byte	lb	1%	0%				
store byte	sb	1%	0%				
branch on equal (zero)	beq	6%	2%				
branch on not equal (zero)	bne	5%	1%				
jump and link	jal	1%	0%				
jump register	jr	1%	0%				
set less than	slt	2%	0%				
set less than immediate	slti	1%	0%				
set less than unsigned	sltu	1%	0%				
set less than imm. uns.	sltiu	1%	0%				

FIGURE 3.26 The frequency of the MIPS instructions for SPEC2000 integer and floating point. All instructions that accounted for at least 1% of the instructions are included in the table. Pseudoinstructions are converted into MIPS-32 before execution, and hence do not appear here. This data is from Chapter 2 of *Computer Architecture: A Quantitative Approach*, third edition.

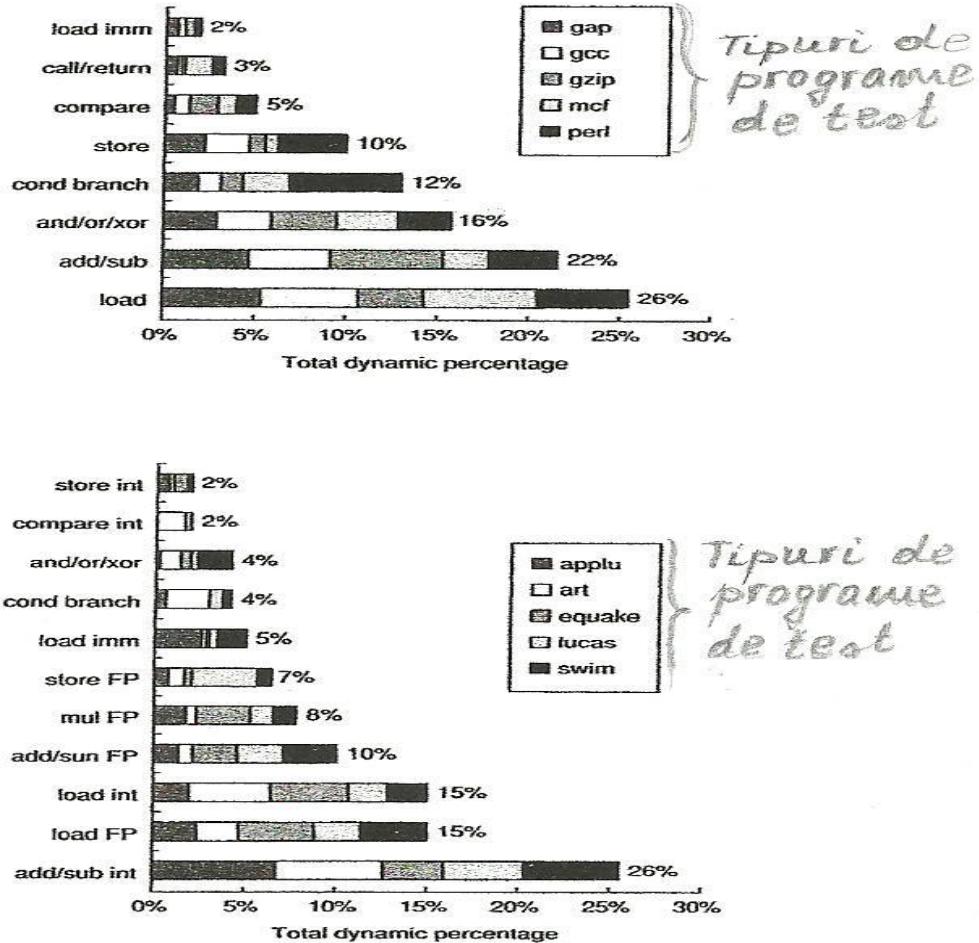


Figure 2.34 Graphical display of instructions executed of the five programs from SPECint2000 in Figure 2.32 (top) and the five programs from SPECfp2000 in Figure 2.33 (bottom). Just as in Figures 2.16 and 2.18, the most popular instructions are simple. These instruction classes collectively are responsible on average for 96% of instructions executed for SPECint2000 and 97% of instructions executed for SPECfp2000.

- ORTOGONALITATEA. Fiecare instrucție din ISA poate opera cu orice tip de date, aceasta accesată prin oricare mod de adresare, conținută în oricare registru, locație de memorie sau I/O. O arhitectură perfect ortogonală ar impune costuri de realizare ridicate, astfel că arhitecturile existente doar se apropie de o ortogonalitate perfectă.
- Un ISA reușit se măsoară prin numărul de procesoare relizate/vândute în arhitectura respectivă și printr-o longevitate de cel puțin 10-15 ani (IBM -360 (1964), INTEL 80X86 (1978); IA -64 (1999))

La ora actuală următoarele patru ISA sunt dominante: **x86** (Intel, AMD), **PowerPC** (Apple, IBM and Motorola), **MIPS** (Silicon graphics,MIPS Computer Systems, Siemens-NixdorfAcer, Digital Equipment Corporation, NEC) și **ARM** (ARM Holdings).

- Evoluția tipurilor de arhitecturi de μP :
 - CISC (Complex Instruction Set Computer), pornind din anii '60
 - RISC (Reduced Instruction Set Computer), pornind la începutul anilor '80
 - EPIC (Explicitly Parallel Instructin Computer) pornind la începutul anilor 2000
 - CPM (Cip Multiprocessors), după 2005

ORGANIZAREA – referă acele atribute legate de structurarea de nivel înalt a microprocesorului în părți componente. Atributele structurale sunt:

- structurare în CALEA DE DATE și CALEA DE CONTROL;
- organizarea din calea de date (ALU, shifter, registre, magistrale, etape de pipe);
- organizarea din calea de control (cablată, micropogramată, nanoprogramată);
- structurarea circuitelor acces la memorie și cache, și realizarea mapării între spațiul real și virtual;
- structurarea circuitelor de acces la I/O;
- structurarea sistemului de întreruperi;
- suport arhitectural pentru compilare și sistemul de operare.

Spre deosebire de atributele arhitecturale care sunt vizibile pentru programator, atributele de organizare sunt transparente pentru programator.

- Organizarea μP urmărește să estompeze gap-ul fundamental, adică: diferența dintre timpul de acces la memoria principală și timpul de acces în calea de date; raportul între acești timpi de acces (accesul la memorie/ accesul în calea de date) este mult supraunitar. În scopul estompării acestor mari diferențe organizarea memoriei se realizează pe niveluri și se mărește numărul de etape de procesare în pipe.

MICROARHITECTURA – referă proiectarea detaliată a circuisticii și implementarea într-o anumită tehnologie.

- Dezvoltările tehnologice. Legea lui (Gordon) Moore – 1965 ”Numărul de tranzistoare integrate pe unitatea de suprafață într-un circuit integrat se dublează în fiecare an”. După anii '90 această creștere, după puterile lui doi, s-a diminuat, ajungând la dublări la intervale de 1,5 ani sau chiar mai mult. Se prevede sfârșitul legii lui Moore prin 2018-2020, adică creșterea densității de integrare să înceeteze pe baza tehnologiei CMOS (ajungând la o lungime de canal de 6nm, respectiv la o tehnologie de integrare de 16-21 nm).

1. Tehnologia circuitelor integrate. Creșterea anuală numărului de componente pe chip este de 60-70% , frecvența de ceas crește cu un procent mai mic, de fapt începând cu 2005 frecvența de ceas nu mai prea depășește 3,5 GHz din cauza puterii disipate foarte mari ($P_d = C \cdot V^2 \cdot f$).

Pentru memorile DRAM densitatea de integrare are aceeași creștere pe când viteza de acces crește doar cam cu 30% în zece ani.

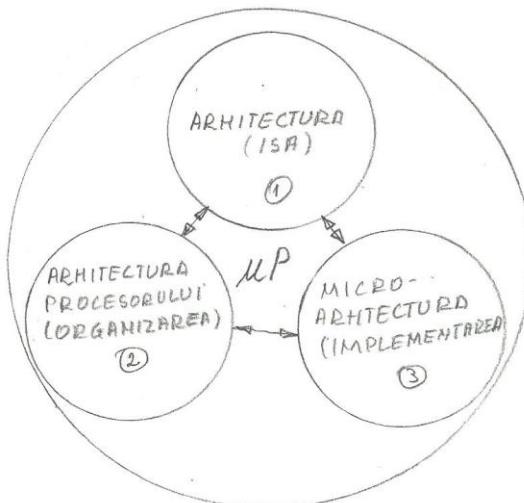
Tehnologia discurilor magnetice permite o cvadriplare a densității de împachetare cam la 3 ani, dar cu o creștere a vitezei de acces de 30% în zece ani.

- Instrumentele software de proiectare, tehniciile EDA (Electronic Design Automation). Creșterea posibilităților oferite de instrumentele software de proiectare este sub posibilitățile oferite de tehnologia de proiectare

Un procesor se proiecteză pentru o tehnologie care va fi state-of-the-art peste 1-2 ani!

- Actual, divizarea pe cele trei componente (arhitectura setului de instrucțiuni, organizarea/structurarea, microarhitectura) se păstrează, dar tot mai mult prin arhitectura microprocesorului se înțelege toate cele trei componente împreună. Această extensie a noțiunii de arhitectură de la (inițiala) Arhitectura Setului de Instrucțiuni se explică prin faptul că tot mai mult elaborarea ISA necesită o cunoaștere și participare și la elaborarea organizării și a microarhitecturii. Astfel că, actual, arhitectura microprocesorului acoperă toate cele trei componente

- Abordarea structurată a proiectării microprocesorului

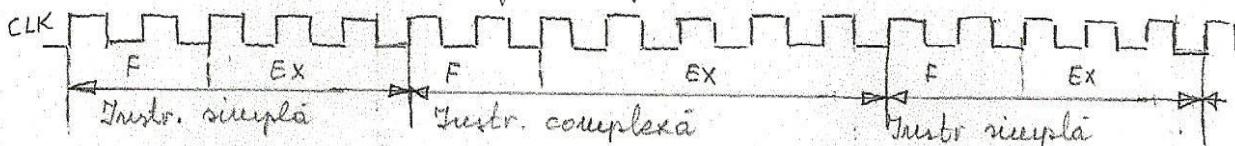


- O arhitectură, ISA, ① poate fi realizată pe organizări/structuri diferite (modelele diferite ale unor realizările)
- În felul organizării ② poate fi implementată în diferite tehnologii ③

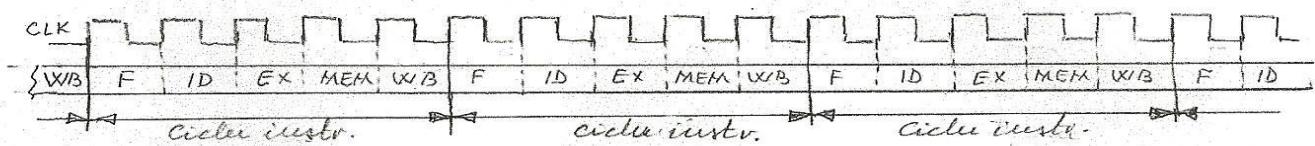
2.2 METRICI DE PERFORMANȚĂ PENTRU MICROPROCESSOR

Performanță (implicită!) pentru un μP este viteza de calcul, pentru că utilizatorul ar dori ca fiecare din problemele sale să fie rezolvate instantaneu. Evident că există și alte caracteristici, încărcă de viteză, care trebuie să fie cuantificate.

- O primă metrică pentru viteza unui micropresor ar putea fi **frecvența de ceas**, deoarece fiecare μP posedă un generator de semnal de ceas, iar fiecare instrucțiune este procesată pe durata unui anumit număr de tacte de ceas (în funcție de complexitatea instrucțiunii).



Procesorul MIPS are instrucțiuni care sunt procesate pe un număr fix de tacte de ceas (instrucțiunea este constituită din cinci etape (**IF**-Instruction Fetch; **ID** – Instruction Decode; **EX** – Execute; **MEM** – Memory acces); **WB** - Write back (înscrie rezultatul)).



- Mai adevarat, decât frecvența de ceas, pentru măsurarea vitezei μP este numărul de tacte pentru procesarea unei instrucțiuni **CPI (Cycle-Per-Instruction)**

$$CPI = \frac{\text{Numarul de tacte de ceas consumate pe program}}{\text{Numarul de instructiuni ale programului}} \left[\frac{\text{nr. tacte}}{\text{instructiune}} \right]$$

Pentru cazul procesării de tip pipeline și accesul la memorie (cache) se face într-un singur tact (iar miss rate este zero) se ajunge la $CPI = 1$, adică un tact pe instrucțiune. La mașinile superscalare CPI poate scădea sub valoare unu, când este mai potrivit să se utilizeze valoarea reciprocă a lui CPI, **IPC** = $1 / CPI$, adică numărul de instrucțiuni procesate pe un tact (**Instruction Per Cycle**).

- O altă metrică este **timpul consumat de CPU** pentru rularea unui program, **T_{CPU}**

$$T_{CPU} \left[\frac{\text{secunde}}{\text{program}} \right] = Nr \left[\frac{\text{instructiuni}}{\text{program}} \right] \times CPI \left[\frac{\text{tacte}}{\text{instructiune}} \right] \times T_{clk} \left[\frac{\text{secunde}}{\text{tact}} \right]$$

iar când procesorul are instrucțiuni cu lungimi diferite de ciclu instrucțiune, deci valori diferite pentru CPI (i- tipul de instrucțiune) relația anterioară devine

$$T_{CPU} = \sum_i (CPI_i \cdot N_{\text{de instructiuni } i}) \cdot T_{clk}$$

Pe baza relației anterioare poate se poate analiza cum timpul de procesare (T_{CPU}) poate fi influențat de deciziile arhitecturale

Caracteristica/ Parametru	Este influențată de :
Nr de instrucțiuni	Arhitectura Setului de Instrucțiuni (ISA), Compilator
CPI	Arhitectura Setului de Instrucțiuni (ISA), Organizarea μ P
Tclk	Tehnologia de implementare, Organizarea μ P

- **MIPS** (Million- Instructions-Per-Second)

$$\text{MIPS} \left[\frac{\text{Nr instr} \cdot 10^{-6}}{\text{secunde}} \right] = \text{Nr} \left[\frac{\text{instr} \cdot 10^{-6}}{\text{program}} \right] \cdot \frac{1}{T_{CPU} \left[\frac{\text{secunde}}{\text{program}} \right]} \rightarrow T_{CPU} = \frac{\text{Nr de instr} \cdot 10^{-6}}{\text{MIPS}}$$

sau

$$\text{MIPS} = \frac{\text{Nr de instr/program}}{T_{CPU}} = \frac{(\text{Nr de instr/program}) \cdot 10^{-6}}{(\text{Nr de instr/program}) \cdot \text{CPI} \cdot T_{CLK}} = \frac{10^{-6}}{\text{CPI} \cdot T_{CLK}} = \frac{f[\text{MHz}]}{\text{CPI}}$$

- MIPS-ul este o metrică intuitivă (dar de utilitate comercială), un procesor care reizează un MIPS mai ridicat apare ca fiind mai rapid, dar nu este o metrică profesională.
- MIPS-ul este dependant de setul de instrucțiuni al procesorului, dar pentru comparație între procesoare nu se poate utiliza deoarece procesoarile au seturi de instrucțiuni diferite.
- Chiar pe același procesor valoarea MIPS nu este concludentă deoarece de la program la program proporția între diferite tipuri de instrucțiuni se modifică.

- **MFLOPS** (Million-Floatingpoint-Operation-Per-Second)

$$\text{MFLOPS} = \frac{(\text{Nr de operatii în virgula flotanta}) \cdot 10^{-6} / \text{program}}{T_{CPU} / \text{program}}$$

- Această metrică se aplică doar pentru programele care conțin operații în virgulă flotantă (de exemplu, compilatoarele nu conțin).
- În seturile de instrucțiuni ale microprocesoarelor nu există aceleași instrucțiuni în virgulă flotantă (dar compararea microprocesoarelor se face prin numărul de instrucțiuni FP existente în program și nu la numărul de operații OP, unele operații FP sunt simulate prin alte instrucțiuni).
- **BENCHMARK SUITS**- pachete de programe de test. S-au elaborat pachete de programe de test SPEC (Standard Performance Evaluation Corporation) special concepute pentru testarea: calculatoare, servere, embedded systems.

Pentru un procesor care rulează doar o singură aplicație (ar fi cazul unor embedded systems) se poate alege procesorul care realizează T_{CPU} cel mai scurt. Pentru microprocesoare de uz general pe care se rulează o multitudine de tipuri de programe alegerea celui mai bun μ P este dificil de realizat, în concluzie un μ P de uz general trebuie să fie proiectat încât să realizeze performanțe cvasi-optime pentru gamă largă de tipuri de programe.

Din punct de vedere al utilizatorului unui calculator este important cât de repede este executat programul său. Dar acest timp depinde de mulți factori din sistem (memorie, I/O, sistem de operare) de modul cum sunt aceștia corelați și numai de performanțele microprocesorului. De exemplu, acest timp de rulare pe calculator/sistem depinde în primul rând de modul cum nivelurile de memorie (cache, principală) pot asigura alimentarea

procesorului cu penalizări minime apoi depinde de: sistemul de operare (care versiune de Windows sau Linux) de compilator (cât de bine acesta generează un cod optim) și de networking conditions (dacă acel calculator partajează resursele I/O din rețea).

"Anyone can built a fast CPU. The trick is to build a fast system"

Seymour Cray (considered the father of the supercomputer)

EXEMPLUL 2.1 Se consideră două calculatoare A și B care au același set de instrucțiuni. Calculatorul A are un ceas cu perioada $T_{CLKA} = 250\text{ps}$ ($f_{CLKA} = 4\text{GHz}$) și $CPI_A = 2$ pentru un anumit program iar calculatorul B pentru același program are $T_{CLKB} = 500\text{ps}$ ($f_{CLKB} = 2\text{GHz}$) și $CPI_B = 1,2$. Care din calculatoare rulează mai repede acel program (care are numărul de instrucțiuni, Nr. instr) și de câte ori.

Soluție.

$$TCPUA = \text{Nr. instr} \cdot CPI_A \cdot T_{CLKA} = \text{Nr. instr} \cdot 2 \cdot 250 \text{ ps}$$

$$TCPUB = \text{Nr. instr} \cdot CPI_B \cdot T_{CLKB} = \text{Nr. instr} \cdot 1,2 \cdot 500 \text{ ps}$$

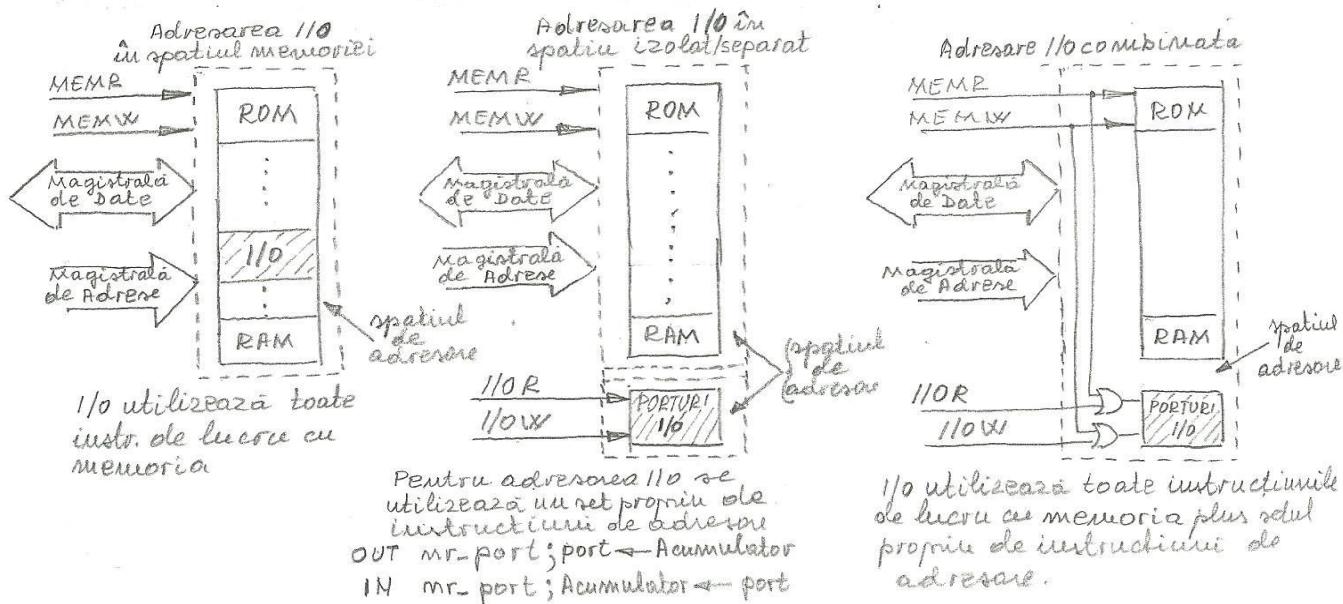
$$\frac{\text{CPU}_{\text{performante A}}}{\text{CPU}_{\text{performante B}}} = \frac{T_{CPUB}}{T_{CPUA}} = \frac{\text{Nr.instr} \cdot 1,2 \cdot 500\text{ps}}{\text{Nr.instr} \cdot 2 \cdot 250\text{ps}} = \frac{600}{500} = 1,2$$

Calculatorul A este de 1,2 ori mai rapid decât calculatorul B pentru acest program.

2.3 SPAȚIUL DE ADRESARE.

Fiecare element din sistem, ca să poată fi adresat de μP , trebuie să aibă o adresă (un element fără adresă nu există în sistem!). Totalitatea adreselor pe care le acceseză μP în sistem constituie spațiul de adresare, adică toatalitatea adreselor memoriei și adreselor componentelor I/O. După modul cum se alocă adresele componentelor de I/O în raport cu cele ale memoriei există trei variante de adresare în sistem:

1. Adresarea I/O în spațiul memoriei, se realizează când capacitatea de adresare a memoriei este 2^n , n fiind numărul de biți în cuvântul de adresă, $A = A_{n-1}A_{n-2}\dots A_1A_0$, iar un număr din aceste adrese sau un segment din spațiul de adresare este alocat pentru componente de I/O. Avantajul acestui mod de alocare a adreselor în sistem constă în faptul că toate instrucțiunile din setul de instrucțiuni care lucrează cu memoria pot fi utilizate și pentru lucru cu I/O, dar apare și dezavantajul, minor, că prin această alocare se consumă o parte din adresele utilizate de memorie.
2. Adresarea I/O în spațiul izolat. Prin această modalitate de alocare a adreselor spațiul de adresare pentru memorie este de 2^n , iar pentru I/O se crează un subspațiu separat de cel al memoriei. Pentru componente I/O nu se mai utilizează instrucțiunile care lucrează cu memoria ci în setul de instrucțiuni există instrucțiuni speciale pentru lucru cu I/O; două dintre cele mai uzuale de instrucțiuni speciale pentru I/O sunt:
 $in\ porti$; μP citește conținutul portului I/O de adresă $porti$ (IN 37; citește portul de adresă 37)
 $out\ portk$; μP înscrie în portul I/O de adresă $porti$ (OUT 25; inscrie în portul de adresă 25)
Numărul de adrese din subsetul de sdrese I/O este mult mai mic decât adresele pentru memorie (2^n).
3. Adresare I/O combinată. Pentru I/O se pot utiliza atât modul/instrucțiunile de lucru cu memoria cât și instrucțiunile dedicate numai pentru I/O.



EXEMPLUL 2.2. Să se implementeze un modul de memorie, adresabil cu un cuvânt de 16 biți,

$A = A_{n-1}A_{n-2}\dots A_1A_0$ (spațiul de adresare $2^{16} = 16K$) care cuprinde:

1. o componentă ROM formată din patru circuite de capacitate 4KB, (4x4KB); ROM#8, ROM#7, ROM#6, ROM#5.
2. o componentă RAM formată din patru circuite de capacitate 1KB, (4x1KB); RAM#4, RAM#3, RAM#2, RAM#1.
3. două porturi PORT1 (input port), PORT2 (output port) pentru care se alocă în spațiul memoriei un segment de 4K adrese.

Soluție.

– ROM. Fiecare chip ROM din modul utilizează, în comun, pentru decodificarea celor 16K adrese, biții $A_{11} - A_0$, ($4K \rightarrow 2^{12}$) iar biții superiori $A_{15} - A_{12}$ vor fi utilizați pentru selectarea celor patru circuite ROM (dintre aceștia vor fi diferenți, de la circuit la circuit, doar biții A_{13} și A_{12}). Segmentul de adrese care este repartizat pentru ROM este de la 48K (C000H) la 64K (FFFFH).

– RAM. Fiecare chip RAM utilizează, în comun, pentru decodificarea celor 4K adrese, biții $A_9 - A_0$ ($1K \rightarrow 2^{10}$) iar biții superiori $A_{15} - A_{10}$ vor fi utilizați pentru selectarea celor patru circuite RAM; de fapt vor fi diferenți, de la circuit la circuit, numai biții A_{11} și A_{10} . Segmentul de adrese de 4K este de la 0K (0000H) la 4K (0FFFH).

– I/O. Se alocă un segment de 4K adrese ($A_{11} - A_0$); cu biții superiori $A_{15} - A_{12}$ se pot selecta 16 segmente de 4K adrese. Se alege pentru I/O segmentul de 4K selectabil cu $A_{15}A_{14}A_{13}A_{12} = 1000$, de la 32K la 36 K, adresele din acest segment cu $A_2 = 0$ vor selecta portul de ieșire (PORT2), iar adresele cu $A_2 = 1$ vor selecta portul de intrare (PORT1).

Pentru selectare se vor utiliza circuite decodificatoare DCD3:8, 74XX138.

- Mapa memoriei pentru modulul de memorie este

BITII CU VÂNTULUI DE ADRESARE																		
SPATIUL DE ADRESARE		Biți de selectare																
64K ADRESE		A ₁₅	A ₁₄	A ₁₃	A ₁₂	A ₁₁	A ₁₀	A ₉	A ₈	A ₇	A ₆	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀	
FFFFH	Segmentul alocat pentru memoria ROM	1	1	-	-	1	1	1	1	1	1	1	1	1	1	1	1	Chip-ul ROM #8 F000H - FFFFH
32K adrese		-	-	1	1	-	-	-	-	-	-	-	-	-	-	-	Chip-ul ROM #7 E000H - EFFFH	
16K adrese		1	1	1	0	-	-	-	-	-	-	-	-	-	-	-	Chip-ul ROM #6 D000H - DFFFH	
48K C000H		-	-	0	1	1	1	1	1	1	1	1	1	1	1	1	1	Chip-ul ROM #5 C000H - CFFFH
BFFFH	Zonă nealocată 12K adrese	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3 segmente de 4K adrese rămasă nealocate
36K 9000H		-	-	1	0	0	0	0	0	0	0	0	0	0	0	0	0	
8FFFH	Segmentul alocat pentru porturi I/O	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	INPUT PORT (A ₂ =1)
		-	-	1	0	x	x	x	x	x	x	x	x	x	x	x	OUTPUT PORT (A ₂ =0)	
32K	8000H	4K adrese	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	INPUT PORT (A ₂ =1)
	7FFFH	Zonă nealocată 28K adrese	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	OUTPUT PORT (A ₂ =0)
4K	1000H		0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	
32K adrese	Segmentul alocat pentru memoria RAM	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	chip-ul RAM #4 0C00H - OFFFH
		-	-	-	1	1	-	-	-	-	-	-	-	-	-	-	chip-ul RAM #3 0B00H - OBFFH	
		0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	chip-ul RAM #2 0400H - 07FFH
0	0	0	0	0	1	0	1	-	-	-	-	-	-	-	-	-	chip-ul RAM #1 0000H - 03FFH	
		-	-	-	0	0	-	-	-	-	-	-	-	-	-	-		

Bitii de selectare

- Selectare ROM.

- Selectarea unei adrese în fiecare circuit ROM se realizează cu biții A₁₁ - A₀

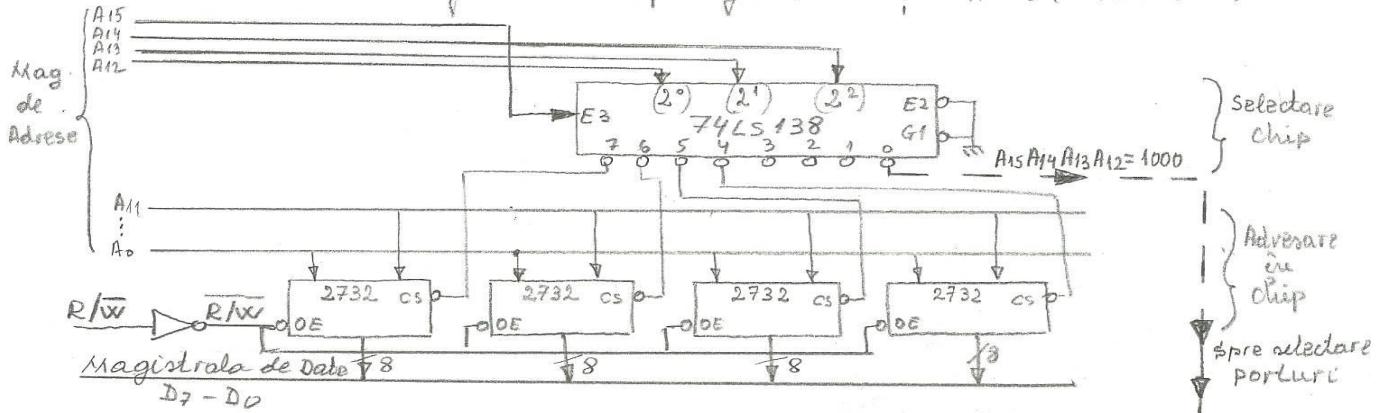
- Selectarea fiecărui circuit ROM se realizează cu biții A₁₅A₁₄A₁₃A₁₂ = 11xx , subsegmentele de adrese sunt:

ROM#8 ⊂ (F000H, FFFFH), ROM#7 ⊂ (E000H,EFFFH), ROM#6 ⊂ (D000H,DFFFH),

ROM#5 ⊂ (C000H,CFFFH).

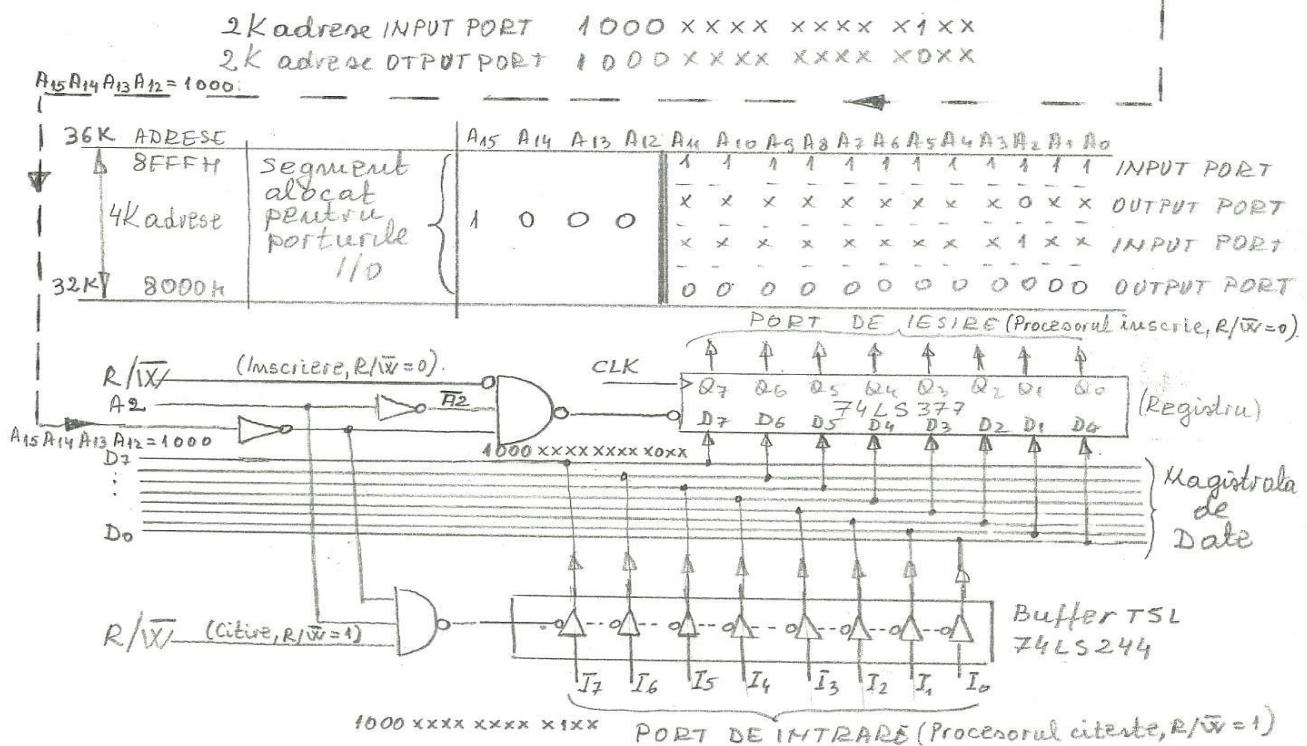
BİȚII CU VÂNTULUI DE ADRESĂ																	
64K	ADRESE	selectare chip				Adresarea în cadrul chip-ului											
		A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
FFFFH	Segment alocat pentru ROM	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
16K adrese		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16K adrese		1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
16K adrese		0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
48K	C 000H	00	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Selectarea chip-urilor se face cu cei patru biți superioiri A₁₅ A₁₄ A₁₃ A₁₂ (A₁₅ A₁₄ = 11)
- Adresarea în cadrul fiecărui chip se face cu biții A₁₁ - A₀ (4K adrese)



• Selectare porturi

- Selectare port de intrare (citire), PORT1, se realizează cu A₁₅ - A₀ = 1000 xxxx xxxx xx1xx
- Selectare port de ieșire (înscriere), PORT2, se realizează cu A₁₅ - A₀ = 1000 xxxx xxxx xx0xx



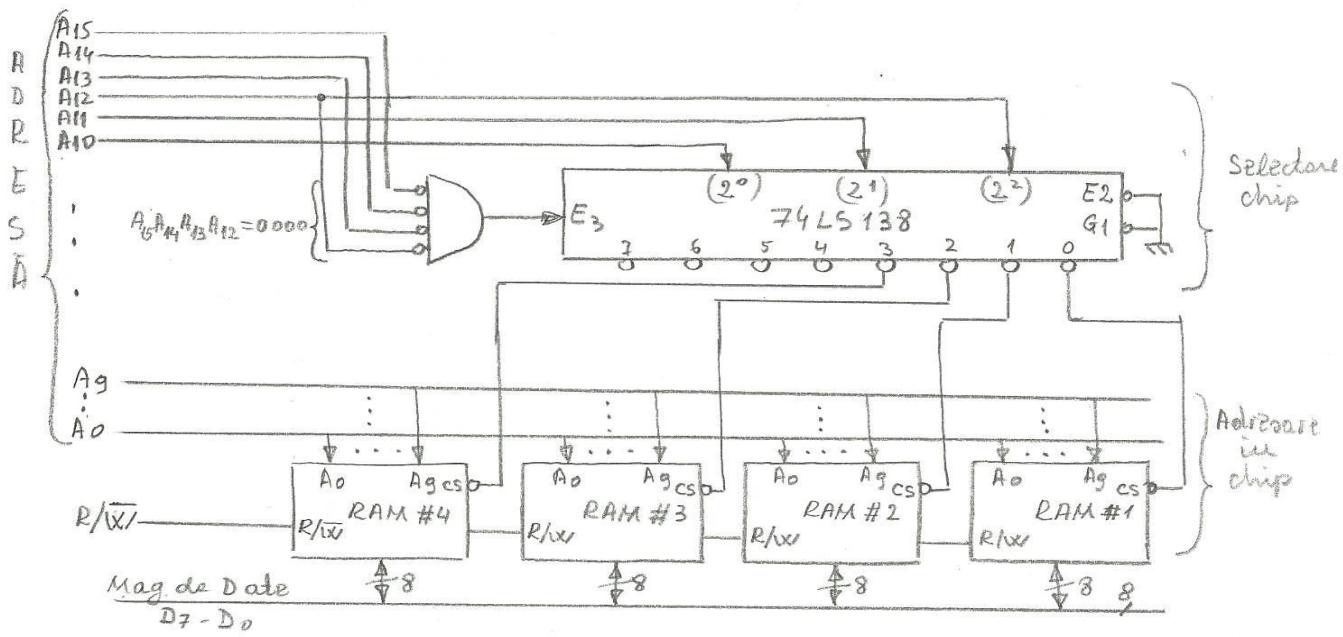
• Selectare RAM

- Selectarea unei adrese în fiecare circuit RAM se realizează cu biții A₉ - A₀ (1K adrese)
- Selectarea fiecărui circuit RAM se realizează cu biții A₁₅ A₁₄ A₁₃ A₁₂ A₁₁ A₁₀ = 0000xx , subsegmentele de adrese sunt: RAM#4 ⊂ (0C00H, 0FFFH), RAM#3 ⊂ (0800H, 0BFFH), RAM#2 ⊂ (0400H, 07FFH), RAM#1 ⊂ (0000H, 03FFH).

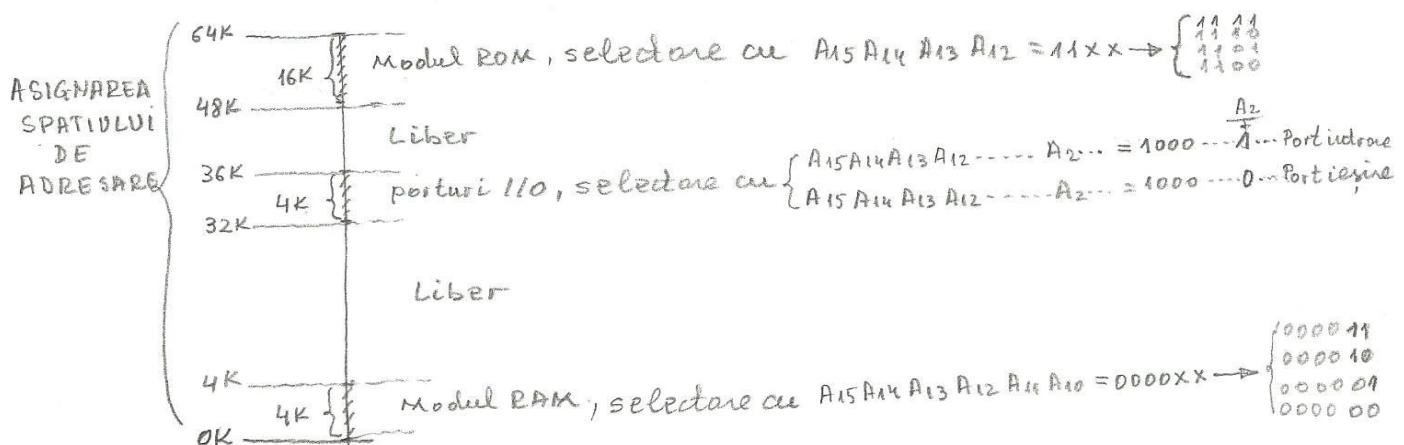
BITII CUVANTULUI DE ADRESĂ

Bitii de selecțare chip Adresare în cadrul chip-ului									
A15 A14 A13 A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0									
4K ADRESE	OFFFH	Segment alocat pentru memoria ROM	0 0 0 0	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	chip-ul RAM#4	000H-0FFFH			
4K adrese	4K			---	---	---	---	---	---
OK	0000H			0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	chip-ul RAM#3	0COOH-0FFFH			
				1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	chip-ul RAM#2	0800H-0BFFFH			
				1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0	chip-ul RAM#1	0400H-0700H			
				0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	chip-ul RAM#1	0000H-03FFFH			

- Selectarea chip-urilor se face cu cei sase biti superioiri $A_{15} A_{14} A_{13} A_{12} A_{11} A_{10}$ ($A_{15} A_{14} A_{13} A_{12} A_{11} A_{10} = 0000xx$)
- Adresarea în cadrul fiecărui chip se face cu bitii $A_{11} A_{10}$ (1K adrese)



- Mapa spațiului de adresare asignat pentru modulul de memorie



2.4 MODURI DE ADRESARE

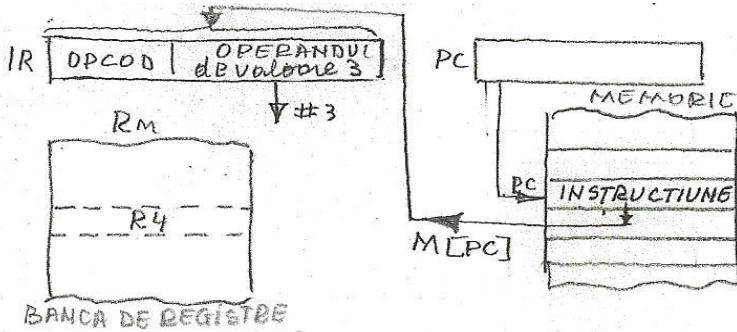
Instrucțiunea adusă în µP, din segmental text al memoriei, în etapa de FETCH, este decodificată și în etapele următoare din subciclul EXECUTE va fi procesată conform codului operației, OPCODE. Majoritatea instrucțiunilor pentru a fi procesate necesită operanzi, operanzi care se găsesc stocați în registre, locații de memorie sau în porturile I/O; deci este necesar ca acești operanzi să fie aduși în procesor (calea de date) pentru efectuarea procesării. *Modalitatea de identificare a locului unde sunt plasați operanzi, citirea lor și aducerea lor în calea de date pentru procesare este referată prin MOD DE ADRESARE.* Informația necesară pentru a realiza modul de adresare pentru o instrucțiune trebuie să fie conținută, explicit sau implicit, în instrucțiunea reapectivă. Aducerea unui operand plasat în banca de registre, în raport cu un operand plasat în memorie, prezintă avantajul unui timp de acces de până la un ordin de măririme mai mic și, în plus, în corpul instrucțiunii un subcâmp de lungime scurtă (un număr redus de biți), pentru specificarea numărului de registru de accesat. Pentru accesarea unui registru dintr-o bancă de 2^n registre este necesar un subcâmp de n biți în corpul instrucțiunii, pe când pentru accesarea unei locații de memorie dintr-un spațiu de memorie de 2^{n_1} adrese în corpul instrucțiunii subcâmpul de specificare are lungimea de n_1 biți ($n_1 >> n$), de exemplu $n_1 = 32$ sau 64biți (în general, neexistând un astfel de număr de biți disponibil în corpul instrucțiunii)

Denumirea de registru general, **GRP (General Purpose Register)** indică faptul că un registru poate fi utilizat atât pentru păstrarea unui operand cât și pentru păstrarea unei adrese necesare accesării unei date (pointer) din memorie. Unele procesoare au anumite registre dedicate pentru pointeri (de exemplu, la MIPS: \$gp = \$28- pointer pentru segmentul de date; \$sp = \$29- pointerul pentru indicarea vârfului stivei; \$fp = \$30- pointerul pentru indicarea bazei stivei).

În continuare se vor prezenta modurile de adresare fundamentale pentru instrucțiunile unui µP.

1. ADRESAREA IMEDIATĂ (sau literală) Adresa operandului este specificată (imediat) în corpul instrucțiunii, de exemplu:

Add \$R4, #3 ; \$R4←\$R4+3, instrucțiune conține (imediat) al doilea operand (3) în corpul său



- Adresarea operaareului odată ce instrucțiunea, slăci prim PC.
 - Adresare utilizată pentru introducerea de constante în programe.
 - Adresare inefficientă d.p.d.v al formulelor de instrucții.
 - Adresarea doar a constantelor de valoare care intăiază subcărereșul imediat.

EXEMPLU: În MIPS addi, addiu, slti, sltiu, andi, ori, xori, lui. Liteau i indică un adresat în corpul instrucțiunii, iar litera **u** specifică faptul că operația se realizează cu numere fară semn (unsigned).

Pentru că imediatele din corpul înstructiunii este numărul de 16 biti ($I_{15} - I_0$) și operațiile se fac pe 32 biti, imediatul înainte de utilizare ca operand este extins la 32 biti astfel că bitul I_{15} care reprezintă semnul imediatului (în complementul de 2) se multiplică de 16 ori, de exemplu pentru un imediat negativ

I_{15} I_0 I_{31} I_{15} I_0 $[I_{15}]^{16} \times [I_{15} - I_0]$
~~1010 1110 1101 1101~~ \rightarrow ~~1111 1111 1111 1111~~ $1010 1110 1101 1101$; ~~Extrame de semn~~
 iar pentru un imediat pozitiv ~~Extrame de semn~~

$$I_{15} \quad I_6 \quad I_{31} \quad I_{15} \quad I_0$$

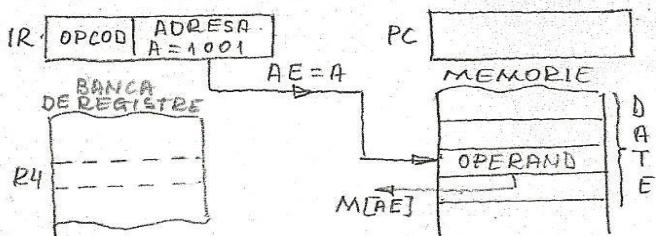
~~0010 1110 1101 1101~~ → ~~0000 0000 0000 0000~~, ~~0010 1110 1101 1101~~; $[I_{15}]^{16} \times [I_{15} - I_0]$

extensie de semn

2. ADRESAREA DIRECTĂ. La acest mod de adresare, adresa operandului (**adresa efectivă**, AE) se află în corpul instrucțiunii fie sub forma unui cuvânt adresă la memorie, fie sub forma unui cuvânt număr registru din banca de registre (adresă de registru), de unde referirea repectiv adresare directă la memorie sau adresare directă la registru. Adresarea directă la registru este utilizată când operandul căutat este deja înscris în registru, iar cea la memorie când se accesează date statice; prima variantă are avantajul unei execuții rapide și o utilizare mai eficientă a câmpului instrucțiunii.

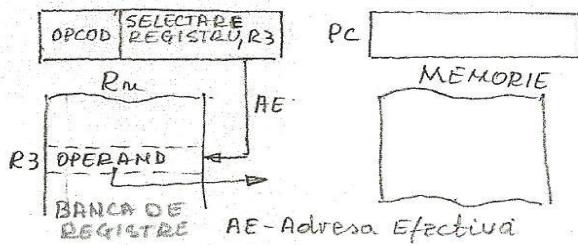
(2) Adresare directă la memorie

add \$R4, (1001) ; \$R4 + \$R4 + M[3001]



(c) Adresare directă la registre

add \$R4, \$R3 ; \$R4 ← \$R4 + \$R3



EXEMPLU. În ISA pentru MIPS adresarea directă la registre este cel mai frecvent mod de adresare; în corpul instrucțiunii sunt 3 adrese de registre.

add \$R1, \$R2, \$R3 ; \$R1 ← \$R2 + \$R3

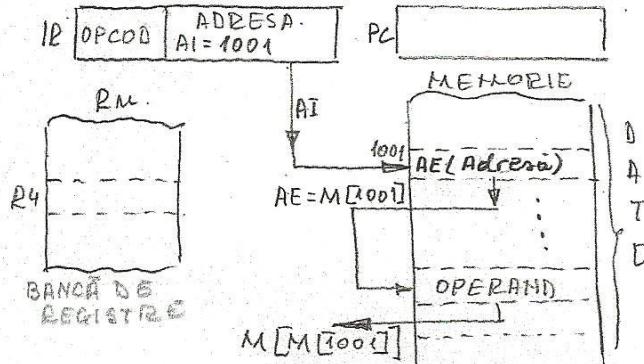
31	26 25	21 20	16 15	11 10	6 5	0
OPCODE	\$Rs	\$Rz	\$Rd	Shamt	funct	
	\$R2	\$R3	\$R1			

3. ADRESAREA INDIRECTĂ. Prin modul de adresare indirectă, în corpul instrucțiunii se specifică o adresă intermediară, AI; apoi, la punctul indicat de adresa intermediară se găsește înscrișă adresa efectivă AE, se extrage adresa efectivă (ca pointer) și se citește, de la locul indicat de AE, operandul necesar în instrucțiune. Există două variante de adresare indirectă:

- 3' adresarea indirectă prin memorie când adresa AI indică o locație de memorie al cărui conținut este AE, apoi se citește adresa efectivă și de la locația de adresă AE se extrage operandul, $M[AE]$;
 3'' adresarea indirectă prin registru când adresa AI indică un număr de registru al cărui conținut este AE, apoi folosind conținutul registrului ca pointer se citește locația de memorie de unde se extrage operandul $M[AE]$;

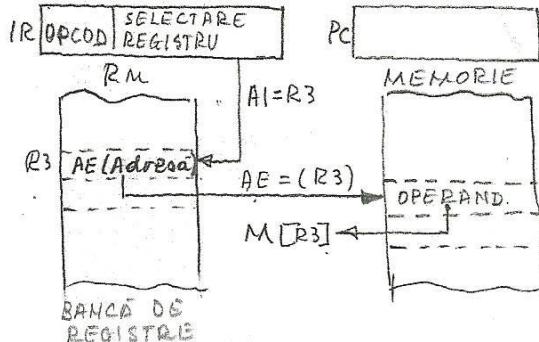
(3) Adresare indirectă prin memorie
(memory deferred).

$\text{add } \$R4, @\{1001\}; \$R4 \leftarrow \$R4 + M[M[1001]]$



(3') Adresare indirectă prin registru
(register deferred)

$\text{add } \$R4, \{R3\}; \$R4 \leftarrow \$R4 + M[R3]$



Adresarea indirectă este recomandată pentru cazurile când adresa efectivă, AE, de stocare a operandului nu se cunoaște în momentul compilării programului sau se modifică pe durata rulării programului. În ambele cazuri atunci când se cunoaște valoarea pentru AE aceasta se înscrie în punctul indicat de AI, ceea ce imprimă programului o mare flexibilitate. De exemplu, pentru accesul unor date successive se înscrie adresa efectivă, AE (prin incrementare sau decrementare) în punctul indicat de adresa intermediară, AI.

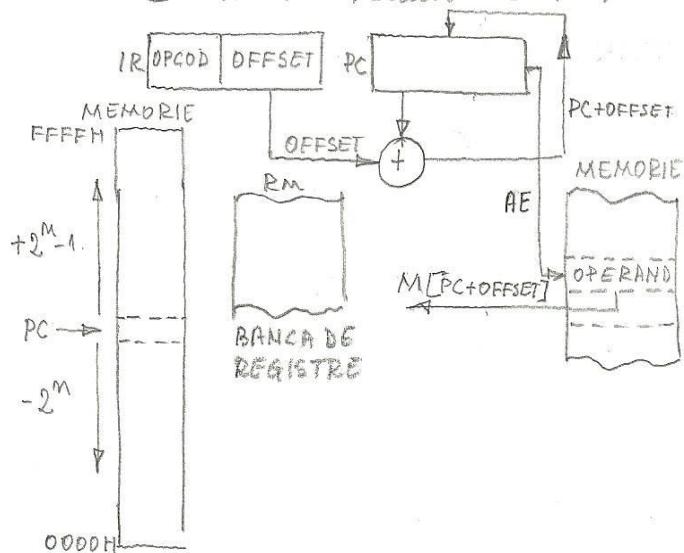
Adresarea indirectă poate fi extinsă prin mărirea numărului de indirectări (**indirectare multiplă**), dar pentru aceasta instrucțiunea necesită în corpul său un câmp (unu sau mai mulți biți) în care se specifică numărul de indirectări. Lanțul de indirectări se realizează: cu adresa AI₁, din corpul instrucțiunii, se accesează primul punct intermediu de unde se citește a două adresă intermedieră, AI₂, cu care se accesează al doilea punct intermediu de unde se citește a treia adresă intermedieră, AI₃, și. a. m. d., citirea din ultimul punct intermediu este AE, cu care se extrage apoi operandul necesar în instrucțiune.

Din analiza aflării adresei effective, de la adresarea indirectă, se deduce că aceasta se obține printr-un proces de calcul, acest proces de calcul este o simplă sau multiplă indirectare. Pentru modurile de adresare următoare, complexitatea calculului adresei efective crește necesitând uneori o unitate de calcul separată de cea care efectuează operația conform opcode-ului din instrucțiune, unitate de calcul dedicată doar pentru calculul necesar modului de adresare.

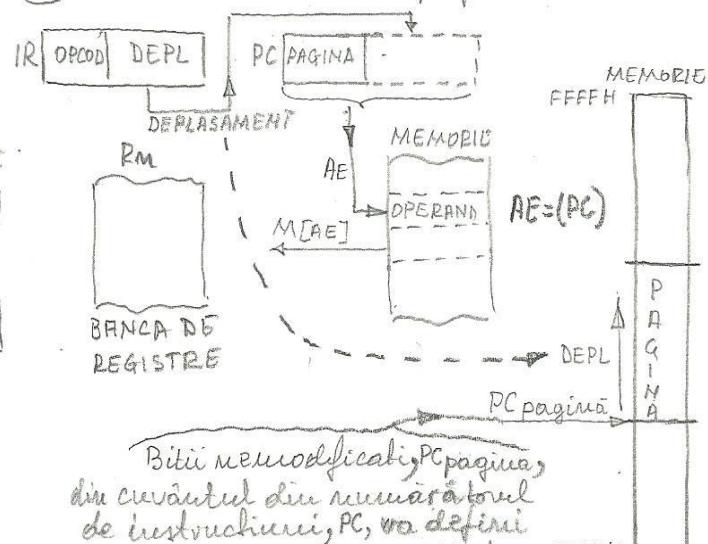
4. ADRESAREA RELATIVĂ. Adresa efectivă, AE, se obține prin sumarea la o adresă de referință (uzual adresa conținută în PC) a unei valori specificată ca un imediat în corpul instrucțiunii denumită, în general, ca DEPLASAMENT (când este un număr fără semn) sau denumită OFFSET (când este un număr cu semn exprimat în complement față de 2). Cu un offset de lungie (n+1) biți se poate acoperi o distanță (salt) de adrese față de adresa conținută în PC de $(2^n - 1)$ adrese înainte $[PC + (2^n - 1)]$ sau de 2^n înapoi $[PC - 2^n]$. Evident că implementarea modului de adresare relativă cu salt (4') necesită un sumator pentru calculul adresei effective.

O altă variantă de adresare relativă, care se poate implementa fără existența unui sumator pentru calculul adresei efective, este adresarea relativă în pagină (4''). La această variantă deplasamentul cu lungimea de (n+1) biți nu se sumează la cuvântul adresă din PC ci va substitui ultimii (n+1) biți din cuvântul adresă conținut în PC (segmentul de adrese care corespunde celor (n+1) biți îl referim ca pagină). Rezultă că adresa efectivă poate fi doar pe lungimea unei pagini (de adrese) față de PC, dar în schimb implementarea este simplă (stergerea ultimilor (n+1) biți din cuvântul existent în PC și realizarea unei operații SAU, $PC \cup \text{DEPLASAMENT}$).

(4) Adresare relativă cu salt



(4") Adresare relativă în pagină

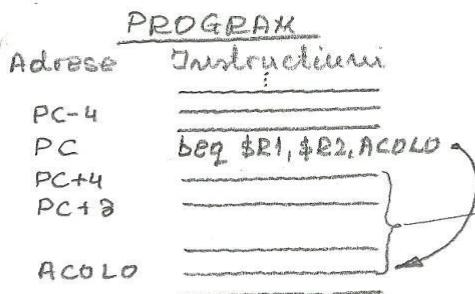


Bitii nemodificati, PC pagina, din cîrcajul din numărătorul de instrucțiuni, PC, va defini pagina din care se extrage operandul.

EXEMPLU (Adresare relativă cu salt implementat în instrucțiunea de salt condițional)

`beg $R1, $R2, ACOLO; if $R1 = $R2 atunci PC ← (ACOLO)
if $R1 ≠ $R2 atunci PC ← (PC+4)`

31	26 25	21 20	16 15	Imediat	1 0
OPCODE	\$Rs	\$Rt	I ₁₅		I ₀
	\$R1	\$R2			



Valoarea Imediat nu este adresa ACOLO, este diferență, ca număr de instrucțiuni, între instrucțiunea de la adresa ACOLO și instrucțiunea de la adresa (PC+4).
 $I_{\text{mediat}} = [(ACOLO) - (PC+4)] / 4$

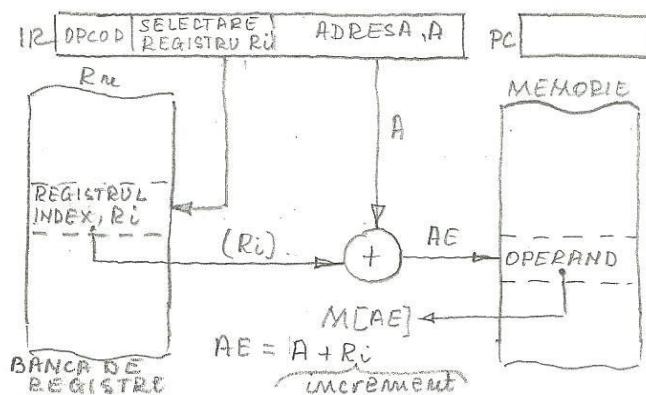
Valoarea Imediat înainte de a se însera cu adresa (PC+4) este extinsă de la un cuvânt de 16 biți la un cuvânt de 32 biți, se extinde bitul de semn (I_{15})¹⁶, apoi se deplasează cu două poziții la stânga (înmulțire cu 4) pentru a se trece de la numărul de instrucțiuni la numărul de adrese (o instrucțiune ocupă patru adrese la memorie organizată pe byte).

$$ACOLO := (PC+4) + [(I_{15})^{16} \times (I_{15} \div I_0) \times 4] \quad \text{adresa țintă de salt.}$$

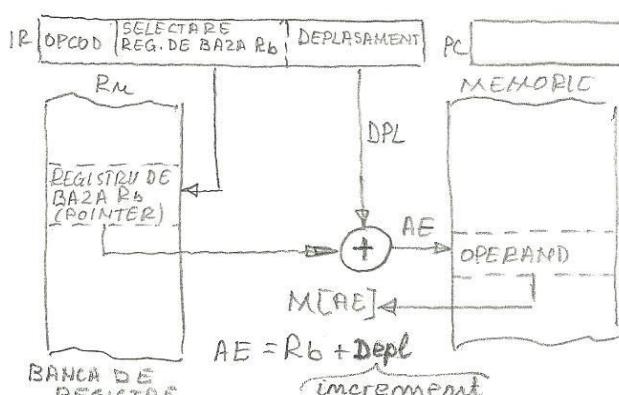
5. ADRESAREA INDEXATĂ sau ADRESAREA LA REGISTRUL DE BAZĂ

Prin acest mod de adresare, adresa efectivă, AE, se obține prin sumarea conținutului unui registru intern, Ri, selectat printr-un câmp al instrucțiunii, cu conținutul unui câmp DEPLASAMENT conținut tot în corpul instrucțiunii, $AE = Ri + Depl$; în această sumă unul din termeni este considerat ca o adresă de memorie iar celălalt ca un increment. Dacă Depl este considerat ca o valoare (fixă) de adresă iar Ri ca un increment modul de adresare (5') se referă ca adresare indexată (valoarea indexului este conținut în Ri). În cazul când Depl este considerat ca un increment iar valoarea registrului ca o adresă de bază, Rb, modul de adresare (5'') este referit ca adresare la (registrul de) bază, $AE = Rb + Depl$.

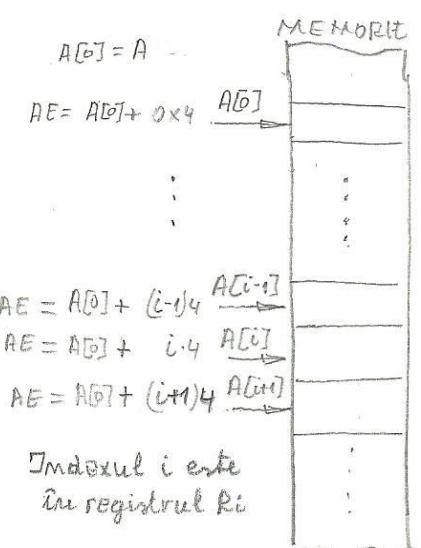
(5) Adresare indexată.



(5') Adresare la (registru de) bază



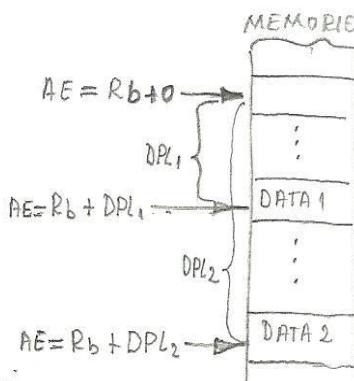
Adresarea indexată este indicată pentru accesarea datelor structurate de nivel superior (matrice, sir). Două componente successive ale iesinței vector coloană diferă, în general, cu o unitate în valoare indexului (îndiceleui) și sunt stocate în locații consecutive în memorie. O operatie asupra unui vector de date în formă unei bucle de program care să ia parcurgerea (iterație) procesorului al îlea elementul al vectorului. Accesarea se poate realiza printr-o adresare indexată, adică la fiecare parcurgere a buclei conținutul registruului index se însumează la adresa elementului $i=0$ al vectorului $A[0]$. Conținutul registruului index Ri este egal de fiecare data cu numărul iterării (eventual multiplicat cu 4). De la iterare la iterare, la procesarea vectorului, se incrementeză / decrementează conținutul registruului index Ri . La următoarele citiri sau înainte se autocrementeză / autodecrementeză.



Adresarea la bază, spre deosebire de cea indexată care este indicată pentru accesare date de tip vector, se utilizează pentru accesare de date grupate într-o anumită zonă. Accesarea fiecărei date se face prin intermediul unui deplasament conținut în corpul instrucțiunii la o adresă de bază inscrisă în registrul de bază. Valoarea deplasamentului depinde de numărul de biti alocati în corpul instrucțiunii.

EXEMPLU: `lw $R1,A($R2)`; $\$R1 \leftarrow M[A + \$R2]$, $\$R2$ -reg. de bază, A-deplasament
`SW $R1,A($R2)`; $M[A + \$R2] \leftarrow \$R1$, $\$R2$ -reg. de bază, A-deplasament

31	26 15	21 20	16 15	Imediat	Io
OPCODE	\$R2	\$R1	I ₁₅	Reg. de bază	A = Iimediat (deplasament)



6. ADRESARE LA STIVĂ.

STIVA este o structură de date care poate fi implementată într-un segment de memorie (referită ca stivă soft sau stivă în memorie) sau pe un grup de registre (referită stivă hard) pentru stocarea temporară a datelor/adreselor cu o funcționare de tip **LIFO** (Last-In-First-Out), iar funcționarea să se bazeze pe două operații de bază: push (înscrie în stivă), pop (citește din stivă), cărora le corespunde în setul de instrucțiuni, instrucțiunea Push și instrucțiunea Pop.

Adresarea stivei implementată în memorie se face printr-un pointer dedicat STACK POINTER, SP. Prin instrucțiunea **PUSH** se înscrie date în vârful stivei, ascunzând toate datele înschise deja în stivă sau se inițializează stiva dacă ea este goală. Prin instrucțiunea **POP** se extrage date existentă din vârful stivei, iar data de sub vârful stivei devine noua data din vârful stivei. Explicațiile următoare vor descrie microoperațiile corespunzătoare instrucțiunilor Push și Pop pentru o stivă construită în memorie. Stiva în memorie, ușor, crește în jos deoarece segmentul alocat pentru stivă este zona de adrese superioare din spațiul de adresare, baza stivei fiind adresa ce mai mare (adresa conținută în registrul frame pointer, FP (\$30)).

– Înscrierea în stivă (PUSH), stiva crește în jos

Push \$R ; SP ← (SP-1), M[SP] ← \$R, înscrie conținutul unui registru în vârful stivă
sau

Push Adresa ; SP ← (SP-1), M[SP] ← M[Adresa], înscrie conținutul unei locații în vârful stivă

– Citire din stivă (Pop), stiva crește în jos

Pop \$R ; \$R ← M[SP], SP ← (SP+1), înscrie vârful stivei într-un registru
sau

Pop Adresa ; M[Adresa] ← M[SP], SP ← (SP+1), înscrie vârful stivei într-o locație de memorie

În unele ISA, uneori, există în afară de incrementarea și decrementarea automată respectiv de la operațiile Pop și Push și instrucțiuni pentru modificarea conținutului pointerului de stivă, SP, de exemplu

Incr SP ; SP ← (SP+1), incrementeză adresa vârfului stivei

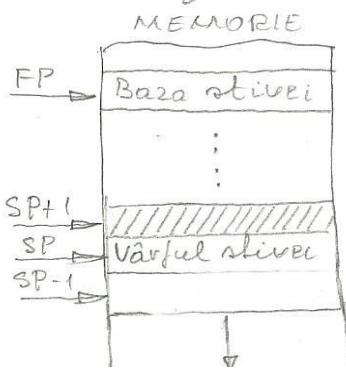
Dcr SP ; SP ← (SP-1), decrementeză adresa vârfului stivei

Load SP, DATA ; SP ← DATA, încarcă o adresă pentru vârful stivei

Pentru o operație unară (un singur operand) se extrage (POP) respectivul operand, care se află în vârful stivei, M[SP], se execută operația (de exemplu: complementare, shiftare) și rezultatul se înscrie (PUSH) tot în vârful stivei, M[SP].

Pentru operații binare (doi operanzi) se extrage (POP) primul operand din vârful stivei, M[SP], se extrage al doilea de la adresa de sub vârful stivei, M[SP+1], se execută operația, iar rezultatul se înscrie în locul celui de al doilea operand (adresa, SP ← SP+1, care acum este vârful stivei).

EXEMPLU: să se realizeze în stivă operația A = B+C (stiva este realizată în memorie și crește în jos)



$\cdot \text{PUSH adr } C ; SP \leftarrow SP-1, M[SP] \leftarrow M[\text{adr } C]$
 $\cdot \text{PUSH adr } B ; SP \leftarrow SP-1, M[SP] \leftarrow M[\text{adr } B]$
 Adol ; $M[SP+1] = M[SP] + M[SP+1], SP \leftarrow SP+1$
 $\cdot \text{POP Adr } A ; M[\text{Adr } A] \leftarrow M[SP], SP \leftarrow SP+1$

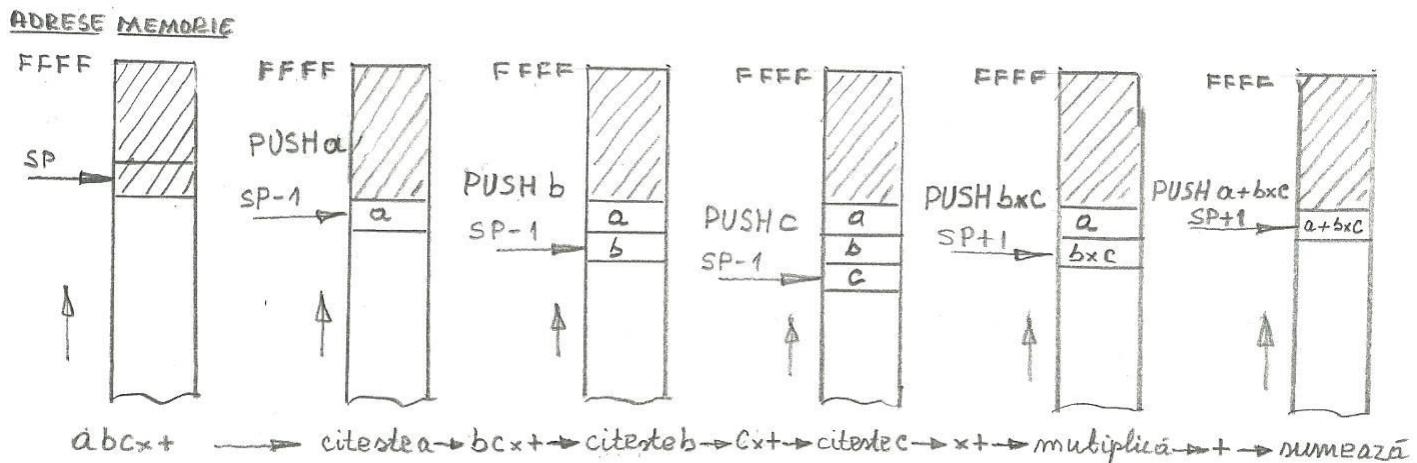
FP (Frame Pointer) conține adresa locației care este baza stivei.

La procesorul MIPS stiva se construiește în memorie (stivă soft) și este plastă în segmentul superior al spațiului de adresare; baza stivei se specifică cu pointerul FP (\$30), iar vârful stivei este indicat prin pointerul SP (\$29).

Pentru ușurință realizării operațiilor cu stiva, *notația normală (infix)* a expresiilor se transformă în *notația postfix* (*poloneză*). Pentru o expresie scrisă în notație postfix, utilizată pentru lucru cu stiva, se procedează în felul următor:

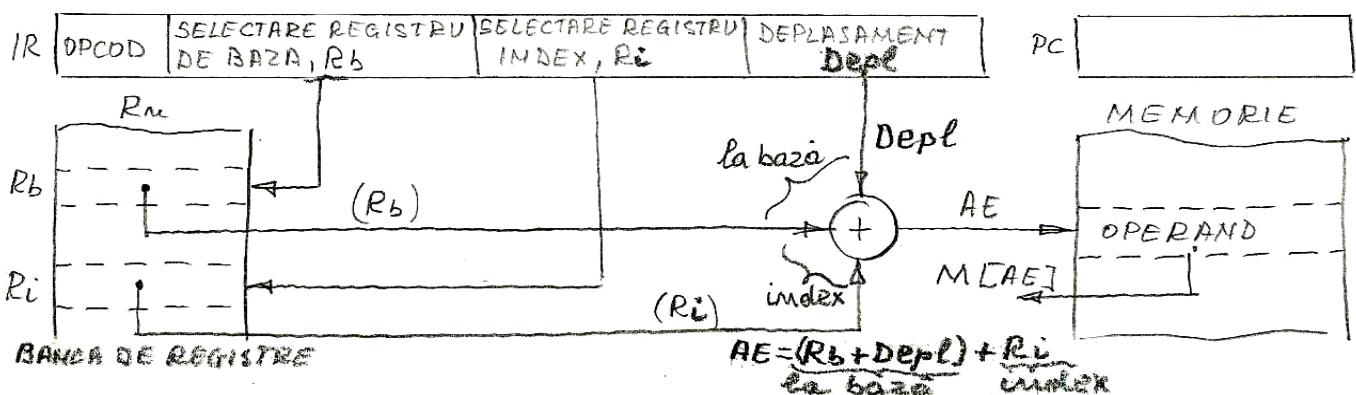
- când se întâlnește un operand, la parcurgerea de la stânga la dreapta a expresiei postfix, acel operand se introduce în vârful stivei (PUSH);
- când se întâlnește un operator, se extrag (POP) cei doi operanzi din vârful stivei iar rezultatul se înscrie (PUSH) în vârful stivei.

Exemplu. Expresia în infix $a + (b \times c)$ are în postfix forma $abcx+$, pentru care prin parcugere de la stânga la dreapta succesiunea de operații pe stivă sunt reprezentate în figura următoare

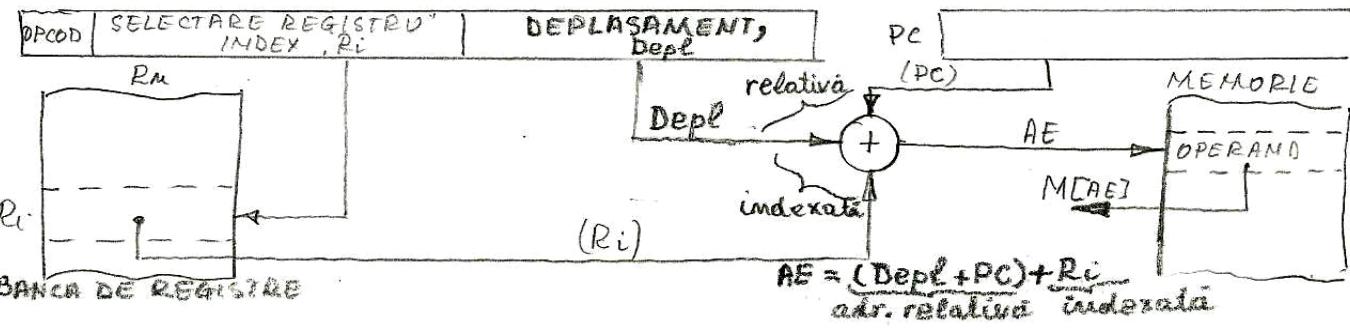


Cu modurile de adresare prezентate până acum se pot face combinații, astfel se pot obține numeroase variante de moduri de adresare. Arhitecturile CISC (x86) prezintă zeci de moduri de adresare, ceea ce nu este cazul la arhitecturile RISC care au doar câteva moduri de adresare. În continuare, se prezintă trei moduri de adresare care sunt combinații ale modurilor de adresare prezентate anterior.

7. ADRESAREA LA BAZĂ INDEXATĂ. Acest mod de adresare combină modul de adresare la (registru de bază (5')) cu modul de adresare indexat (5'). Evident că, pentru acest mod de adresare în corpul instrucțiunii trebuie să se specifice un registru de bază (Rb) și un deplasament (Depl) pentru a se obține deplasarea la bază (Rb + Depl), apoi mai este necesar un câmp în care să se specifică registrul index (Ri) pentru realizarea adresării indexate [(Rb + Depl) + Ri].

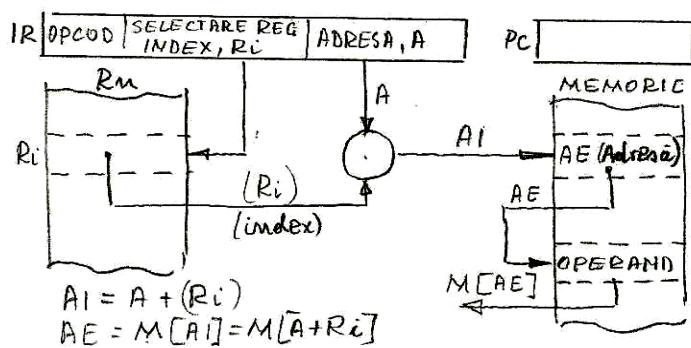


8. ADRESAREA RELATIVĂ INDEXATĂ. Prin acest mod de adresare se combină modul de adresare relativă, la PC (4'), cu modul de adresare indexat (5'). În corpul instrucțiunii trebuie să existe un subcâmp pentru deplasament, Depl, care sumat cu conținutul PC generează modul de adresare relativă cu salt (Depl + PC); de asemenea, trebuie să existe un subcâmp pentru specificarea registrului index, Ri, și apoi prin indexare se obține adresa efectivă, $AE = (Depl + PC) + Ri$.

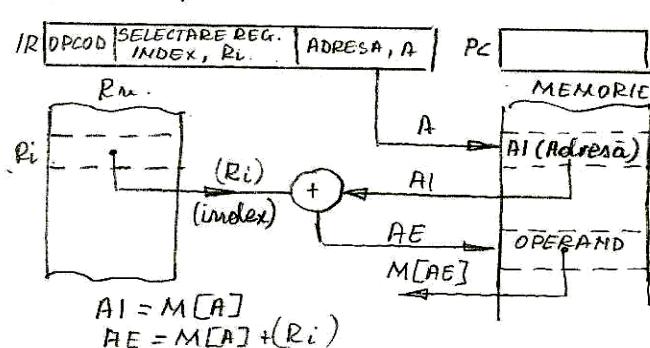


9. ADRESARE INDIRECTĂ CU INDEXARE. Acest mod de adresare este o combinare între adresarea indirectă (3) cu adresarea cu indexare (5'). Pot fi două variante pentru această combinare: 1. adresarea indirectă cu preindexare când indexarea se aplică (înainte) pentru calculul adresei indirecte, $AI = A + Ri$; 2. adresarea indirectă cu post indexare când indexarea se aplică după extragerea adresei indirecte, $AE = M[A] + Ri$.

(9') Cu preindexare



(9'') cu postindexare



Dintre modurile de adresare prezentate se poate observa că în construcția modului de adresare (aflarea valorii adresei efective, AE) se pot distinge trei operații (primitive de adresare):

1. Deplasarea (deplasament cu semn (offset) sau fără semn, Depl);
2. Indirectarea ($AE = M[AI]$)
3. Scalarea. Un registru, Ri (index) utilizat în calculul adresei efective poate fi scalat (înmulțit) cu o valoare de scalare fsc (valoare, în general, puteri ale lui 2, pentru ca prin index să poată fi accesate cuvinte care sunt multiplu de 1, 2, 4, 16, 32 bytes), rezultând valoarea de scalare, $Ri \cdot fsc$.

Existența a numeroase moduri de adresare la un μ P crează posibilitatea ca aceeași dată (operand) să poată fi accesată/găsită în diferite variante, altfel spus se pot parcurge mai multe trasee prin memorie pentru a ajunge la aceeași dată. O astfel de multitudine de trasee prin memorie pentru a ajunge la data căutată constituie o dificultate (prin care traseu să ajung la dată?) pentru scriitorul de compilator, dar pe de altă parte această varietate poate constitui soluții pentru diferite cazuri paticulare (trick-uri în limbajul de asamblare). De exemplu:

- un registru cu facilitatea de autoincrementare utilizat în modul de adresare indirectă (3'') poate simula funcționarea numărătorului de instrucțiuni, PC;
- instrucțiunile STORE și LOAD cu adresare indirectă printr-un registru index, Ri , (3'') cu posibilitatea de autoincrementare/autodecrementare sunt similare cu instrucțiunile Pop respectiv Push, iar registru index Ri emulează pointerul de stivă, SP.

La microprocesoarele CISC există un repertoriu foarte larg, peste 20, de moduri de adresare, pe când la cele bazate pe o arhitectură RISC, uzual, există 4-5 moduri de adresare. De exemplu, microprocesorul MIPS prezintă numai cinci moduri de adresare, prezentate în continuare.

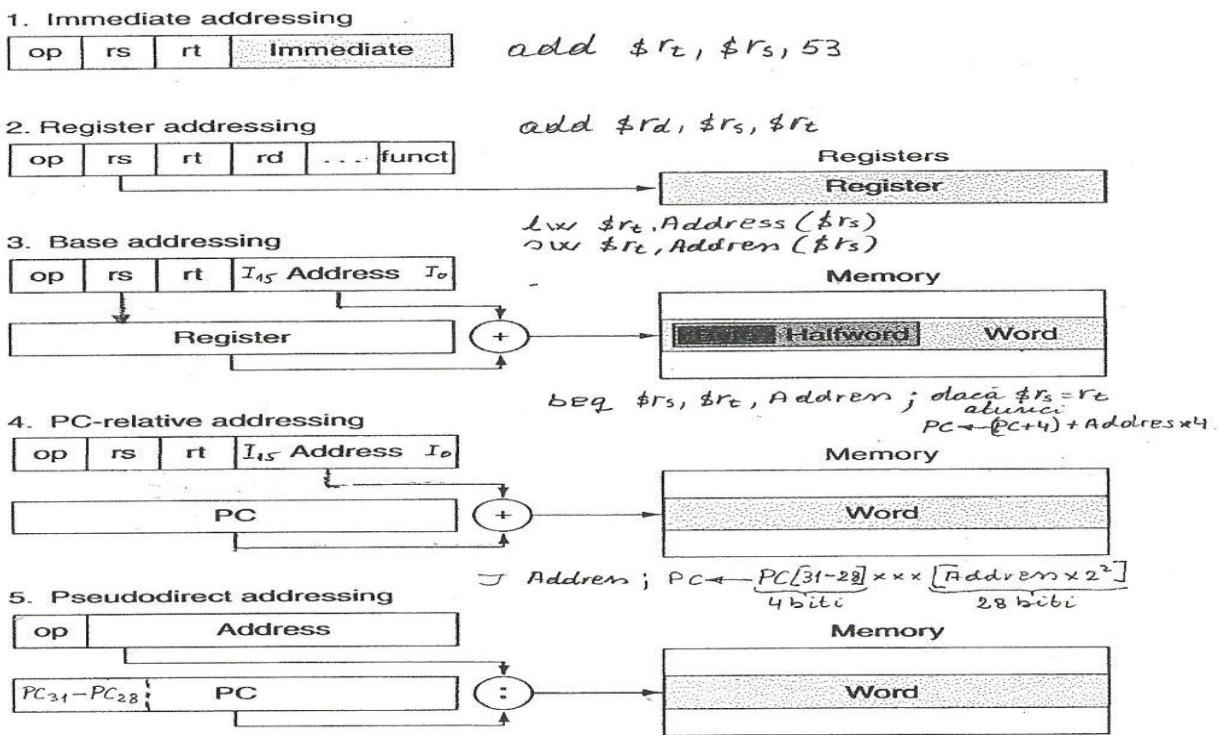


FIGURE 2.24 Illustration of the five MIPS addressing modes. The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, or words. For mode 1, the operand is 16 bits of the instruction itself. Modes 4 and 5 address instructions in memory, with mode 4 adding a 16-bit address shifted left 2 bits to the PC and mode 5 concatenating a 26-bit address shifted left 2 bits with the 4 upper bits of the PC.

Microunțiul MIPS fiind o mașină RISC prezintă foarte puține (cinci) și simple moduri de adresare.

2.5. TIPURI DE INSTRUCȚIUNI

Setul de instrucții pentru un µP se alege anticipând spectrul de aplicații, încât acestea să poată fi ușor programate. Un microprocesor specializat (coprocesoarele pentru diferite funcții, procesor de semnale, DSP) are un set de instrucții care sunt suport pentru anumite funcții dedicate și care este diferit de cel al unui procesor de utilitate generală.

Cuvântul instrucție cu lungimea de n biți ar putea codifica 2^n acțiuni ale µP, ceea ce, în general, ar depăși cu mult necesarul. Pentru mai multe motive (lejeritatea codificării, atribuirea denumirii, ușurința decodificării, limitări impuse lungimii, realizarea unei ortogonalități) instrucția se împarte în grupuri de biți (subcâmpuri) pe care se face o codificare locală. Pe aceste subcâmpuri se poate atribui o semnificație individuală fiecărui bit (codificare liniară) sau semnificația rezultă din codul care utilizează împreună toți biții din subcâmpul respectiv.

Primul subcâmp care trebuie să existe în oricare instrucție este cel care codifică acțiunea procesorului - codul operației, **OPCODE** (Operation CODE). Mnemonicul pentru acest cod al operației se alege printr-o abreviație a verbului acțiunii asupra procesorului : deplasează – move ; încarcă – load ; adună – add etc. Uneori mnemonicul operației exprimă și modul de adresare : addi – sumare cu un operand exprimat ca un imediat.

În funcție de efectul operației asupra datelor, instrucțiunile setului de instrucții pot fi clasificate în:

1. Instrucții de deplasare/ transfer a datelor
2. Instrucții de transformare a datelor
3. Instrucții de control al programului
4. Instrucții de control al procesorului

Acestă clasificare încercă să cuprindă toate instrucțiunile procesorului, dar există și combinații de mai multe operații în aceeași instrucție; în general aceste combinații generează instrucții care sunt referite ca

5. Instrucții de nivel înalt.

2.5.1 Instrucțiuni de deplasare/ transfer a datelor

Acet tip de instrucțiune realizează o deplasare de date între diferite componente ale sistemului (CPU, memorie, I/O) sau în interiorul acestora. De regulă, prin aceste instrucțiuni *nu se modifică conținutul sursei de unde se citește informația care se transferă*. Se vor prezenta, generic, doar unele (posibile) din astfel de instrucțiuni de deplasare/transfer.

Load	; $R_i \leftarrow M[\text{Adresă}]$.
Restore	; poate combina o succesiune de Load-uri dintr-o zonă de memorie, M, într-un bloc de registre, $R_n, R_n \leftarrow M$.
Store	; $M[\text{Adresă}] \leftarrow R_i$
Save	; poate combina o succesiune de Store într-o zonă de memorie dintr-un bloc de registre $M \leftarrow R_n$.
Push	; se înscrive în vârful stivei o valoare dintr-un registru sau dintr-o locație de memorie.
Pop	; se extrage valoare conținută în vârful stivei și se transferă într-un registru sau într-o locație de memorie.
Move	; deplasează cuvinte de la un registru la altul, $R_i \leftarrow R_j$, sau de la o locație de memorie la o altă locație de memorie $M[\text{Adresă}1] \leftarrow M[\text{Adresă}2]$. Mai corect, denumirea ar fi de Copy deoarece conținutul sursei nu se modifică.
Swap	; conținuturile a două registre sunt schimbată între ele $R_i \leftrightarrow R_j$ sau două locații de memorie $M[\text{Adresă}1] \leftrightarrow M[\text{Adresă}2]$.
Exchange	; schimbă conținuturile a mai multor registre între ele sau a mai multor locații de memorie.
Input port	; transferă conținutul unui port de adresă “port” într-un registru din banca de registre a procesorului, $R_i \leftarrow \text{“port”}$
Output port	; transferă conținutul unui registru din banca de registre într-un port de adresă “port”, “port” $\leftarrow R_i$.

Când sunt utilizate *coprocesoare de intrat/ieșire*, procesorul master poate avea un grup restrâns de intrare/ieșire, pentru că coprocesorul are el propriile instrucțiuni de transfer cu exteriorul. În general, instrucțiunile de intrare/ieșire ale procesorului master se pot limita la trei:

1. Start I/O ; asignează un program pentru coprocesorul I/O.
2. Test I/O ; permite procesorului master să testeze starea unui dispozitiv de I/O care este controlat de coprocesorul I/O.
3. Halt I/O ; permite masterului să opreasă o operație I/O în orice moment.

Drivele de lucru cu perifericele, incluse în general în sistemul de operare, sunt programe/rutine care realizează transferuri de date de la periferice la memorie și invers.

În general, operațiile de transfer de date sunt mari consumatoare de timp, în consecință se caută să se minimizeze durată și ca număr **“transferul de date costă mai mult decât procesarea”**

2.5.2. Instrucțiuni de transformare a datelor

Pot fi incluse în acest tip acele instrucțiuni care generează date noi în urma unor operații asupra operanzilor. Generic, unele din astfel de instrucțiuni (posibile) sunt prezentate în continuare.

- ARITMETICE: Add, Subtract, Multiply, Division, Increment, Decrement, Negate, BCD adjust, Modulo, Absolut value, Normalize etc.
- LOGICE: AND, OR, XOR, NOT, SHIFT/ROTATE, SET, RESET, CLEAR etc
 - AND poate fi privit ca un operator de mascare pentru resetare (într-un cuvânt poate înscrive biți în valoarea 0, $0 \cap a = 0$, sau să nu le modifice valoarea, $1 \cap a = a$).

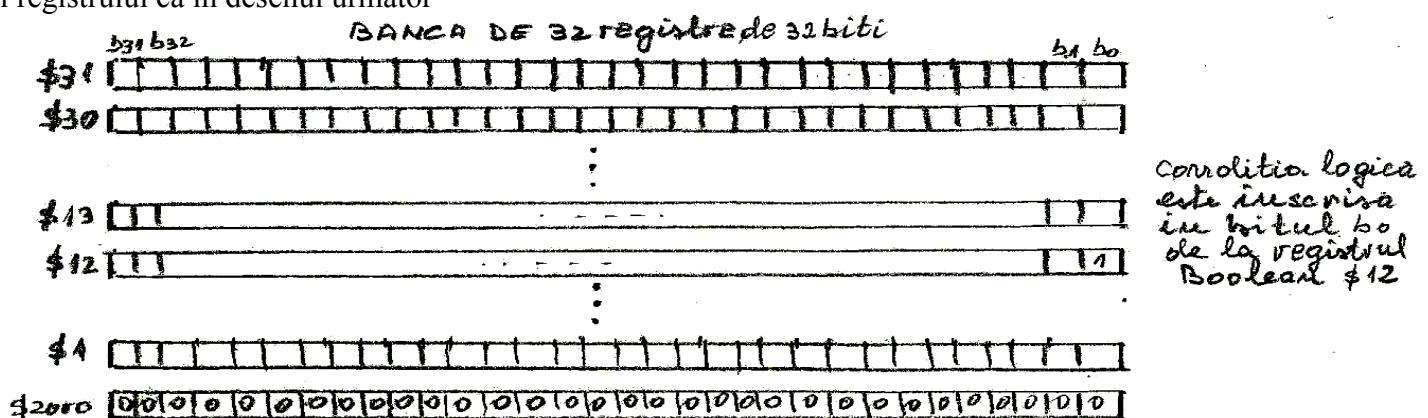
- OR poate fi privit ca un operator de mascare pentru setare (într-un cuvânt poate înscrie biți în valoarea 1, $1 \cup a = 1$, sau să nu le modifice valoarea, $0 \cup a = a$).
- XOR poate fi privit ca un operator de complementare (într-un cuvânt poate înscrie biți în valoarea complementată, $1 \oplus a = \bar{a}$, sau să nu le modifice valoarea, $0 \oplus a = a$).
- INSTRUCȚIUNI COMPARĂ/TESTEAZĂ (RELAȚIONALE): $=, \neq, <, \leq, >, \geq$

Rezultatul unei instrucțiuni relaționale este o valoare logică, adică se înscrie un bit/fanion în valoarea adevărat sau fals. Testând apoi în program valoarea acestui bit **se decide** traseul de urmat în contiunare. În funcție de locul în procesor unde este plasat fizic bitul cu valoarea de adevărat "1" sau fals "0" se pot distinge trei variante:

1. *Registrul codurilor de condiții (cc)*. Biții/fanioanele care se înscriu, cu valoarea zero sau unu, în urma unor operații relaționale sunt grupați toți sub forma unui registru denumit registrul codurilor de condiții, de exemplu la arhitectura IA-32, ca în reprezentarea următoare

<i>Overflow</i>	<i>carry</i>	<i>Greater Than</i>	<i>zero</i>	<i>Positiv</i>	<i>Negative</i>	<i>Equal</i>	<i>Less-Equal Than</i>	<i>cc</i>
OV	CY	GT	Z	P	N	E	LE	ET

2. *Registre Booleene* (de exemplu la procesorul Alpha 21264). Rezultatul unei operații relaționale se înscrie în unul din registrele din banca de registre, care este referit ca registru Boolean; se pot înscrie cu valoarea condiției toți biții registrului Boolean, dar ușual se înscrie doar bitul cel mai puțin semnificativ, LSB (Less Significant Bit) al registrului ca în desenul următor



3. *Registre predicative* (de exemplu la arhitectura IA-64). Un bloc de registre, ușual 32 sau 64 de registre cu lungimea de un bit, în care se înscriu rezultatele operațiilor relaționale.

BLOC DE REGISTRE PREDICATIVE

P63
P62
:
P23
P22
:
P1
P0

În urma efectuării unei instrucțiuni relaționale se înscriu cu valoarea 1/0 o sau mai multe registre predicative PJ/PJ din blocul de registre predicative în modul următor:
 condiție logică adevărată $\rightarrow P1 = 1, PJ = 0$
 condiție logică falsă $\rightarrow P1 = 0, PJ = 1$
 Exemplu: Instrucțiunea relatională este compge (comparare-great-equal)

P23, P22 compege \$5,10; dacă $R5 \geq 10$ atunci P23 $\leftarrow 1$ și P22 $\leftarrow 0$
 ; dacă $R5 < 10$ atunci P23 $\leftarrow 0$ și P22 $\leftarrow 1$

EXEMPLUL 2.3 Să se scrie un program în limbaj de asamblare pentru determinarea valorii logice a expresiei $(B > C) \text{ AND } (D = E)$.

1. pe o arhitectură de procesor cu registrul pentru codurile de condiții (cc)

```
CMP B, C      ; Compara B și C, înscrie bitul corespunzător din registrul codurilor de condiții cu valoarea
                 ; 1 dacă  $B > C$  sau cu valoarea 0 dacă  $B \leq C$ .
SGT $1         ; Bitul "Great Than" din registrul cc este citit și înscris în registrul $1
CMP E, D      ; Compara E cu D, se înscrie fanionul "Equal" =1 din registrul cc, altfel "Equal" =0
SEQ $2         ; Fanionul "Equal" din registrul cc este citit și înscris în registrul $2
AND $1, $1, $2 ; $1 ← ($1 AND $2) valoarea logică a expresiei se înscrie în registrul $1
```

2. pe o arhitectură de procesor cu registre Booleene

```
CMPGT $B1, B, C ; (Compare-Great-Than), dacă  $B > C$  bitul LSB (sau toți) din registrul boolean
                  ; $B1 este înscris în 1, iar pentru  $B \leq C$ , LSB ← 0.
CMPEQ $B2, E, D ; (Compare-Equal) dacă  $E = D$  atunci bitul LSB (sau toți) din registrul Boolean
                  ; $B2 este înscris în 1, iar pentru  $E \neq D$  LSB ← 0.
AND $B3, $B1, $B2 ; B3← ($B1 AND $B2), valoarea logică a expresiei se înscrie în registrul $B3
```

2.5.3 Instrucțiuni de control al programului

Traseul prin memorie în rularea unui program, atât timp cât $\text{PC} \leftarrow \text{PC} + 1$, parcurge succesiv adresele din memorie din segmentul de text. Modificarea acestui traseu se produce când în PC adresa următoare se obține prin $\text{PC} \leftarrow \text{Adresă}$, unde cuvântul Adresă se înscrie din exterior în PC, iar următoarea instrucțiune este extrasă din locația de adresă Adresă. Procesele care modifică traseul normal (succesiv, de parcurgere a adreselor în ordinea numerelor naturale) de extragere a instrucțiunilor sunt:

1. Execuția ramificațiilor de tipul: – instrucțiunile de salt necondiționat (Jump)
– instrucțiunile de salt condiționat (Branch)
2. Apelarea procedurilor (Call/Return)
3. Generarea de intreruperi (I) de tip : – externe
– interne (EIT*): – Exceptions (E)
– Faults
– Aborts
– Traps (T)
– etc

* EIT (Exceptions, Interrupts, Traps)

2.5.3.1. Instrucțiuni de ramificație

A. *Instrucțiuni de salt necondiționat (Jump)*. Deoarece o instrucțiune de salt necondiționat totdeauna se realizează, acest tip de instrucțiune se utilizează atunci când este necesară evitarea unui segment de program. Exemplu: Variante de instrucțiuni de salt necondiționat în setul de instrucțiuni MIPS.

1. Salt adresă.

Sintaxă: J Acolo Format:

	31	26	25		0
	OPCODE		b_{26}	Imediat	I_0

– Imediatul cu lungimea de 26 biți este deplasat la stânga cu două poziții (x4) și apoi concatenat cu cei patru biți mai semnificativi din PC (care are adresa $PC+4$)

Adresa Acolo = [PC₃₁–PC₂₈] xx (I₂₆–I₀)· 2²

2. Salt regisztr

Sintaxă: $J \quad \$r$

Format:

31	26	25	24	20	16	15	11	10	6	5	0
0	0	0	0	0	0	0	0	0	0	0	0

– Salt la adresa indicată în regisztrul \$r.

3. Jump-and-link

Sintaxă: $Jal \quad Subrutină$

Format:

31	26	25	0
OPCODE	immediat		

– Imediatul cu lungimea de 26 biți este deplasat la stânga cu două poziții (x4) și apoi concatenat cu cei patru biți mai semnificativi din PC (care are adresa PC+4).

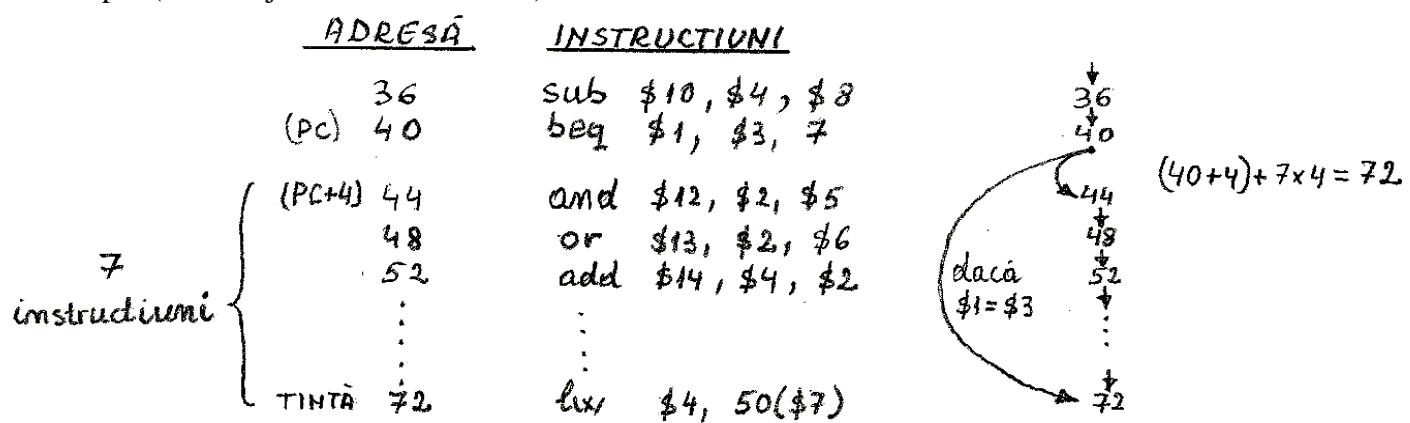
Adresa Subrutină = [PC₃₁–PC₂₈] xx (I₂₆–I₀)· 2²

– Conținutul (PC+4) din programul counter este introdus în regisztrul \$31. Păstrând adresa (PC+4), în regisztrul \$31, a instrucțiunii imediat următoare instrucțiunii Jal, după terminarea rulării (subrutinei) prin segmentul text din memorie, traseul se întoarce în programul principal la punctul imediat după instrucțiunea de ramificație/salt, adică la adresa (PC+4, păstrat în \$31).

B. Instrucțiuni de salt condiționat(branch)

- O instrucțiune de salt condiționat va executa salt la adresa calculată a instrucțiunii ȚINTĂ dacă este adevărată condiția de salt, iar dacă nu este adevărată condiția de salt va fi procesată instrucțiunea următoare din program.

Exemplu (în limbaj de asamblare MIPS)



- Acțiunea de salt condiționat în ISA poate fi realizată în două variante:

a) cu două instrucțiuni

- Instrucțiune TEST/COMPARE ; testează condiția de salt și înscrie valoarea logică obținută în regisztrul ; codurilor de condiții (cc) sau într-un regisztr Boolean
- Instrucțiune de salt ; testează condiția de salt înscrisă, dacă are valoarea adevărată atunci salt la ; adresa ȚINTĂ, altfel se continuă cu instr. următoare de adresa PC+1.

b) cu o singură instrucțiune; instrucțiunea de salt realizează atât testarea condiției de salt cât și saltul (dacă condiția de salt este adevărată), cum este implementat în MIPS.

Exemple de instrucțiuni de salt din MIPS

1. Branch equal, beq.

Sintaxă: $beq \quad \$1, \$2, ACOLo$ Format

OPCODE	\$1	\$2	ACOLo
--------	-----	-----	-------

Dacă $\$1 == \2 atunci $PC \leftarrow (PC+4) + ACOLo \times 2^2$, altfel $PC \leftarrow PC+4$

2. Branch not equal, bne.

Sintaxă $bne \quad \$1, \$2, DINCOLo$ Format

OPCODE	\$1	\$2	DINCOLo
--------	-----	-----	---------

Dacă $\$1 \neq \2 atunci $PC \leftarrow (PC+4) + DINCOLo \times 2^2$, altfel $PC \leftarrow PC+4$

3. Pentru următoarele relații de ordonare $<$, \leq , $>$, \geq există pseudoinstrucțiuni. Aceste pseudoinstrucțiuni se realizează cu instrucțiunea slt care testează condiția $a < b$ și apoi salt cu beq sau cu bne la instrucțiunea tintă (dacă este cazul). Următoarele două pseudoinstrucțiuni

blt \$a, \$b, TINTĂ ; salt la TINTĂ dacă mai mic decât ($$a < b)

bge \$a, \$b, TINTĂ ; salt la TINTĂ dacă mai mare sau egal ($$a \geq b)

sunt simulate în secțiunea 1.7.5 Exemplul 4.

Exercițiu. Să se scrie programul de simulare pentru următoarele două pseudoinstrucțiuni

- Branch – greater – then, bgt

bgt \$a, \$b, TINTĂ ; salt dacă mai mare s ($$a > b)

- Branch – less – equal, ble

ble \$a, \$b, TINTĂ ; dacă $$a \leq b atunci salt la instrucțiunea TINTĂ

- Instrucțiunile de salt condiționat produc mari scăderi de viteză în procesarea programelor când această procesare este de tip pipeline. Una din tehniciile moderne de a reduce scăderea de viteză la procesarea de tip pipeline constă în realizarea instrucțiunilor cu execuție condiționată (cu execuție predicativă, a nu se confunda cu instrucțiunea de salt condiționat!). Un program care conține instrucțiuni cu execuție condiționată va elimina instrucțiunile de salt necondiționat și va nulifica sau nu instrucțiunile de salt condiționat.

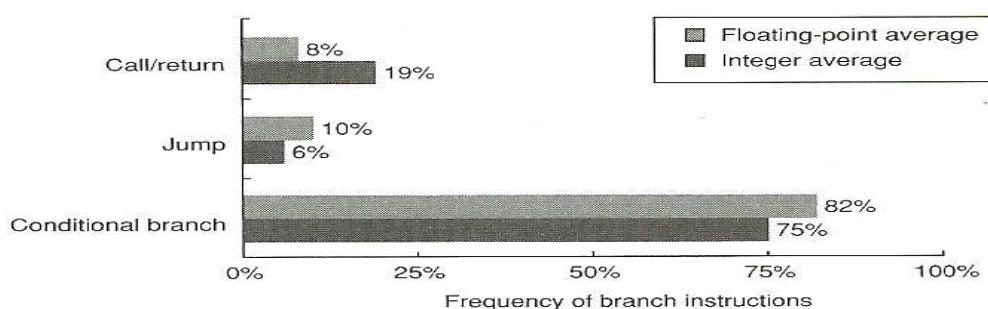


Figure 2.19 Breakdown of control flow instructions into three classes: calls or returns, jumps, and conditional branches. Conditional branches clearly dominate. Each type is counted in one of three bars. The programs and computer used to collect these statistics are the same as those in Figure 2.8.

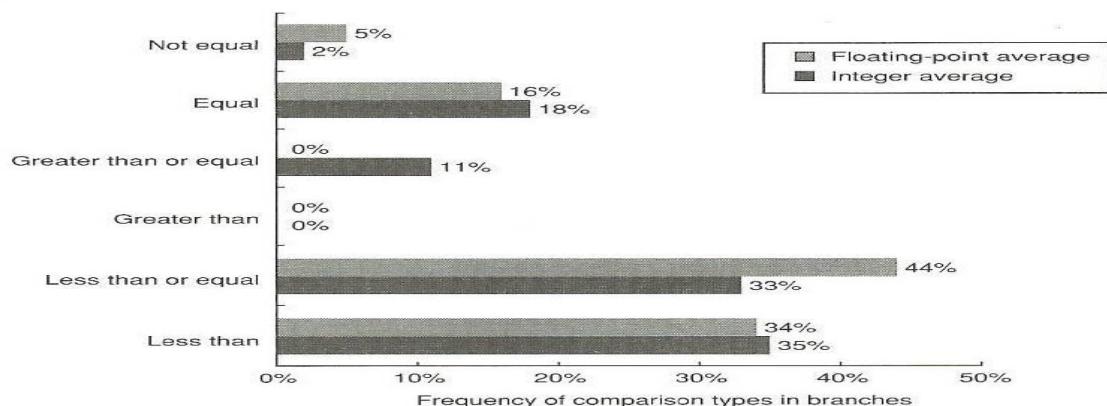


Figure 2.22 Frequency of different types of compares in conditional branches. Less than (or equal) branches dominate this combination of compiler and architecture. These measurements include both the integer and floating-point compares in branches. The programs and computer used to collect these statistics are the same as those in Figure 2.8.

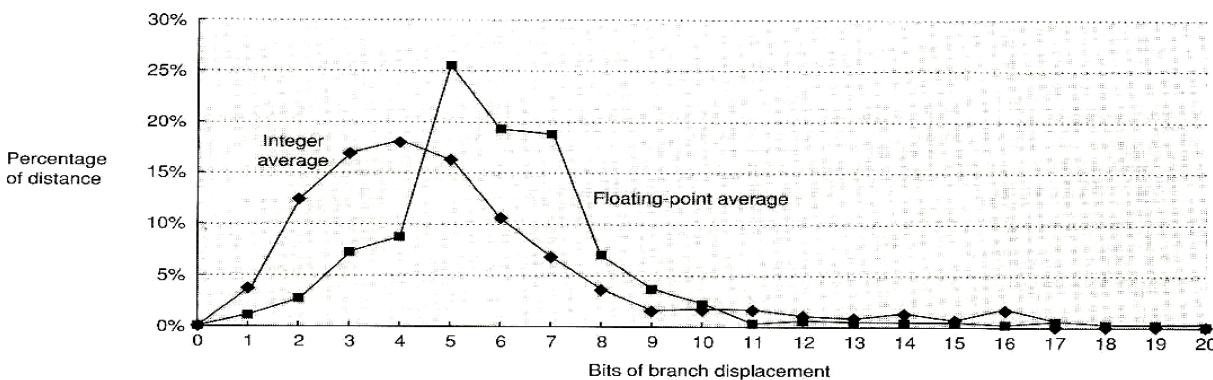


Figure 2.20 Branch distances in terms of number of instructions between the target and the branch instruction. The most frequent branches in the integer programs are to targets that can be encoded in 4–8 bits. This result tells us that short displacement fields often suffice for branches and that the designer can gain some encoding density by having a shorter instruction with a smaller branch displacement. These measurements were taken on a load-store computer (Alpha architecture) with all instructions aligned on word boundaries. An architecture that requires fewer instructions for the same program, such as a VAX, would have shorter branch distances. However, the number of bits needed for the displacement may increase if the computer has variable-length instructions to be aligned on any byte boundary. Figure 2.42 for Exercise 2.5 shows the accumulative distribution of these branch displacement data. The programs and computer used to collect these statistics are the same as those in Figure 2.8.

Din Figura precedentă rezultă că pentru majoritatea salturilor condiționate lungimea distanței de salt, față de adresa instrucțiunii de salt condiționat, se poate exprima cu un imediat de cel mult 8–10 biți (această particularitate rezultă din principiul localizării (în spațiu a) programelor). Rezultă că un imediat cu lungimea de 16 biți în corpul instrucțiunii este suficient pentru realizarea saltului. Mai mult, acest imediat în corpul unei instrucțiuni MIPS, care exprimă numărul de instrucțiuni peste care se efectuează saltul, este multiplicat cu 2^2 (deplasat spre stânga cu două poziții) și se obține numărul de adrese peste care se efectuează saltul în modul următor

beq, bne	31 OPCODE	26-25 \$r ₁	24-20 \$r ₂	16-15 I _{15, I₁₄}	Immediat	0 I ₀
-------------	--------------	---------------------------	---------------------------	--	----------	---------------------

beq \$r₁, \$r₂, TINTA; dacă (\$r₁ == \$r₂) atunci PC ← TINTA = (PC+4) + [(I₁₅)¹⁶ × (I₁₅ - I₀) × 2²]
bne \$r₁, \$r₂, TINTA, dacă (\$r₁ != \$r₂) atunci PC ← TINTA = (PC+4) + [(I₁₅)¹⁶ × (I₁₅ - I₀) × 2²]

C. Instrucțiuni cu execuție condiționată

– Formatul instrucțiunii cu execuție condiționată

Sintaxă <PI> INSTRUCȚIUNE; Format

OPCODE	GPR1	GPR2	GPR3	NRP
--------	------	------	------	-----

 Bloc de registre predicative (de 1 bit)

în care:

- PI - valoarea predicatorului înscrișă în registrul predicatoriu RP, registrul specificat în subcărpușul NRP al instrucțiunii.
- GPR1, GPR2, GPR3 - registrele de utilizare generală din blocul de registre ale UP, folosite de instrucțiunea de execuție condiționată.
- NRP - subcărpușul din instrucțiunea cu execuție condiționată în care se specifică numărul de registru predicatoriu, din blocul de registre predicative, registru în care este înscrișă valoarea predicatorului instrucțiunii, <PI>.

Bloc de registre predicative (de 1 bit)
 :
 RP_{i+2} P_{i+2}
 RP_{i+1} P_{i+1}
 RP_i P_i
 :

Instrucțiunea este adusă din memorie, în ciclul de FETCH, decodificată și executată în ciclul EXECUȚIE conform OPCODE, dar rezultatul obținut se înscrie la destinație (registru, memorie) NUMAI dacă valoarea predicatorului, <PI>, atașat instrucțiunii, înscriș în registrul predicatorie (specificat în câmpul N RP) are valoarea 1,

iar dacă are valoarea 0, rezultatul produs prin execuția instrucțiunii nu se înscrie la destinație (deci instrucțiunea consumă timp dar nu produce rezultat, ceea ce este echivalent cu rularea instrucțiunii NOP - No OPeration).

- Formatul instrucțiunii relaționale (de (com)parație care înscrie valorile (predicalele) în registrele predicative)

Sintaxă $\langle PJ, PK, \text{com}(relație) \rangle$: Format $|OPCODE|GPR1|GPR2|GPR3|NRP_J|NRP_K|$

în care:

PJ, PK – predicalele înschise în registrele predicative în urma testării *relație*;
 NRP_J, NRP_K – numerele registrelor predicative în care instrucțiunea predicativă înscrie rezultatul testării *relație*.

În urma execuției instrucțiunii relaționale, dacă valoarea relației testate este adevărată se înscrie în cele două registre predicative NRP_J, NRP_K respective valorile $PJ = 1, PK = 0$, iar dacă valoarea relației testate este falsă se înscrie în registrele predicative $PJ = 0, PK = 1$.

- Formatul instrucțiunii relaționale cu execuție condiționată (și instrucțiunea relațională poate fi executată condiționat!)

Sintaxă: $\langle PI \rangle \langle PJ, PK, \text{com}(relație) \rangle$: Format $|OPCODE|GPR1|GPR2|GPR3|NRP_J|NRP_K|NPR|$

Pentru: $\langle PI \rangle = 1$ instrucțiunea este o instrucțiune relațională normală, înscrie registrele predicative NRP_J, NRP_K ;

$\langle PI \rangle = 0$ instrucțiunea este o instrucțiune relațională dar nu înscrie valorile în registrele predicative NRP_J, NRP_K ; este echivalentă cu o instrucțiune NOP.

Modul de operare al instrucțiunii relaționale cu execuție condiționată este sintetizat în tabelul următor:

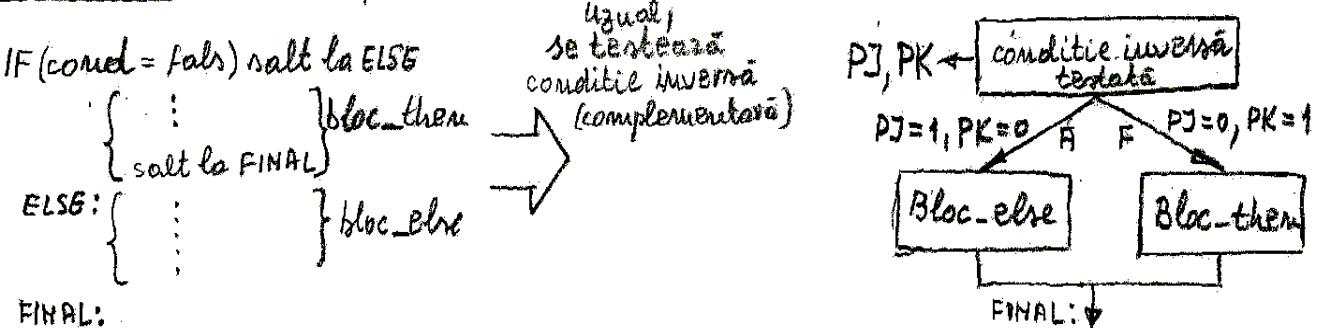
Valoarea predicatului PI	Valoarea calculată a relației	Valoile care se înscriu în registrele predicative		Instrucțiunea se execută și se înscriu cele două registre predicative (NRP_J, NRP_K)
		PJ	PK	
0	ADEVĂRAT	0	0	Instrucțiunea se execută dar nu înscrie registrele predicative (NOP)
0	FALS	0	0	Instrucțiunea se execută și se înscriu cele două registre predicative (NRP_J, NRP_K)
1	ADEVĂRAT	1	0	Instrucțiunea se execută și se înscriu cele două registre predicative (NRP_J, NRP_K)
1	FALS	0	1	Instrucțiunea se execută și se înscriu cele două registre predicative (NRP_J, NRP_K)

Pentru un ISA cu execuție condiționată μP trebuie să aibă un **mecanism de nulificare**, de neînscrire a rezultatului la sfârșitul etapei de execuție (realizare NOP).

Cu ajutorul instrucțiunilor cu execuție condiționată/predicative salturile necondiționate (jump) sunt eliminate din program iar instrucțiunile de salt condiționat/branch sunt transformate în instrucțiuni care se nulifică sau nu se nulifică. Unele μP actuale au toate instrucțiunile din ISA cu execuție condiționată (IA-64).

EXEMPLUL 2.4 Realizarea primitivei **IF conditie THEN bloc_then ELSE bloc_else** pe un ISA cu instrucțiuni cu execuție condiționată.

EXEMPLUL 1 IF condiție THEN bloc_then ELSE bloc_else



IF ($\$1 == \2) THEN $\$3 = \$4 + \$5$ ELSE $\$6 = \$4 - \$5$

DUPA COMPILEARE:

CU INSTRUCȚIUNI NORMALE

```
bne $1,$2,ELSE ; salt pe cond inversă
add $3,$4,$5 ; bloc_then
J FINAL ; salt necondiționat
ELSE: sub $6,$4,$5 ; bloc_else
FINAL;
```

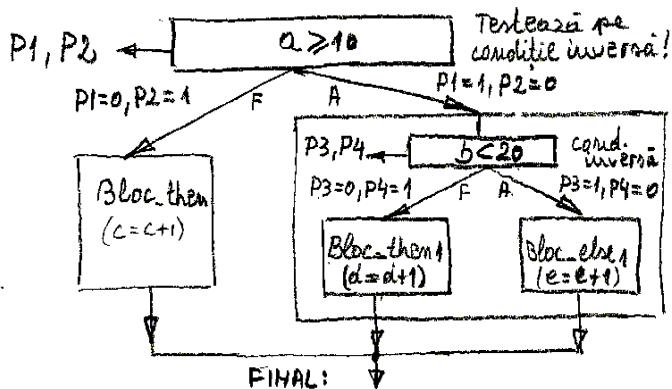
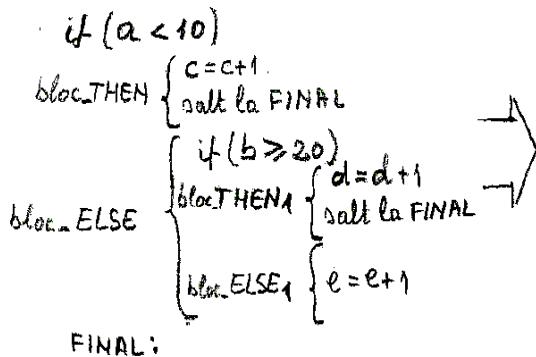
CU INST CU EXECUȚIE CONDIȚIONATĂ

```
(1) P1, P2 compara $1, $2 ; com daca $1 ≠ $2
(2) < P2 > add $3, $4, $5 ; pt P2=1, normal
(3) < P1 > sub $3, $4, $5 ; pt P1=1, normal
                                P1=0, multiplicare.
```

Dacă $\$1 \neq \2 se inscrie $P1=1, P2=0$, iar pentru $\$1 == \2 se inscrie $P1=0, P2=1$. Pentru:
 setul $P1=0, P2=1$ se execută instrucția (2) (adică bloc_then) iar instrucția (3) se multiplică (bloc_else)
 setul $P1=1, P2=0$ se execută instrucția (3) (adică bloc_else) iar instrucția (2) se multiplică (bloc_then)

EXEMPLUL 2.5 Realizarea primitivei

IF (a<10) THEN (c=c+1) ELSE [IF (b ≥ 20) THEN (d = d+1) ELSE(e = e+1)]
 pe un ISA cu instrucțiuni cu execuție condiționată



DUPA COMPILEARE:

CU INSTRUCȚIUNI NORMALE

```

bge $a,10,L1 ; $a > 10?
add $c,$c,1 ; bloc_then
J FINAL ;
L1: blt $b,20,L2 ; $b < 20?
add $d,$d,1 ; bloc_then1
J FINAL ;
L2: add $e,$e,1 ; bloc_else1
FINAL;
  
```

- În programul scris cu instrucțiuni nepredicative traseele corespunzătoare valoilor condițiilor testate se realizează cu ajutorul instrucțiunilor de salt conditional și neconditional. În programul cu instrucțiuni cu execuție condiționată se execută cele două trasee, elor se înscriv rezultatul doar pentru traseul care are valoarea adevărată pentru condiția testată, instrucțiunile de pe celălalt traseu se nulifică.

- Utilizând instrucțiuni predicative se procesează în paralel în pipeline atât ramura ELSE cât și ramura THEN, cănd condiția este verificată adică se înscriz $P1=1, P2=0$ sau $P1=0, P2=1$ se nulifică respectiv ramura THEN sau ramura ELSE.
- Utilizând instrucțiuni cu salturi condiționate, conform predicii, se alege în pipe fie ramura ELSE fie ramura THEN dar dacă predicia e fapt greșită atunci pipe-ul trebuie golit și execuția ramură se oprește.

2.5.3.3. Instrucțiuni de lucru cu subrute

O modalitate de scriere a programelor, care duce totdeauna la economie, sistematizare și depanare ușoară a programelor, este cea de modularizare, adică programul este divizat în componente/module independente. Aceste module sunt compilate independent, iar jonctionarea lor este realizată de către programul LINKER și plasarea programului linkat, în memoria calculatorului, se face de către programul INCĂRCĂTOR. O serie de module care implementează procesări uzuale (funcții uzuale, drivere I/O etc) nu se mai elaborează de programator deoarece acestea există în bibliotecă de unde se apelează, ca module independente, de către programul principal. Aceste programe de bibliotecă sunt referite ca subrute sau proceduri (methods în Java). Pentru lucrul cu subrute setul de instrucțiuni trebuie să implementeze următoarele două primitive:

1. SALT LA SUBRUTINĂ sub forma instrucțiunii

Call Sub ; $PC \leftarrow (\text{Sub})$, $M[SP] \leftarrow (PC+k)$

iar particularizată pentru MIPS este instrucțiunea jal (jump-and-link)
 jal Sub ; PC ← (Sub), \$31 ← (PC+4)

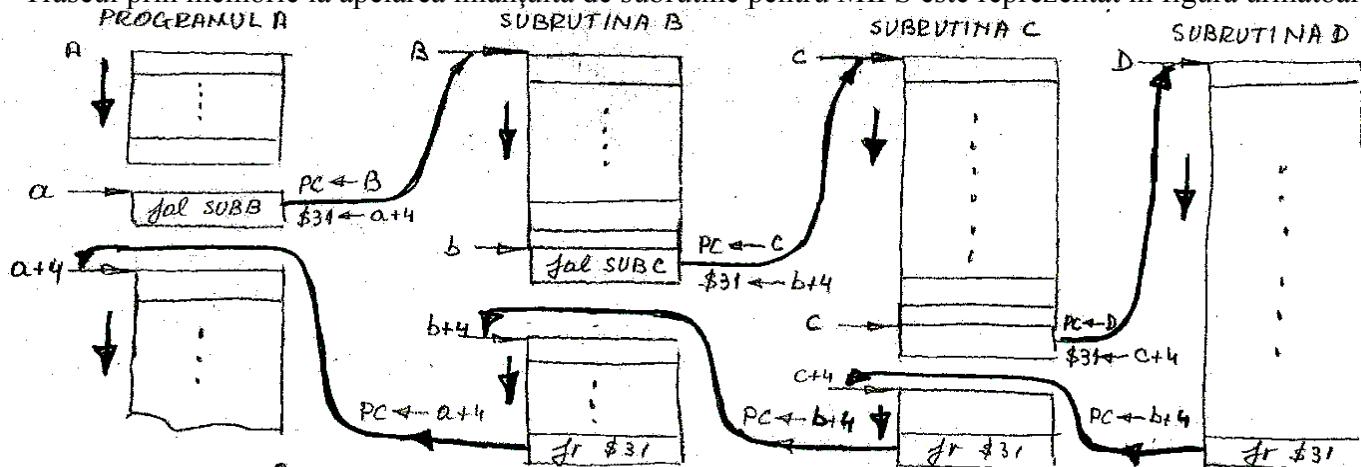
La apelarea subruteinei Sub (prin instrucțiunea Call sau jal la MIPS) adresa instrucțiunii următoare (PC+k sau PC+4 la MIPS) este salvată în stivă sau în registrul \$31 (la MIPS), în scopul reîntoarcerii în acel punct din programul principal după rularea subruteinei.

2. REVENIRE DIN SUBRUTINĂ sub forma instrucțiunii Return ; PC ← M[SP]

iar particularizată pentru MIPS este instrucțiunea jr (jump registru)
 jr \$31 ; PC ← \$31

Instrucțiunea Return sau jr \$31 este ultima din subrutină și încarcă PC, din stivă sau din registrul \$31, cu adresa de revenire în programul apelant. Uzual în cazul subruteinelor ce servesc cererei de intrerupere prima instrucțiune din subrutină este DI (Disable Interrupt), această instrucțiune blochează eventuala servirii unei alte intreruperi ivite pe durata rulării subruteinei curente, iar penultima instrucțiune din subrutină este EI (Enable Interrupt), care redă posibilitatea de întreupere. De regulă, instrucțiunile EI și Return sunt jonctionate într-o singură instrucțiune, care este neîntreruptibilă (atomică), adică μP nu poate fi întrerupt pe durata rulării acestei instrucțiuni, care este ultima din subrutină, decât numai după înscrierea adresei de reîntoarcere în programul apelant.

Traseul prin memorie la apelarea înlănțuită de subruteine pentru MIPS este reprezentat în figura următoare



ATENȚIE . De fiecare dată când se apelează de către o subrutină o altă subrutină, care la rândul său apelează o altă subrutină (subruteine înlănțuite) adresa de reîntoarcere a subruteinei apelante, din registrul \$31, trebuie salvată în stivă, iar la reîntoarcerea subruteinei apelante trebuie să refacă (din stivă) în registrul \$31 adresa de reîntoarcere a subruteinei apelante.

La apelarea unei subruteine programul apelant transmite subruteinei un număr de parametri, iar la reîntoarcere subruteina transmite programului apelant valorile calculate. Transmiterea parametrilor la subrutină se poate realiza prin una din următoarele metode:

1. prin valoare (se transmite valoarea parametrului într-un registru/locație, alocată implicit);
2. prin adresă (se transmite adresa registratorului/locației unde este stocată valoarea parametrului);
3. prin nume (se transmite relația de calcul a parametrului).

Pentru MIPS conform convenției de alocare a registratorilor (vezi secțiunea 1.7.5) transmiterea parametrilor se face prin registratorle \$a0, \$a1, \$a2, \$a3, iar dacă sunt mai mulți de patru parametri atunci cei de la patru în sus se transmit prin stivă. Reîntoarcerea în programul principal a valorilor calculate de subrutină se face prin registratorle \$v0 și \$v1.

Instrucțiunea Call Sub poate fi privită tot ca o instrucțiune de salt, dar o instrucțiune de salt care își salvează adresa de reîntoarcere din programul apelant.

SECVENTIEREA OPERATIILOR PENTRU CALL-RETURN

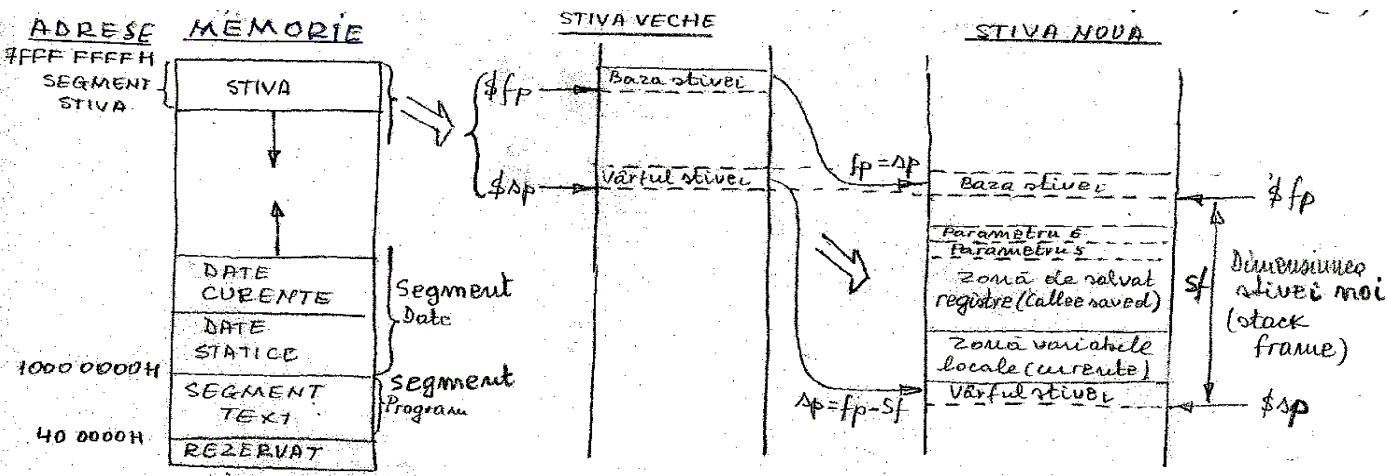
Înainte de apelarea unei subrutine (programul apelat), programul apelant trebuie să pregătească parametrii de transmis și să salveze toate informațiile de proces (registre) care îi sunt necesare nemonificate la reîntoarcere din subrutină, dar care pot fi nedorit modificate de apelat. La rândul său, apelatul nu trebuie să modifice acele valori necesare apelantului ca să reînceapă rularea corectă după reîntoarcerea din subrutină și totodată să reîntoarcă în programul apelant valorile calculate. Rezultă că pentru procesul CALL-RETURN trebuie fixate, prin convenție, cum să se desfășoare următoarele trei etape:

1. ce face apelantul înainte de apelare;
2. ce face apelatul imediat după apelare;
3. ce face apelatul imediat înainte de reîntoarcere înspre programul apelant.

Se vor analiza aceste etape pe MIPS, ținând cont de convenția de alocare a registrelor (vezi secțiunea 1.7.5).

Procesorul MIPS nu are mecanism (hard) automat pentru realizarea lucrului cu stiva (stivă soft), în consecință acest mecanism trebuie realizat în software de către programator. În acest scop se utilizează registrul \$29 ca pointer de stivă (SP, adresa liberă din vârful stivei), registrul \$30 ca frame pointer (baza stivei), iar registrul \$31 ca pointer al adresei de reîntoarcere în programul apelant.

1. (Imediat) înainte de apelare, programul apelant efectuează:
 - a) Transferul parametrilor. Parametrii de transmis apelatului sunt înscrise în registrele \$a0 – \$a3, iar pentru parametrii care depășesc numărul de patru aceștia se vor transmite prin zona de stivă pe care și-o va construi apelatul;
 - b) Salvarea registrelor pe care (se așteaptă) să fie utilizate de apelat fără ca apelatul să le salveze în prealabil (caller-saved – registers sunt: \$t0 – \$t9, \$a0 – \$a3);
 - c) Executarea apelării, cu instrucțiunea jal Sub; PC ← Sub, \$ra ← PC+4.
2. (Imediat) la începutul programului apelat:
 - a) Fixarea dimensiunii stivei, sf (stack frame), utilizată de programul apelat. Baza noii stive se fixează la adresa indicată de vârful stivei (vechi) de la programul apelant, iar vârful noii stive va fi cu un număr sf de adrese mai jos (subi \$sp, \$sp, sf ; \$sp ← \$sp – sf), stiva crește în jos;
 - b) Salvarea registrelor pe care apelatul (se așteaptă) să le utilizeze, dar a căror valori trebuie păstrate intacte pentru apelant, aceste sunt din: \$s0 – \$s7, \$fp, \$ra. Registrul \$fp (baza stivei vechi, a programului apelant) este salvat în noua stivă de către fiecare apelat, pentru că fiecare apelat își alocă propriu său început de stivă (baza stivei noi). Registrul \$ra se salvează doar atunci când apelatul apelează la rândul său un alt apelat (subrutină);
 - c) Stabilirea bazei stivei noi, fp (frame pointer). Frame pointerul, fp, pentru noua stivă va avea valoarea egală cu adresa indicată de stack pointerului sp de la stiva veche și se calculează cu instrucțiunea addi \$fp, \$sp, sf ; \$fp ← \$sp + sf, (se consideră că \$sp indică prima locație liberă din noua stivă).
3. Rularea programului apelat.
4. (Imediat) înainte de reîntoarcerea din programul apelat:
 - a) Plasarea valorilor calculate în registrele \$v0, \$v1;
 - b) Se refac (dacă sunt) registrele salvate de apelat din: \$s0 – \$s7 (Callee-saved-registers);
 - c) Se restabilește vârful stivei vechi, addi \$sp, \$sp, sf ; \$sp ← \$sp + sf. Aceste restabiliri sunt în ordine inversă ("în oglindă") în raport cu salvările de la începutul programului apelat (de la punctul 2);
 - d) Salt la adresa de revenire din programul apelant, jr \$ra ; PC ← (\$ra).



EXEMPLUL 2.6. Modul de apelare înlănțuită a subruteinilor la MIPS

PROGRAM A

Adrese lstr.

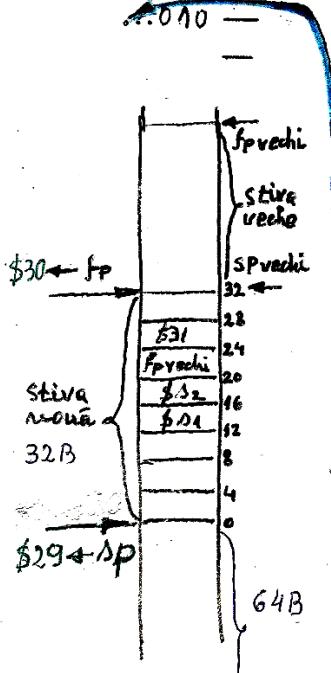
...000 —

004 —

008 —

...00C jal B

...010 —

SUBRUTINA B

Adrese lstr.

...200 subi \$29, \$29, 32 ; la nouăriu de 32 bytes refixați sp
 204 sw \$31, 24(\$29) ; salvează adresa de reinicioare
 208 sw \$30, 20(\$29) ; salvează vechiul pointer fp
 20C addi \$30, \$29, 32 ; fixează nouul fp (= vechiul sp)
 210 sw \$18, 16(\$29) ; salvează registrul \$A2 } Callee
 214 sw \$19, 12(\$29) ; salvează registrul \$A3 } save

PC ← (...200)
 ...010 →
 PC ← (...010)

...252 jal C
 ...256 —
 PC ← (...256)

SUBRUTINA C

Adrese lstr

...800 subi \$29, \$29, 64 ; nouă slivă
 ...804 — ; de 64 bytes

...856 —
 ...860 jr \$31 ; reinicioare

...280 lw \$19, 12(\$29) ; refac reg. \$A3
 284 lw \$18, 16(\$29) ; refac reg. \$A2
 288 lw \$30, 20(\$29) ; refac vechiul fp
 28C lw \$31, 24(\$29) ; refac adresa de reinicioare
 ...290 addi \$29, \$29, 32 ; refac vechiul sp

294 jr \$31

PROGRAMA L

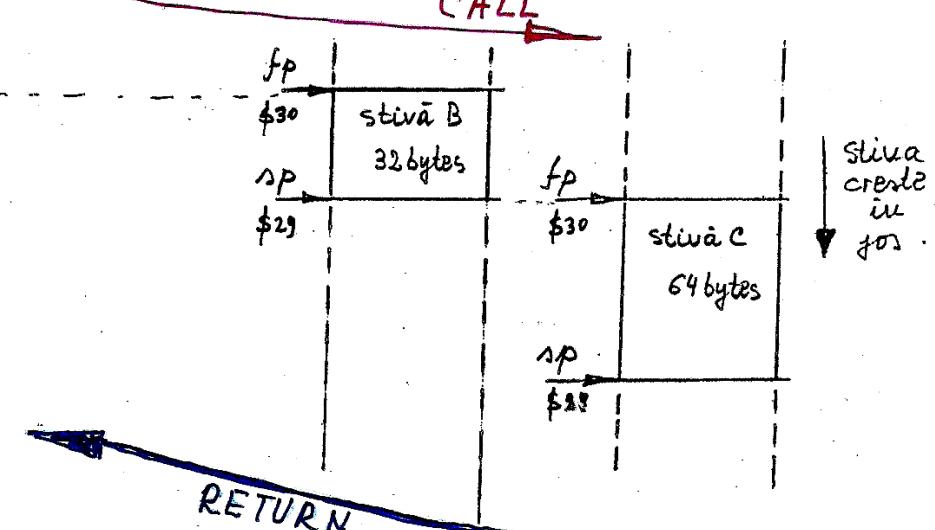
fp

\$30

STIVĂ A

sp

\$29



2.5.3.4 Evenimente de excepție, EIT (Exceptions, Interrupts, Traps)

În afară de instrucțiunile de ramificație, de cele de apelare de subrutine există și alte evenimente care îintrerup secvențialitatea accesării adreselor din program în ordinea de creștere a numerelor naturale, aceste evenimente sunt referite prin EIT. La producerea unora din aceste evenimente, la fel ca și instrucțiunile CALL-RETURN, se poate salva sau nu adresa de reîntoarcere în programul curent după rularea unei subrutine care rezolvă cauza care a generat acel eveniment. Dar spre deosebire de cele două tipuri de instrucțiuni anterioare (de ramificație, CALL-RETURN) care îintrerup secvențialitatea de accesare în ordinea numerelor naturale a adreselor, evenimentele de tip EIT nu sunt prevăzute în program, ele apar în mod aleatoriu pe durata rulării programului. Aceste evenimente sunt referite printr-o multitudine de termeni, care, uneori, sunt proprii firmei producătoare microprocesorului, după cum se exemplifică prin termenii din tabelele următoare.

Câteva atrbute ale evenimentelor de excepție, existente în tabelul de mai jos, sunt explicate în continuare.

– Evenimentul este *sincron* dacă apare totdeauna în același punct al programului dacă programul este executat cu aceleași date și aceeași alocare de memorie.

– *User request versus coerced*, dacă programul utilizator cere situația de excepție, spre deosebire de coerced (silit, constrâns) exception care este cauzată de anumite evenimente hardware care nu sunt sub controlul programului.

– *Between versus within instruction*, evenimentul care produce excepția este recunoscut/acceptat chiar în punctul în care apare din interiorul instrucțiunii (within) sau recunoașterea evenimentului este amânat până la sfârșitul (execuției) instrucțiunii (between), adică între două instrucțiuni.

– *Resume versus terminate*, dacă procesorul are mecanismul ca să reia rularea programului curent din punctul unde a apărut excepția (resume) sau se termină programul curent (terminate).

– *User maskable versus nonmaskable*, dacă utilizatorul poate opri/(mascable) ca excepția să fie recunoscută de procesor sau nu poate fi mascată de către utilizator (nonmaskable).

În cadrul acestui curs vor fi utilizați termenii de excepție când cauza este internă (uzual de natură software-întrerupere soft) și termenul de întrerupere când cauza este externă (uzual întrerupere hardware).

	Exception event	IBM 360	VAX	Motorola 680x0	Intel 80x86
1	I/O device request	Input/output interruption	Device interrupt	Exception (level 0...7 autovector)	Vectored interrupt
2	Invoking the operating system service from a user program	Supervisor call interruption	Exception (change mode supervisor trap)	Exception (unimplemented instruction)—on Macintosh	Interrupt (INT instruction)
3	Tracing instruction execution	Not applicable	Exception (trace fault)	Exception (trace)	Interrupt (single-step trap)
4	Breakpoint	Not applicable	Exception (breakpoint fault)	Exception (illegal instruction or breakpoint)	Interrupt (breakpoint trap)
5	Integer arithmetic overflow or underflow; FP trap	Program interruption (overflow or underflow exception)	Exception (integer overflow trap or floating-point underflow fault)	Exception (floating-point coprocessor errors)	Interrupt (overflow trap or math unit exception)
6	Page fault (not in main memory)	Not applicable (only in 370)	Exception (translation not valid fault)	Exception (memory-management unit errors)	Interrupt (page fault)
7	Misaligned memory accesses	Program interruption (specification exception)	Not applicable	Exception (address error)	Not applicable
8	Memory protection violations	Program interruption (protection exception)	Exception (access control violation fault)	Exception (bus error)	Interrupt (protection exception)
9	Using undefined instructions	Program interruption (operation exception)	Exception (opcode privileged/ reserved fault)	Exception (illegal instruction or breakpoint/unimplemented instruction)	Interrupt (invalid opcode)
10	Hardware malfunctions	Machine-check interruption	Exception (machine-check abort)	Exception (bus error)	Not applicable
11	Power failure	Machine-check interruption	Urgent interrupt	Not applicable	Nonmaskable interrupt

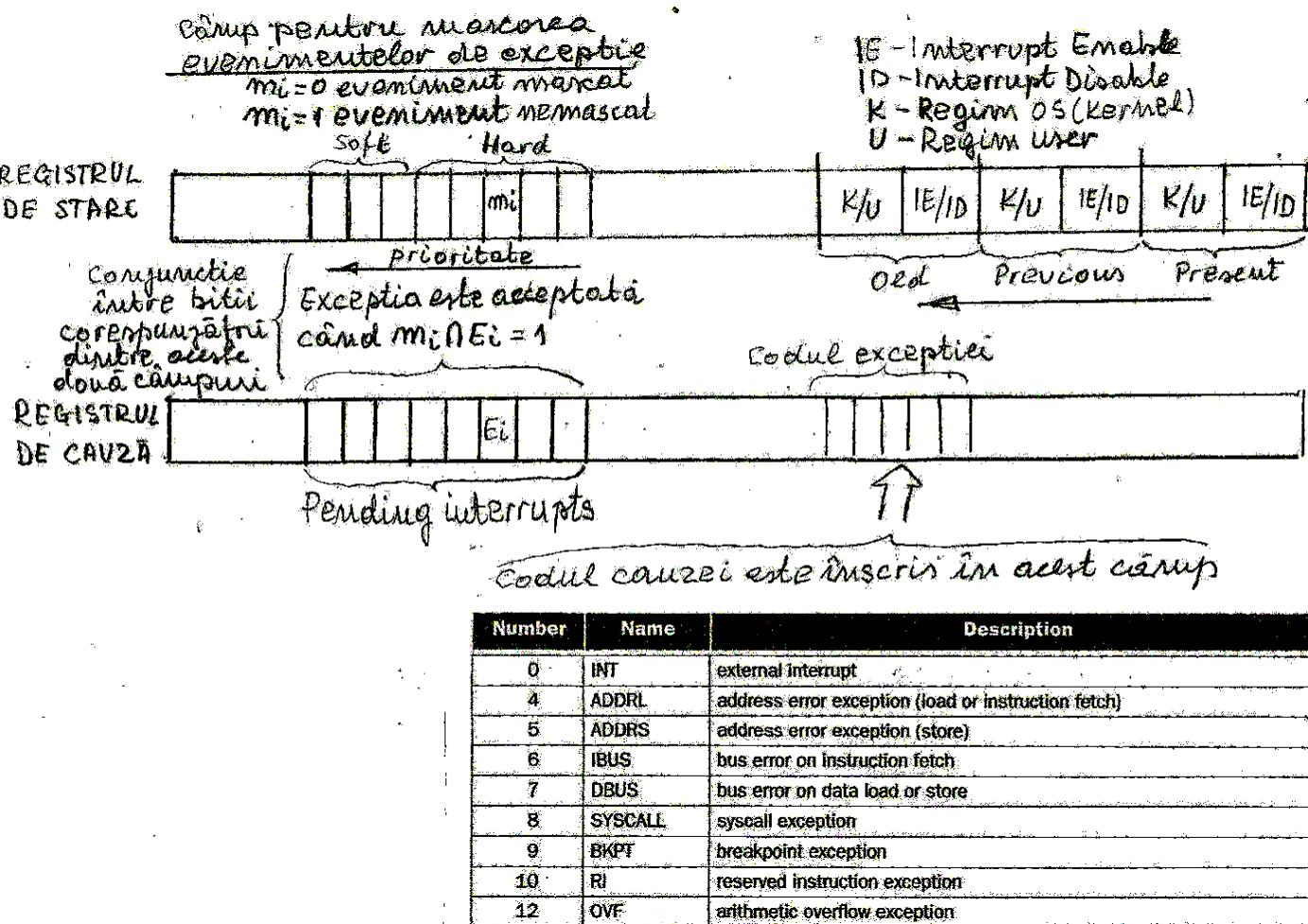
Figure A.26 The names of common exceptions vary across four different architectures. Every event on the IBM 360 and 80x86 is called an *interrupt*, while every event on the 680x0 is called an *exception*. VAX divides events into *interrupts* or *exceptions*. Adjectives *device*, *software*, and *urgent* are used with VAX interrupts, while VAX exceptions are subdivided into *faults*, *traps*, and *aborts*.

Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
1 I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
2 Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
3 Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
4 Breakpoint	Synchronous	User request	User maskable	Between	Resume
5 Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
5 Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
6 Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
7 Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
8 Memory protection violations	Synchronous	Coerced	Nonmaskable	Within	Resume
9 Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
10 Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
11 Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

Figure A.27 Five categories are used to define what actions are needed for the different exception types shown in Figure A.26. Exceptions that must allow resumption are marked as resume, although the software may often choose to terminate the program. Synchronous, coerced exceptions occurring within instructions that can be resumed are the most difficult to implement. We might expect that memory protection access violations would always result in termination; however, modern operating systems use memory protection to detect events such as the first attempt to use a page or the first write to a page. Thus, CPUs should be able to resume after such exceptions.

- Rezolvarea excepțiilor/întreruperilor. În principiu o excepție/întrerupere este rezolvată în modul următor:
 - Dacă evenimentul de excepție/întrerupere fost acceptat/recunoscut de μP, se salvează în registrul program counter de excepții, EPC (Exception PC), adresa instrucțiunii de unde se va relua programul curent întrerupt (uzual este adresa instrucțiunii care a provocat întreuperea/excepția);
 - Controlul μP este trecut de la programul curent la coordonarea de către Sistemul de Operare (SO);
 - SO determină care este cauza evenimentului de excepție/întrerupere. Pentru aceasta SO trebuie să primească o informație care poate proveni:
 1. dintr-un registru de cauză, în care este înscrisă cauza de către evenimentul care a apărut sub forma unui cod (asignat pentru fiecare tip de cauză, în general pentru întreruperile soft);
 2. sub forma unui vector (specific fiecărei cauze, în general pentru întreruperile hard) generat chiar de către evenimentul care a generat întreuperea (referită ca întreupere vectorizată).
 - Pe baza acestei informații obținute, SO determină adresa de început unde este plasată în memorie subrutina specifică pentru rezolvarea/servirea evenimentului respectiv. Fiecare tip de eveniment are o subrutină de servire specifică stocată în memorie;
 - După rularea subrutinei de servirea evenimentului se decide dacă programul curent se reia (resume) sau se termină.
- Rezolvarea excepțiilor la MIPS. Excepțiile la MIPS sunt efectuate de către un coprocesor, iar mecanismul de rezolvare a acestor excepții se bazează pe informația din următoarele patru registre

Denumire registru	Explicații
BadVAddress	Bad Virtual Address. În momentul când se accesează o adresă virtuală, care nu există în memorie (Page Fault Address) adresa respectivă este înscrisă în registrul BadVAddress.
EPC	PC pentru excepții. În acest registru se înscrive adresa instrucțiunii la care a apărut evenimentul de excepție (în vederea, eventual, a reluării programului de la această adresă).
Cause	Registru de cauză. În acest registru se înscrive codul pentru fiecare tip de cauză care a generat un eveniment, acest cod este informație de identificare a cauzei pentru sistemul de operare.
Status	Registru de stare. În acest registru se înscriu biții măștii prin care se permite recunoașterea/acceptarea unui eveniment de excepție, precum și regimul de funcționare al μP în care a apărut evenimentul (regim user sau regim SO - kernel)

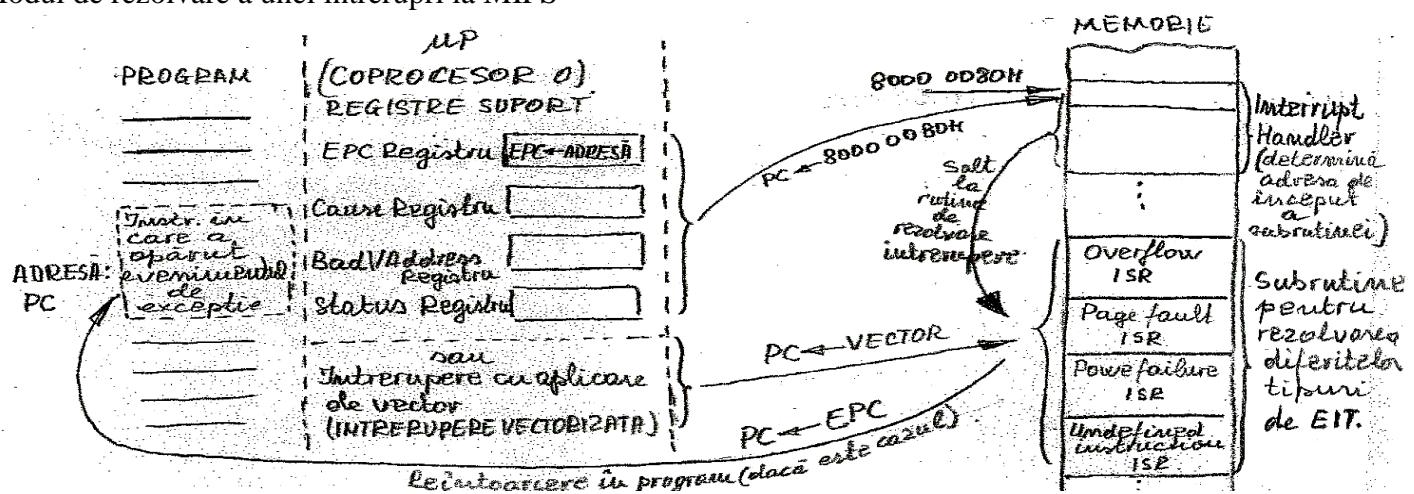


În registrul de stare cele opt niveluri de evenimente (trei soft și cinci hard) pot fi măcate prin înscrierea bitului corespondent: $m_i = 0$, sau nemăcate $m_i = 1$. Totodată, se poate cunoaște dacă evenimentul a fost generat în regimul de funcționare sub SO ($k/u = 0$) sau în regimul de funcționare utilizator ($k/u = 1$), atât pentru evenimentul curent cât și pentru cele două evenimente anterioare (cei șase biți sunt într-un registru de deplasare). Înscrierea bitul IE/ID specifică dacă intreruperea este permisă ($IE/ID = 1$) sau nu este permisă ($IE/ID = 0$).

În registrul de cauză, în câmpul pending interrupts, fiecare din cele opt niveluri de intreruperi, în momentul când sunt generate va înscrise bilul respectiv, $E_i = 1$. Acceptarea intreruperii corespondătoare unui din cele opt niveluri se realizează numai dacă nivelul respectiv nu este măcat ($m_i = 1$) și $E_i = 1$, ($m_i \cap E_i = 1$), iar $IE/ID = 1$. Cauza care a generat un eveniment, la producerea acestuia, este înscrisă sub forma unui cod în câmpul codul excepției din registrul de cauză (codurile pentru cauzele evenimentelor sunt specificate în tabelul de sub registrul de cauză).

La acceptarea unei excepții, adresa instrucției care a generat evenimentul de excepție este înscrisă în registrul EPC, iar adresa fixă 8000 0080H (adresă situată în segmentul kernel din memorie) este introdusă în PC; controlul µP este preluat de SO. La această adresă există o subrutină (interrupt handler) care pe baza codului din registrul de cauză sau pe baza vectorului livrat de eveniment (întrerupere vectorizată) determină un salt la adresa de început a subrutinei de servire specifică tipului de eveniment de excepție. Subrutina de servire a evenimentului va rezolva evenimentul respectiv după care se va relua programul curent din punctul de intrerupere sau se va opri rularea programului curent.

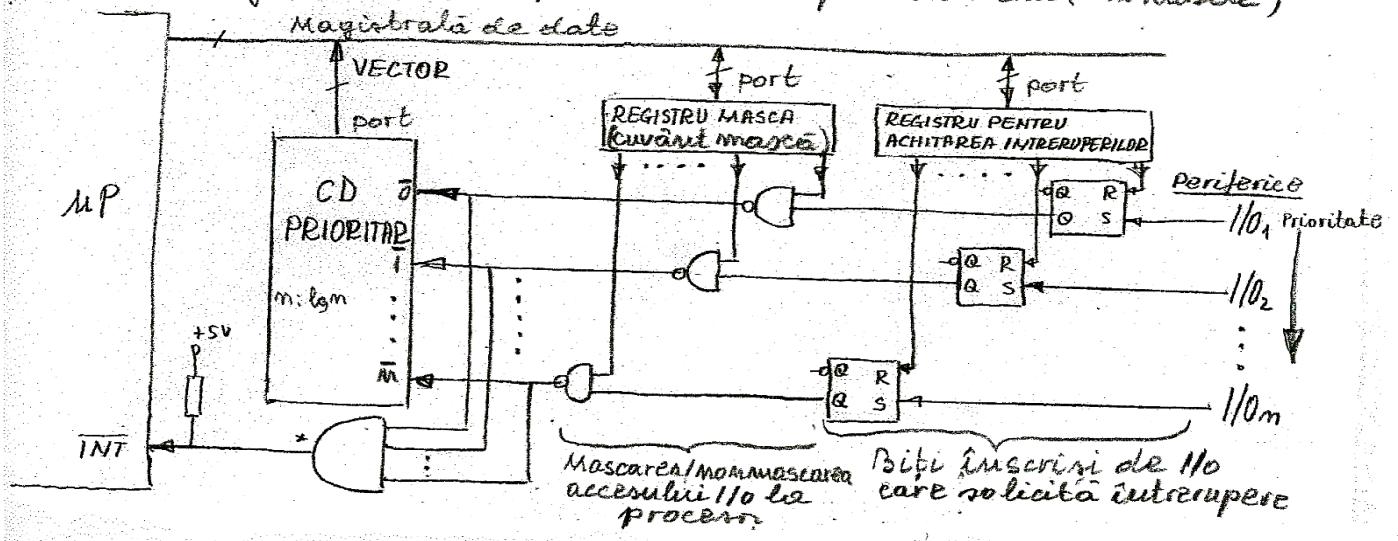
Modul de rezolvare a unei intrerupri la MIPS



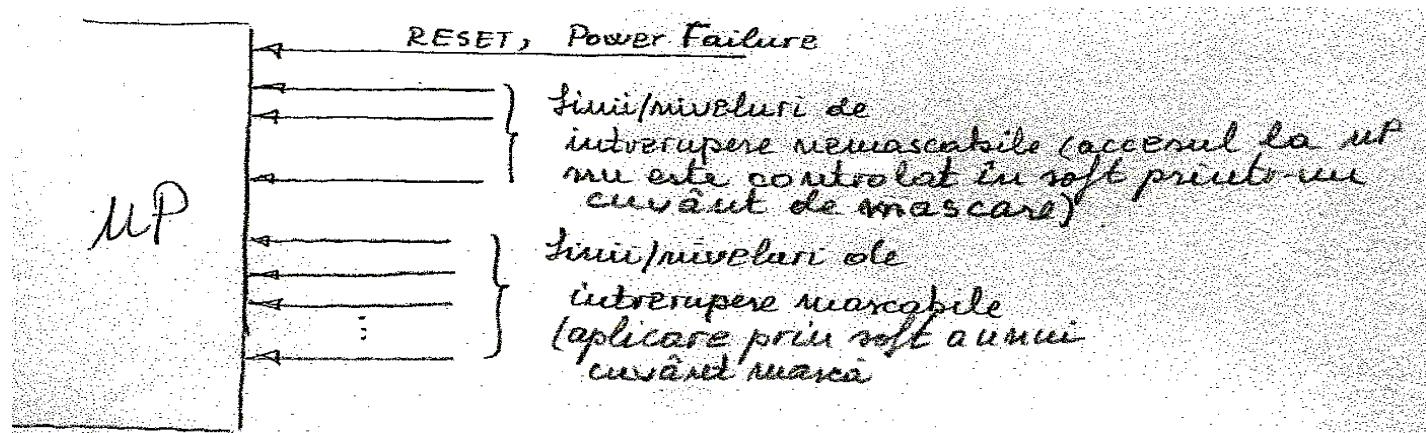
- Structurarea unui sistem de intreruperi externe vectorizate.

Structura din desenul următor prezintă o variantă posibilă (simplificată) de implementare a intreruperilor externe pentru setul de periferice I/O₁ – I/O_n. La activarea unei sau unor cereri de intrerupere către μP, de la unul sau mai multe din periferice, această/aceste cerere se memorează în latch-ul RS și se aplică, dacă nu este mascată, la codificatorul prioritar, CD_n:ln n și la poarta AND cu colectorul în gol. Când cel puțin o cerere de intrerupere este activă semnalul la borna INT a μP devine activ, iar la ieșirea CD se obține codul perifericului cu prioritatea cea mai ridicată care a cerut intreruperea. μP va citi codul (VECTORUL) perifericului, aplicat pe magistrala de date și va determina pe baza acestui vector adresa de început a subroutinesi specifică de servire a acelui periferic. Totodată, μP aplică pe magistrala de date un cuvânt care înscrie în (portul) Registrul pentru Achitarea Intreruperilor, prin conținutul acestui cuvânt latch-ul RS al perifericului care a solicitat intreruperea este comandat în zero (Q=0); deci perifericul în curs de servire de μP nu va mai solicita în continuare intrerupere către μP decât numai când rutina de servire s-a terminat, dacă între timp a activat din nou o cerere de intrerupere. Prin cuvântul înscris în Registrul Mască se selectează care din cererile de intrerupere de la periferice au acces la μP. Dacă mai multe periferice solicită în același timp servirea de către μP, va fi servit doar perifericul care are prioritățea cea mai ridicată și nu este mascat (Nivelul de prioritățe se stabilește în funcție de ordinea de realizare a conexiunilor prifericelor la intrarea codificatorului prioritar, CD).

- Organizare de principiu pentru intreruperi vectorizate (marcabile)

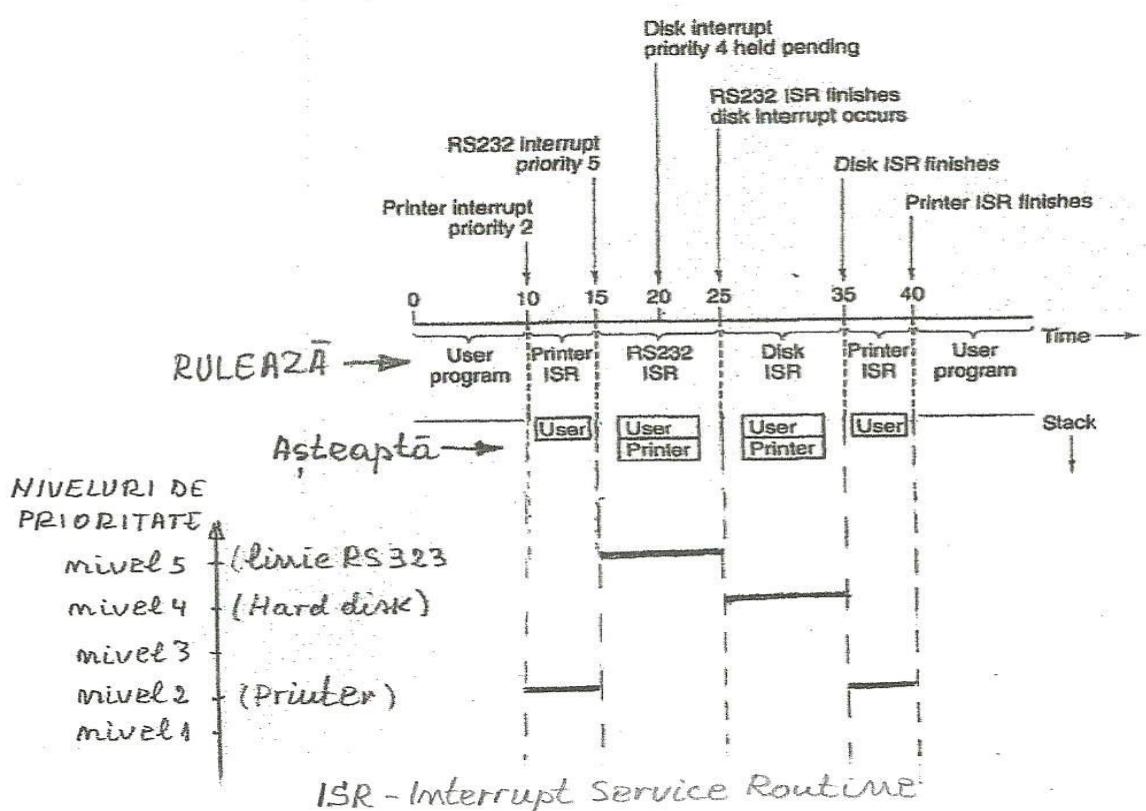


- Niveluri (linii) de întreruperi



Nivelurile de întrerupere (linii) la un μP li se asignează diferite niveluri de prioritate. Nivelurile de prioritate sunt introduse pentru ca fiecare periferic să fie (dacă se poate) servit în timp real. Servirea în timp real înseamnă că perifericul să aibă sau să transmită date atunci când este nevoie (datele sunt valide, sunt necesare). Dar viteza perifericelor pentru schimbul de date este foarte diferită, în consecință rezultă că un periferic de viteză mare, pentru a se asigura servirea în timp real, trebuie servit înaintea unui periferic de viteză scăzută, dacă solicită în același timp intervenția microprocesorului. În desenul următor se exemplifică modul de servire de către μP a unor periferice care au niveluri de prioritate diferite în sistem.

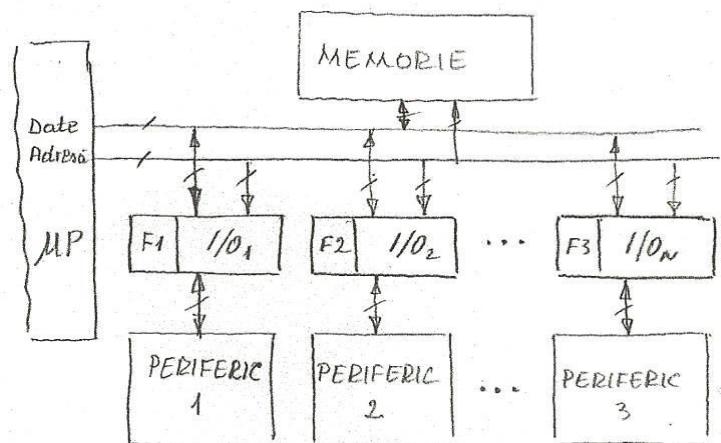
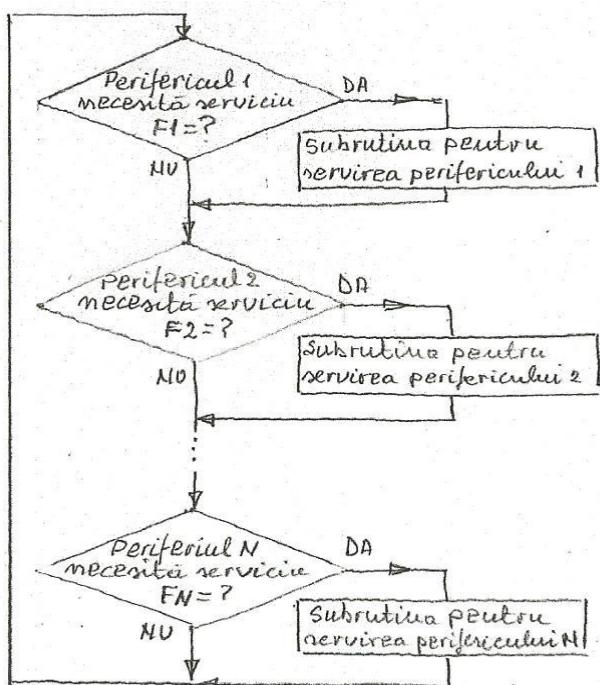
- Niveluri de prioritate în servirea exceptiilor/interrupțiilor



2.5.4 Modalități de lucru ale procesorului cu perifericele

Într-un sistem există trei modalități de lucru ale procesorului cu I/O: 1. metoda prin interogare ; 2. metoda prin întreruperi; Metoda DMA.

1. *Metoda prin interogare/scrutare (polling)*. Procesorul testează pe rând fiecare periferic din sistem (registru de stare al acestora) și constată dacă perifericul respectiv a solicitat intervenția procesorului, dacă da o oferă dacă nu trece la scrutarea următorului periferic; de fapt este o buclă programată în care operează procesorul (de unde metoda se mai numește și I/O programată). Registrul de stare testat se poate reduce la un singur bit, fanion de stare, F_i , a cărui valoare se testează



Avantajele metodei de lucru prin interogare:

- simplitate pe partea de hardware;
- scutarea perifericelor este sincronă cu modul de execuție al programului (buclei de scutare).

Dezantajele metodei de lucru prin interogare:

- încarcă sistemul cu timpul de testare în buclă (timp în care μP ar putea executa altă activitate);
- timpul de răspuns al sistemului la o necesitate de servire a unui periferic este mare, fiind cel puțin egal cu intrevalul de apelarea subruteinei de apelare.

Metoda este potrivită pentru periferice cu viteză relativ redusă.

2. Metoda prin întreruperi (interrupt-driven I/O).

Utilizând cererea de întrerupere către μP, activată de către un I/O; nu mai este nevoie de scrutarea periodică în buclă a fiecărui periferic ca la metoda programată, deci timpul consumat de procesor se reduce , totodată se reduce și timpul de răspuns la servirea unui periferic. Modul cum se realizează procesul de întreruperi a fost prezentat în secțiunea 2.6.3.4. Metoda este potrivită pentru periferice de viteză ridicată.

2. Metoda prin accesul direct la memorie, DMA (Direct Memory Acces).

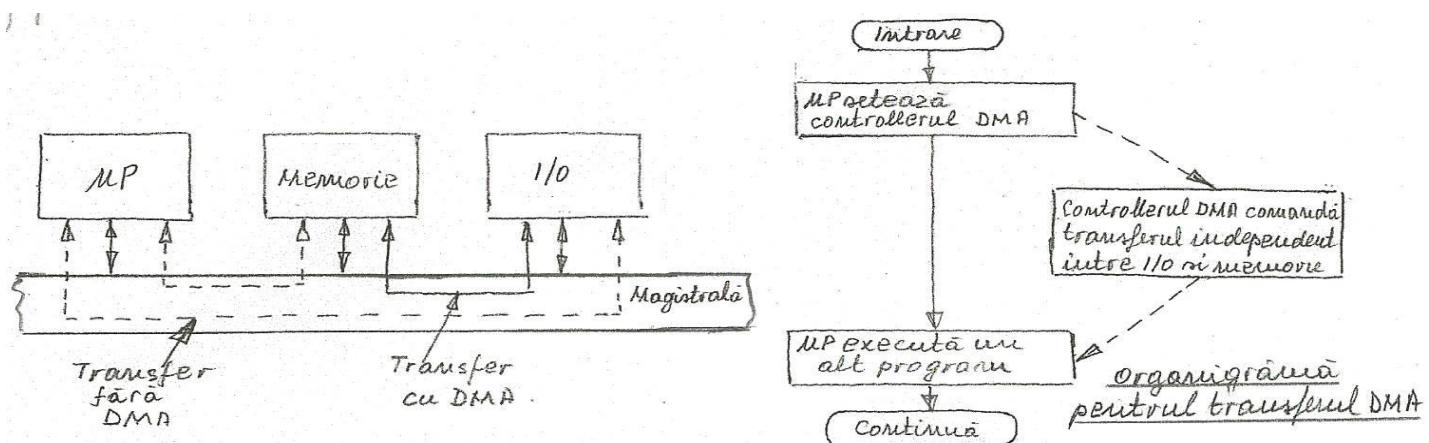
Această metodă este recomandată pentru schimb de date între memorie și periferice cu Bandwidth foarte ridicat, de exemplu ; hard disk, dispozitive video, rețele, camcordere etc. După cum și denumirea metodei exprimă, schimbul de date (pe blocuri) între un periferic și memorie se face direct , fără intervenția microprocesorului, controlul transferului pe magistrală este preluat de controllerului DMA, care devine master de magistrală. Există trei pași în realizarea unui transfer DMA.

a) Procesorul prescrie în controllerul DMA adresa perifericului care va fi implicat în transfer, tipul de operație pe care se va efectua (read, write), adresa de început din memorie pentru sursă (citirea memoriei) sau pentru destinație (înscrierea memoriei), numărul de bytes implicat în transfer. Pentru realizarea acestor operații, într-o organizare minimală controllerul DMA trebuie să conțină un registru în care μP înscrie adresa de început (sursă sau destinație) din memorie și apoi acest registru va funcționa ca un Program Counter pentru adresele din memorie. Un alt registru în care se înscrie numărul de bytes ai blocului de transfer și apoi acest registru are rolul de contor cu decrementare, când ajunge la zero s-a terminat transferul.

b) Controllerul DMA preia controlul pe magistrală și începe transferul direct între I/O și memorie, deci datele nu mai au traseul I/O-procesor-memorie sau memorie-procesor-I/O ci doar memorie \leftrightarrow I/O.

c) Odată terminat transferul, controllerul DMA printr-un semnal de intrerupere semnalizează procesorul să preia în continuare controlul pe magistrală.

De fapt transferul DMA poate fi privit ca o instrucțiune de deplasare masivă de date (bloc de date).



Într-un sistem pot exista mai multe controllere DMA, de exemplu într-un sistem cu o singură magistrală memorie-procesor și multiple magistrale I/O, pe fiecare magistrală I/O există câte un controller DMA care coordonează transferul între memorie și un dispozitiv I/O de pe acea magistrală.

În sisteme pentru a reduce implicarea procesorului în transferurile I/O și prin aceasta micșorarea timpului consumat de procesor, controllerele I/O sunt dotate cu inteligență. Controllerele inteligente sunt referite ca *procesoare I/O*. Astfel de procesoare specializate execută o serie de operații I/O denumite programe de intrare- ieșire, aceste programe de intrare- ieșire pot fi stocate în procesorul I/O sau pot fi stocate în memorie de unde sunt accesate de procesorul I/O. Când se utilizează un procesor I/O sistemul de operare printr-un program special de intrare- ieșire fixează operațiile care trebuie executate, mărimea blocurilor transferate, adresele sursă și destinație.

Un controller DMA este în esență un procesor pentru utilizări speciale (uzual pe un chip și neprogramabil) în timp ce un procesor specializat I/O adesea este implementat pe baza unui microprocesor de uz general care rulează un program specializat în operații de intrare- ieșire. Actual multe din aceste operații de intrare- ieșire sunt implementate în chip-seturile north și south bridge.

Lucrul cu perifericele într-un sistem se reduce, în fond, la transferuri de date, deci driverele pentru periferice (programe speciale de intrare- ieșire) sunt componente software sub sistemul de operare.

2.5.5 Instrucțiuni de control al procesorului

Instrucțiunile din această clasă, printr-un suport hardware adekvat, pot fi utilizate în supravegherea și dirijarea procesorului/sistemului. Cu cât mai complexe și mai performante devin μP cu atât se extinde această clasă de instrucțiuni. În continuare se vor prezenta unele din instrucțiuni care pot exista în ISA, corespunzătoare acestei clase.

- Instrucțiuni de: setare a biților din registru de stare, de validare/devalidare a biților din registrul de control al intreruperilor, de oprire temporară (halt, wait) etc.
- Instrucțiuni suport pentru testarea procesorului: prin fixarea opririi programului într-un punct fixat prin programare, prin testarea anumitor componente interne etc.

- Înstrucțiuni de sincronizare.

La sistemele uniprocesor multiprogram, deși există impresia că mai multe programe rulează în același timp, funcționarea se reduce doar la rularea unui singur program la un moment dat (ocupă procesorul); fiecărui program î se alocă o coantă de timp când este rulat pe procesor, deci resursele sunt alocate procesului respectiv pe durata acelei coante de timp, neexistând concurență la resurse între procese pe durata respectivei coante. La sistemele multiprocesor poate apărea concurență pentru aceeași resursă între procesele care rulează în același timp, deci se impune o sincronizare între aceste procese, în consecință în ISA sunt introduse instrucțiuni în acest scop. Sincronizarea între procese se impune, fie din cauza concurenței în același timp la aceeași resursă (resursă critică, RC), de exemplu pentru procesele de tip producer-consumer (producere și consumare de date), fie se impune o funcționare corelată, de exemplu terminarea tuturor iteratiilor unei bucle în programe paralele. Un exemplu de concurență a două procese la o resursă critică este prezentat în continuare, resursa critică fiind variabila A pe care o accesează două programe (procese) procesul P1 înscrie $A \leftarrow A+1$, iar procesul P2 înscrie $A \leftarrow A+2$.

Proces P1

 $A \leftarrow A+1$

```
load R1, A      ; R1 ← M[adr A]
addi R1, R1, 1   ; R1 = R1 + 1
store R1, A      ; M[adr A] ← R1
```

a)

```
load R1, A      ; R1 ← M[adr A]
încărcarea variabilei A de către P1
necesită un timp mai lung decât
încărcarea variabilei A de către P2
addi R1, R1, 1   ; R1 = R1 + 1
store R1, A      ; M[adr A] ← R1 = A + 1
```

b)

Proces P2

 $A \leftarrow A+2$

```
load R1, A      ; R1 ← M[adr A]
addi R1, R1, 2   ; R1 = R1 + 2
store R1, A      ; M[adr A] ← R1
```

```
load R1, A      ; R1 ← M[adr A]
addi R1, R1, 2   ; R1 = R1 + 2
store R1, A      ; M[adr A] ← R1
```

Cazul a) când este un singur procesor evident că procesele P1 și P2 sunt executate secvențial, indiferent care este executat primul, rezultatul este $A+3$. În cazul multiprocesor în funcție de timpii de acces spre variabila A și înscrierea acesteia în R1 dacă acești timpi sunt diferenți pentru P1 și P2 valoarea rezultată pentru A poate fi $A+1$, $A+2$, $A+3$. De exemplu, în cazul b) dacă timpul de accesare și înscriere în registrul pentru P1 (variabila A nu este în cache-ul elementului de procesare de la P1) este mai mare decât pentru P2 (variabila A este în cache-ul elementului de procesare de la P2) rezultatul ar putea fi $A+1$. Variabila A este citită de P2 și începe procesarea dar înainte de terminare este citită și de P1, între timp P2 termină și înscrie $A+2$, dar apoi termină și P1 care înscrie în final $A+1$; se pot analiza toate cazurile posibile în funcție de diferențe întârzieri de accesare și înscriere. Pentru a evita astfel de cazuri, resurselor critice li se alocă un zăvor (lock), adică un bit într-o locație de memorie; accesul la resursa critică este posibilă numai când zăvor = 0 și nu este posibilă când zăvor = 1.

Pentru sincronizarea la resurse critice în ISA s-au introdus instrucțiuni de sincronizare, una dintre primele instrucțiuni introdusă și cea mai simplă este instrucțiunea **test-and-set** (zăvor). Înainte de accesarea resursei critice procesul testează zăvorul, dacă zăvor = 1 va testa în contiunare iar când găsește zăvor = 0, instrucțiunea se execută prin: citește zăvor, modifică/înscrie $\text{zavor} \leftarrow 1$ și înscrie înapoi în memorie zăvor = 1, deci se preia în continuare controlul asupra resursei critice (începând din acest moment oricare instrucțiune care dorește să acceseze resursa critică va găsi zăvor = 1). Instrucțiunea test-and-set este indivizibilă (atomică), adică nu poate fi întreruptă pe timpul execuției. Odată preluat controlul asupra resursei critice (zăvor = 1) se menține acest control până terminarea sarcinii care necesită resursa critică, iar la sfârșit se eliberează resursa critică prin înscrierea în memorie $\text{zavor} \leftarrow 0$

while ($\overline{\text{zavor}} = 0$) { testare zăvor}

se utilizează resursa critică

zăvor←0

Dezavantajul aceste instrucțiuni simple de sincronizare este provocat de timpul (care poate fi) lung de testare în buclă a zăvorului, deoarece procesorul este într-o buclă de așteptare (do-while). Pentru alte instrucțiuni de sincronizare mai complexe acest dezavantaj este atenuat sau chiar eliminat. În IA-32 cele mai multe instrucțiuni aritmetice și logice pot fi transformate în instrucțiuni atomice prin prescrierea unui prefix (lock prefix) atașat instrucțiunii.

Programul anterior, pentru accesarea resursei critice, variabila A, corespunzător procesului P1, prin utilizarea instrucțiunii test and-set este:

```

Loop: test-and-set R2, zăvor ; test-and-set valoarea din locația zăvor
      bnz      R2, loop ; dacă zăvor =1, resursa ocupată, atunci salt la Loop
      dacă zăvor =0 se preia controlul asupra resursei critice:
      ld      R1, A ; resursa critică (A) a fost preluată
      addi    R1,R1,1 ; variabila A se incrementează
      st      R1, A ; se stochează valoarea variabilei A
      se elibereză controlul asupra resursei critice:
      st      R0, zăvor ; se înscrive zăvor = 0, resursa critică devine liberă, poate fi accesată de un
                      ; alt proces

```

- **Înstrucțiuni privilegiate.** Procesoarele evoluate permit două moduri de funcționare: **modul sistem** (modul sistem de operare, supervisor sau kernel) și **modul utilizator** (user).

Fixarea unuia din modurile de funcționare se face prin înscrierea unui bit de mod sistem (user/kernel), corespunzător în ISA există două tipuri de instrucțiuni: *instrucțiuni privilegiate și instrucțiuni normale*. Rularea în mod supervisor poate utiliza toate instrucțiunile, pe când rularea în mod user poate utiliza doar instrucțiunile normale, orice tentativă de a utiliza din mod user o instrucțiune privilegiată generează o intrerupere indicând violarea sistemului. Printre instrucțiunile privilegiate se găsesc toate instrucțiunile care prin execuția lor fac sistemul vulnerabil cum sunt: unele instrucțiuni de I/O, instrucțiuni care controlează parametrii de protecție a memoriei, instrucțiuni care controlează parametrii de mapare a adreselor, instrucțiuni de control al mascării intreruperilor și priorităților, a unor registre de stare etc. Trecerea din mod user în mod sistem se face prin înscrierea bitului mod sistem. Această trecere este controlată implicit prin hardware, tipic, se realizează prin apariția evenimentelor de excepție/intrerupere. Când un astfel de eveniment este acceptat nu se salvează numai registrele dar se devalidează și accesul altor cereri de intreruperi care ar putea apărea și se înscrive bitul în mod sistem, iar la reîntoarcere din sistemul de operare, ultima instrucțiune Return From Exception (rfe) automat va înscrive bitul în mod user.

2.5.6 Instrucțiuni de nivel înalt, HLL

Sarcina, normală, care o urmărește un utilizator este să își rezolve problemele cu calculatorul, adică să transforme universul problemei în universul calculatorului. Acest transfer este realizat printr-un limbaj de programare. Universul calculatorului (procesorului) se bazează pe transferări și transformări de siruri de biți (cuvinte), acestea le corespund printr-o aplicație bijectivă anumite mnemonice care formează limbajul de asamblare. Pe de altă parte, generalitatea (și apoi decompozabilitatea) universului problemei necesită o abstractizare a abordării (algoritmice). Pentru o astfel de abordare abstractă, mai potrivită și independentă de procesor, sunt limbajele de nivel înalt **HLH (High Level Language)**, fie de tip imperativ fie de tip funcțional. Limbajele imperative (C, BASIC; FORTRAN, PASCAL, ADA, JAVA etc) descriu procesările prin secvențe de comenzi adresate unui procesor. În schimb limbajele funcționale (LISP, PROLOG) permit ca întregul program să fie văzut ca descrierea unor funcții în matematică. Între universul problemei (algoritmice) și universul calculatorului (siruri de biți), între limbajele de nivel înalt (abstract) și respectiv limbajul de asamblare (mașină) există o diferență conceptuală, o **diferență semantică (semantic gap)**.

UNIVERSUL PROBLEMEI
(Abstract, Algoritmic)



UNIVERSUL CALCULATORULUI
(Hardware, Siruri de biți)

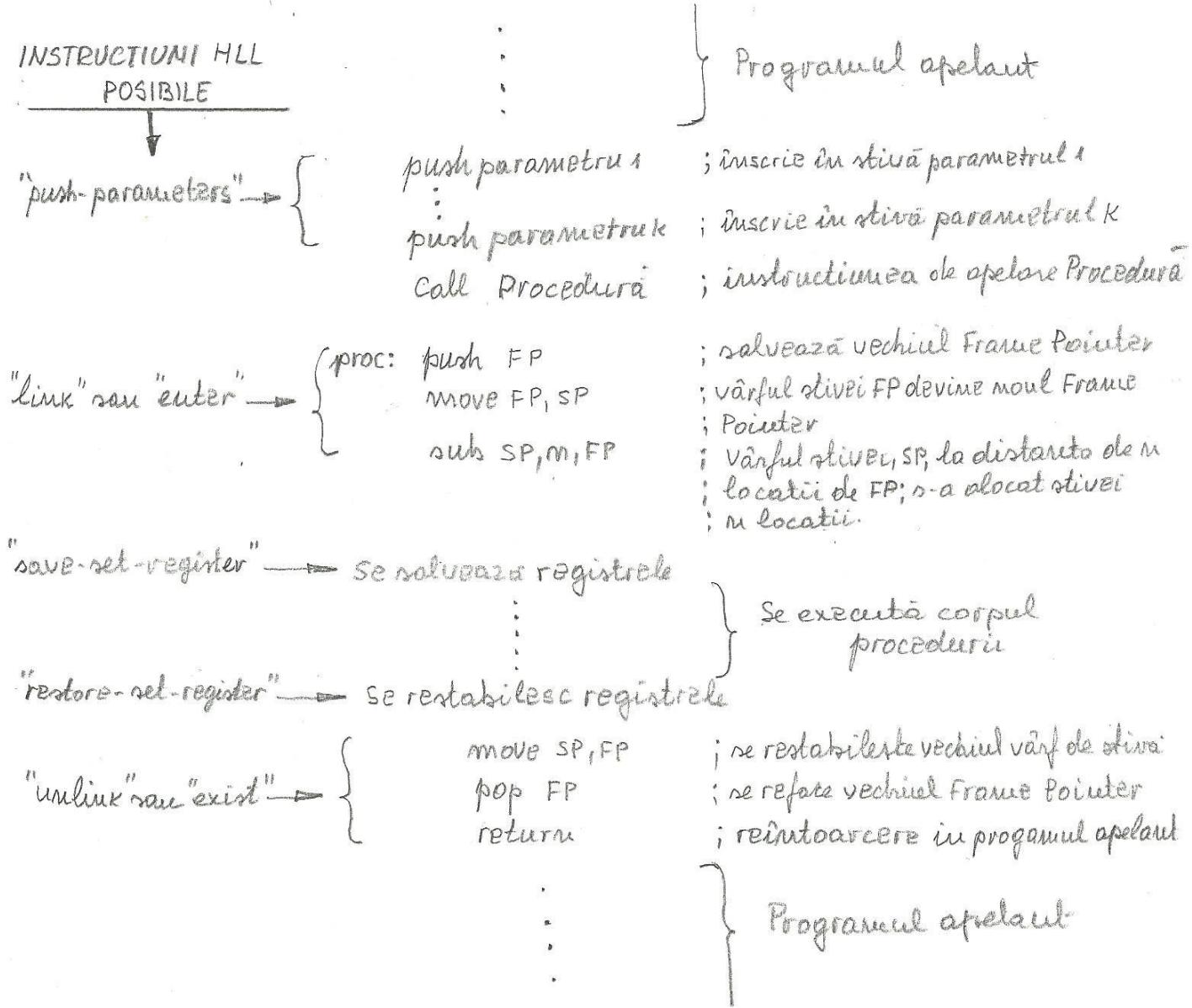


Pentru reducerea diferenței semantice o parte din instrucțiunile limbajului de asamblare au fost extinse la nivelul semantic al instrucțiunilor din HLL, apărând astfel instrucțiuni (mașină) de nivel înalt. Unei instrucțiuni de nivel înalt, implementată în procesor fie cablat sau microprogramat (deci devenită instrucțiune în limbaj de asamblare), la fel ca tuturor instrucțiunilor de asamblare clasice, îi corespunde unu-la-unu o instrucțiune în cod mașină. Rezultatul în procesare pentru o instrucțiune HLL este echivalent cu cel obținut de toate instrucțiunile (clasice) de asamblare produse în urma compilării acestei instrucțiuni de limbaj înalt când aceasta nu este implementată în ISA. Instrucțiunile de nivel înalt acoperă procesări cum ar fi: accesul datelor structurate de tip matriceal, procesarea de date de tip multimedia, manevrarea sirurilor, apelarea procedurilor și schimbarea cotextului. I/O; în continuare se va exemplifica prin analiza unor astfel de (posibile) instrucțiuni.

– Instrucțiunea Compară-între-limite determină dacă un index calculat pentru un element al unei structuri de date de tip matriceal se încadrează între valoarea inferioară și superioară a indexului (dacă există depășirea intrevalului). La o situare în afara intrevalului se înscrive un bit de cod de condiție (cc) și, eventual, se generează o excepție.

– Instrucțiuni pentru procesarea de texte care manipulează siruri sau câmpuri de biți. Astfel de instrucțiuni pot fi: Copiază-sir, deplasează un sir dintr-o zonă de memorie într-o altă zonă de memorie, o variantă a acestei instrucțiuni ar putea fi Deplasează-bloc care specifică blocul (lungimea) de copiat. Sfârșit-sir localizează caracterul de sfârșit de sir, această instrucțiune poate fi utilizată pentru calculul lungimii unui sir sau combinată cu Copiază-sir poate realiza operația de anexează sir. Caută-caracter, realizează într-un sir o căutare secvențială (caracter după caracter) până când găsește caracterul specificat, când caracterul specificat este găsit se întrerupe căutarea iar adresa caracterului se înscrive într-un registru. Acțiunea de comparare doar pentru un caracter se poate extinde la acțiunea de comparare a două siruri cu posibilitatea de contor (incrementare/decrementare). Instrucțiuni asupra câmpurilor de biți (nu neapărat cu lungimi de byte sau multipli de bytes) pot fi: Inserare-câmp, Extragere-câmp, Căutare- câmp, Alăturare-câmp etc.

– Instrucțiuni pentru lucru cu proceduri. La apelarea unei proceduri (vezi secțiunea 2.6.3.3) trebuie salvat contextul, iar la revenire acesta trebuie refăcut. Termenul de **context** referă toate date care sunt necesare pentru reluarea rulării unui proces (registre, registrul de stare al procesului (PSW- Program Status Word- care include PC+ registrul codurilor de condiții (cc)), adresele din memorie ale tabelului procesului etc). Pentru pașii de realizat la salvare și la refacerea contextului grupurile de instrucțiuni de asamblare pot fi combinate/integrate doar în câteva instrucțiuni de nivel înalt, cum se exemplifică mai jos



2.6 FORMATUL INSTRUCȚIUNILOR

Cuvântul instrucție, într-o concepție structurată, este împărțit în câmpuri/subcâmpuri, fiecărui câmp atribuindu-i-se responsabilitatea unei comenzi/acțiuni în cadrul instrucționii. Cei n biți ai unui câmp pot fi utilizati pentru o codificare globală sau liniară. La *codificarea liniară*, fiecare bit din cei n biți ai subcâmpului prin valoarea sa, unu sau zero, este responsabil de o anumită comandă/acțiune, în total n comenzi sunt posibile, iar decodificarea nu este necesară. La *codificarea globală* un cuvânt de cod este responsabil de o anumită comandă, în total 2^n comenzi sunt posibile (numărul total de cuvinte de cod formate cu n biți), dar apoi este necesar un decodificator DCD n: 2^n . Modul de codificare afectează nu numai formatul instrucționii și mărimea programului după compilare, ci afectează însăși implementarea procesorului în ceea ce privește decodificarea rapidă în aflarea operației și a operanzilor (deci viteza de procesare).

Pentru obținerea unui cod cât mai compact al programului se poate realiza o codificare pe câmpuri în funcție de frecvența comenzi/acțiunii respective, pentru acțiunile cu frecvență mai ridicată se impun codificări scurte ca număr de biți sau chiar codificare liniară. Într-o instrucție există câmpuri care există în fiecare instrucție, ca de exemplu câmpurile OPCODE, operanzii sau modurile de adresare și câmpuri care se introduc numai pentru o anumită interpretare cerută de instrucție. Deoarece modurile de adresare și câmpurile de adresare ale regisrelor

pot apărea de mai multe ori în aceeași instrucțiune, codificarea aleasă pentru acestea este de cea mai mare importanță în contabilitatea bitilor dintr-o instrucțiune precum și pentru viteza de decodificare.

Realizarea unui set de instrucțiuni ortogonal este o cerință majoră. Un set de instrucțiuni este ortogonal dacă nu are restricții (sau foarte puține) în ceea ce privește combinarea celor trei componenente (subcâmpuri) primare dintr-o instrucțiune: 1. operațiile (OPCODE); 2. tipurile de date; 3. modurile de adresare. Proprietatea de ortogonalitate se reflectă prin faptul că pentru fiecare operație asupra unui tip de date se poate aplica oricare mod de adresare. Ortogonalitatea unui ISA simplifică dezvoltarea unui compilator de bună calitate și îmbunătățește productivitatea software.

Există trei modalități pentru formatul (lunginea) de instrucțiune din ISA.

– *Formatul variabil*, adică poate fi orice lungime de instrucțiune, evident multiplu de byte. Acest format permite, virtual, implemenetarea tuturor modurilor de adresare aplicabile pentru operații; se recomandă acest format când sunt multiple moduri de adresare.

– *Formatul fix*, o singură lungime de instrucțiunen (multiplu de byte), acest format, uneori, combină în câmpul OPCODE și unele moduri de adresare, de exemplu addi (îl specifică și adresarea imediată). Evident că formatul fix aproape că obligă la utilizarea unui număr restrâns de moduri de adresare și operații. Formatul fix este foarte potrivit pentru procesarea pipeline, este formatul pentru mașinile RISC. Acest format prezintă următoarele avantaje:

1. instrucțiunile nu vor ieși niciodată din pagină, alinierea instrucțiunilor în program este automată (modulo $k = 0$);
2. permite codificarea cu câmpuri fixe. Codificarea cu câmpuri fixe înseamnă că operanzii sursă și valorile de imediat sunt totdeauna în aceleași poziții fixe în instrucțiune, ceea ce permite citirea acestora (din acele locuri știute) înainte sau în paralel cu decodificarea OPCODE, deci în momentul când decodificarea este efectuată deja operanzii sunt obținuți (din câmpurile fixe) și se poate trece la execuția instrucțiunii, rezultând un timp redus pe ciclu instrucțiune.

– *Formatul mixt* este o combinație între primele două formate. În general, se admit două formate de instrucțiuni, unul normal și unul scurt. Cu formatul normal se obține o bună ortogonalitate, iar cu formatul scurt se obține un cod mai compact, deci o viteză mai ridicată și un consum de putere mai redus. Formatul scurt este cerut în special de aplicațiile de tip PMD, în acest sens arhitectura ARM a introdus o variantă de 16 biți (Thumb instruction set), la fel și arhitectura MIPS a introdus microMIPS instruction set. La formatul scurt, în raport cu formatul normal, se păstrează același set de operații, dar cu restricția că operanzii, moduri de adresare, operanzii immediat sunt un subset al celor din formatul normal.

Operation and no. of operands	Address specifier 1	Address field 1	• • •	Address specifier n	Address field n
-------------------------------	---------------------	-----------------	-------	-----------------------	-------------------

(a) Variable (e.g., VAX, Intel 80x86)

Operation	Address field 1	Address field 2	Address field 3
-----------	-----------------	-----------------	-----------------

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

Operation	Address specifier	Address field
-----------	-------------------	---------------

* Address specifier de adresare

Operation	Address specifier 1	Address specifier 2	Address field
-----------	---------------------	---------------------	---------------

Operation	Address specifier	Address field 1	Address field 2
-----------	-------------------	-----------------	-----------------

(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

Figure 2.23 Three basic variations in instruction encoding: variable length, fixed length, and hybrid. The variable format can support any number of operands, with each address specifier determining the addressing mode and the length of the specifier for that operand. It generally enables the smallest code representation, since unused fields need not be included. The fixed format always has the same number of operands, with the addressing modes (if options exist) specified as part of the opcode (see also Figure C.3 in Appendix C). It generally results in the largest code size. Although the fields tend not to vary in their location, they will be used for different purposes by different instructions. The hybrid approach has multiple formats specified by the opcode, adding one or two fields to specify the addressing mode and one or two fields to specify the operand address (see also Figure D.7 in Appendix D).

2.7 INTREDEPENDENȚA ARHITECTURA SETULUI DE INSTRUCȚIUNI-COMPILATOR

Performanțele unui procesor se obțin prin **sinergismul arhitectură-compilator**, deci arhitectura trebuie corroborată cu tehnologia compilatoarelor.

A. Cerințele impuse compilatorului:

1. să genereze un cod corect
2. să genereze un cod de înaltă calitate (compact și de viteză ridicată)
3. cerințe secundare:
 - compilare rapidă
 - suport pentru depanare
 - interoperabilitate între limbaje

Etapele și optimizările în procesul de compilare

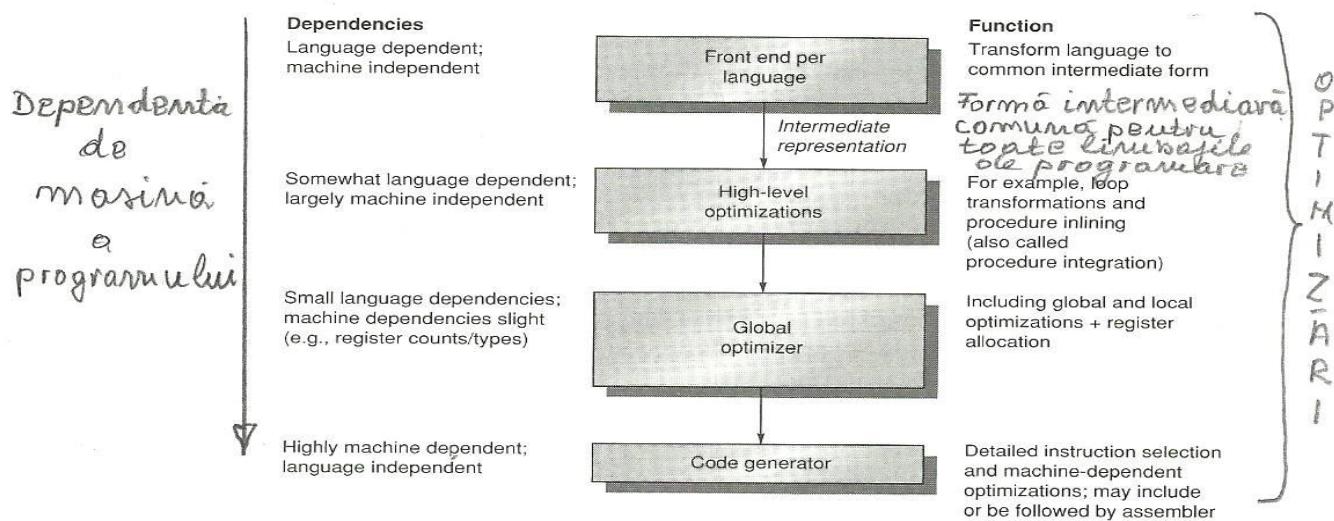


Figure 2.24 Compilers typically consist of two to four passes, with more highly optimizing compilers having more passes. This structure maximizes the probability that a program compiled at various levels of optimization will produce the same output when given the same input. The optimizing passes are designed to be optional and may be skipped when faster compilation is the goal and lower-quality code is acceptable. A *pass* is simply one phase in which the compiler reads and transforms the entire program. (The term *phase* is often used interchangeably with *pass*.) Because the optimizing passes are separated, multiple languages can use the same optimizing and code generation passes. Only a new front end is required for a new language.

Operații în cadrul fiecărui nivel de optimizare

Optimization name	Explanation	gcc level
<i>High level</i>	<i>At or near the source level; processor independent</i>	
Procedure integration	Replace procedure call by procedure body	O3
<i>Local</i>	<i>Within straight-line code</i>	
Common subexpression elimination	Replace two instances of the same computation by single copy	O1
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	O1
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	O1
<i>Global</i>	<i>Across a branch</i>	
Global common subexpression elimination	Same as local, but this version crosses branches	O2
Copy propagation	Replace all instances of a variable A that has been assigned X (i.e., $A = X$) with X	O2
Code motion	Remove code from a loop that computes same value each iteration of the loop	O2
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	O2
<i>Processor dependent</i>	<i>Depends on processor knowledge</i>	
Strength reduction	Many examples; replace multiply by a constant with shifts	O1
Pipeline scheduling	Reorder instructions to improve pipeline performance	O1
Branch offset optimization	Choose the shortest branch displacement that reaches target	O1

FIGURE 2.32 Major types of optimizations and examples in each class. The third column shows when these occur at different levels of optimization in gcc. The Gnu organization calls the three optimization levels medium (O1), full (O2), and full with integration of small procedures (O3).

Optimizations performed	Percent faster
Procedure integration only	10%
Local optimizations only	5%
Local optimizations + register allocation	26%
Global and local optimizations	14%
Local and global optimizations + register allocation	63%
Local and global optimizations + procedure integration + register allocation	81%

FIGURE 3.27 Performance effects of various levels of optimization. Performance gains are shown as what percent faster the optimized programs were compared to the unoptimized programs. When register allocation is turned off, data are loaded into, or stored from, the registers on every individual use. These measurements are also from Chow [1983] and are for 12 small FORTRAN and Pascal programs.

Optimizări:

- La nivel înalt, se realizează asupra programului sursă, iar ieșirea se aplică (influențează) etapele următoare ale compilării;
- La nivel local, se aplică asupra codului numai din cadrul unui bloc de bază (un segment de program între intrarea în acel segment până la prima instrucțiune de salt);
- La nivel global, extinde optimizarea locală și asupra salturilor și introduce un set de transformări în scopul optimizării buclelor;
- La nivel de mașină, se încearcă să se folosească avantajele particulare ale mașinii. Pentru această fază a compilatorului important este procesul de alocare a reștrilor mașinii

B. Suportul ISA pentru tehnologia compilatoarelor:

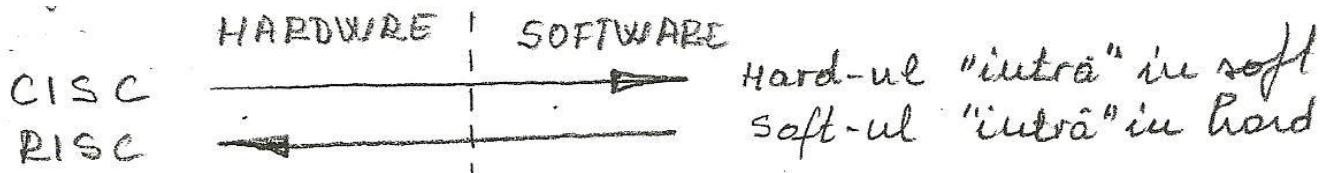
1. Set ortogonal de instrucțiuni
2. Generarea de primitive și nu ”soluții” potrivite doar unui singur limbaj
3. Reducerea numărului de alternative disponibile oferite compilatorului
4. Favorizarea instrucțiunilor care alocă în timpul compilării (static) în raport cu cele care alocă dinamic

Pot apărea următoarele nuanțe pentru proiectarea unei arhitecturi de procesor:

- Arhitectură orientată (ajută) spre compilator (compiler oriented architecture)
- Arhitectură orientată (ajută) spre sistemul de operare (operating system oriented architecture).

2.8 ARHITECTURI CISC ȘI RISC

Aproape toate concepțele care se găsesc în microprocesoarele actuale erau cunoscute, în domeniul calculatoarelor, până în anii '70, tehnologia microprocesoarelor nu a făcut decât să le implementeze având ca suport tehnologia de integrare. Dezvoltarea calculatoarelor/microprocesorelor de tip von Neumann a fost una evolutivă și nu una revolutivă. Se pare că "discontinuitatea/ruptura" în această evoluție este provocată de apariția procesoarelor de tip muticore, care atrage după sine impunerea abordării procesării paralele. În domeniul calculatoarelor până în prezent problema a fost *cât hard și cât soft în sistem*. Aceasta, de fapt, reflectă în ce constă diferența între cele două arhitecturi **CISC (Complex Instruction Set Computer)**, **RISC (Reduced Instruction Set Computer)**, adică balansul dintre hard și soft



- Principii prezente într-o arhitectură de tip RISC ("less mean more").
1. Set de instrucții redus (ca număr) și simplu
 - instrucții rapide pentru store, load, add (cele mai frecvente)
 - instrucțiunile complexe se implementeză numai dacă sporul de viteză în execuție justifică creșterea de complexitate
 - număr redus de moduri de adresare (în general două, relativ la PC, indexat)
 - un singur format de instrucțiuțe (sau cel mult două), cu câmpuri fixe ceea ce atrage o decodificare simplă și rapidă (în paralel cu decodificarea OPCODE)
 - operațiile mai complexe se sintetizează în soft prin succesiuni de instrucții simple
 - execuția instrucții într-un singur ciclu (tact de ceas); instrucțiunile pe mai multe cicluri (aritmetice, virgulă flotantă) sunt direcționate spre coprocesoare specializate
 2. Unitate de control simplă, implementată cablat
 3. Execuție de tip registru-la-registru (arhitectură load-store, GPR- General Purpose Registers)
 4. Execuție de tip pipeline
 5. Sinergismul arhitectură-compilator.

Exemplu de ISA pentru un microprocesor RISC implementat în GaAs (anii '80, pentru SDI-Space Defence Initiative)

Single-precision integer add	ADD,SRC1,SOURCE2*,DEST	Load word indexed	LDW,SOURCE2 SRC1],DEST
Single-precision integer add with carry	ADDC,SRC1,SOURCE2,DEST	Load halfword indexed	LDH,SOURCE2 SRC1],DEST
Single-precision integer subtract	SUB,SRC1,SOURCE2,DEST	Load byte indexed	LDB,SOURCE2 SRC1],DEST
Single-precision integer subtract with carry	SUBC,SRC1,SOURCE2,DEST	Store word indexed	STW,SRC2,SYMBOL SRC1]
Single-precision integer subtract reverse	SUBR,SRC1,SOURCE2,DEST	Store halfword indexed	STH,SRC2,SYMBOL SRC1]
Single-precision integer multiply step	MSTEP,SRC1,SRC2,DEST	Store byte indexed	STB,SRC2,SYMBOL SRC1]
Logical AND	AND,SRC1,SOURCE2,DEST	Store word direct	STW,SRC2,LITERAL
Logical inclusive OR	OR,SRC1,SOURCE2,DEST	Store halfword direct	STH,SRC2,LITERAL
Logical exclusive OR	XOR,SRC1,SOURCE2,DEST	Store byte direct	STB,SRC2,LITERAL
Shift right arithmetic by one	SRA,SOURCE2,DEST	Trap conditional	TRAPCND,COND,TRAP_CODE
Shift right logical by one	SRL,SOURCE2,DEST	Jump indexed	JMP,SOURCE2 SRC1]
Move register or immediate	MOV,SOURCE2,DEST	Call indexed	CAL,SOURCE2 SRC1]
Implicit source move	MOV,IMPL_SRC,DEST	Return from trap	RETT,LITERAL,DEST
Implicit destination move	MOV,SOURCE2,IMPL_DEST	Branch conditional	BRC,COND,SOURCE2]
Load high immediate	LDHI,H0_BITS,DEST		

(a)

*SOURCE2 is either a sign extended literal or the register indicated by src2.

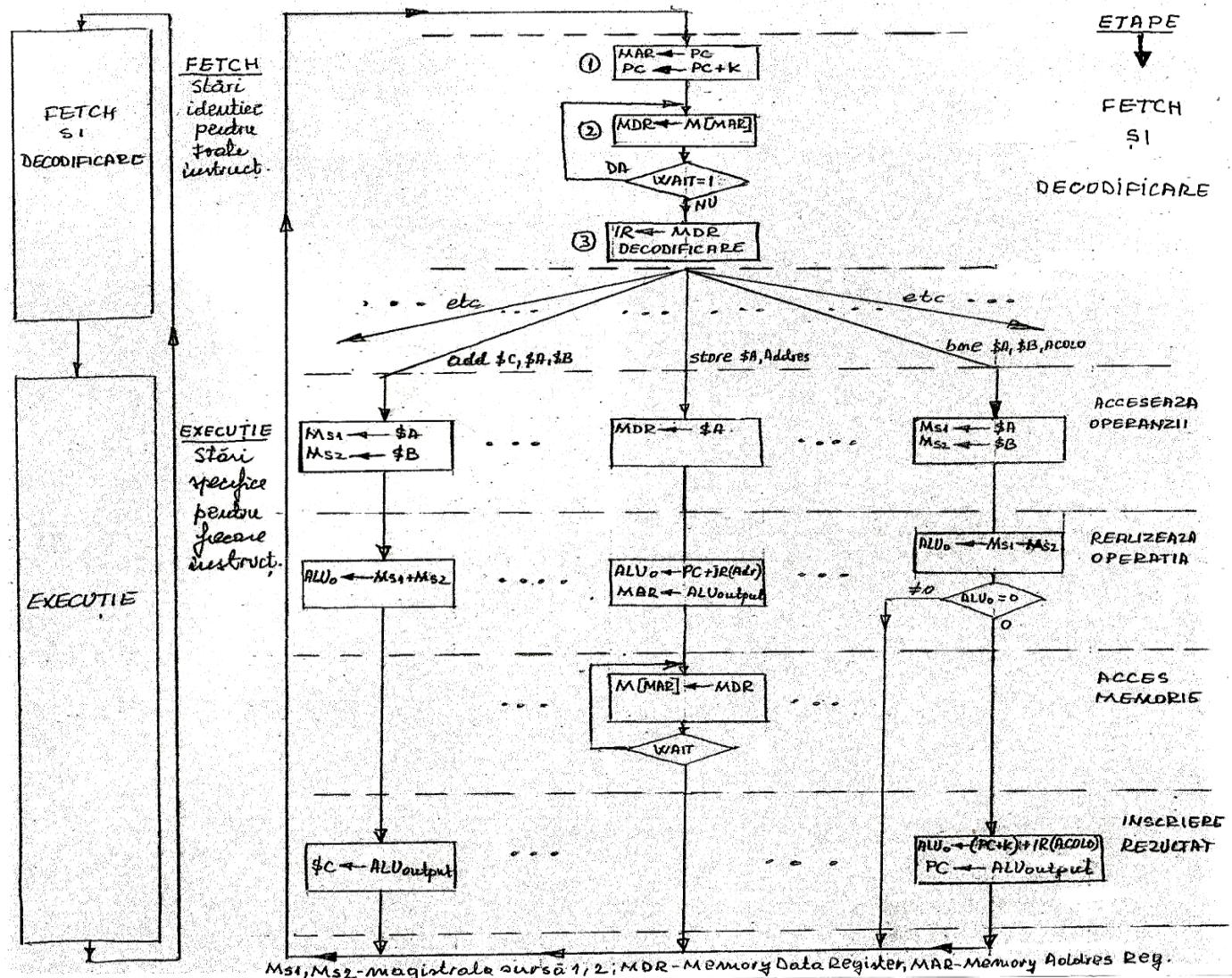
μP-RISC = " Orice mașină după 1985"

CAP 3. CALEA DE DATE

3.1 ORGANIZAREA DE PRINCIPIU A CĂII DE DATE

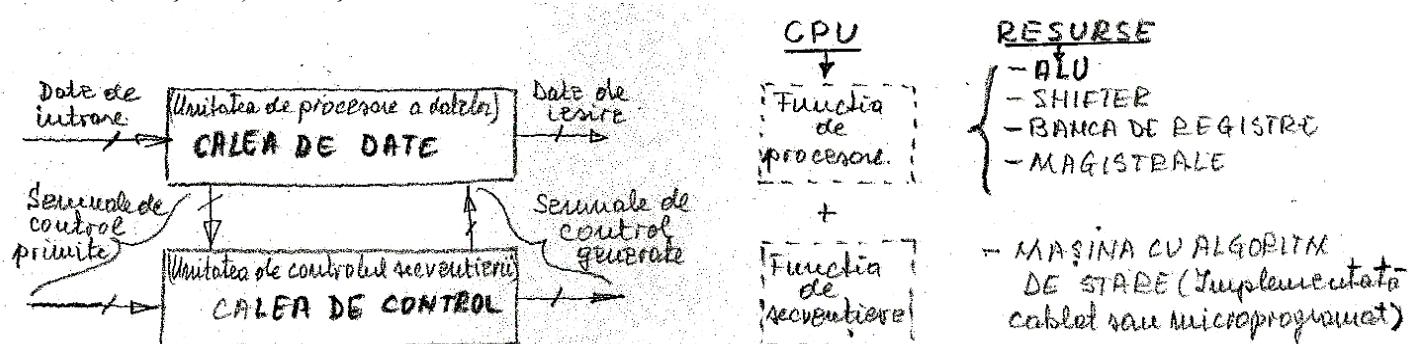
Intepretarea unei instrucțiuni de către procesor, pe durata unui ciclu instrucțiune, cuprinde două subcicluri/cicluri: unul de adresarea și extragerea instrucțiunii din memorie, aducerea acesteia în registrul de instrucțiuni (IR) din procesor și decodificarea sa (**subciclu FETCH**), apoi celălalt, **subciclu EXECUȚIE**, care procesează conform codului operației din instrucțiunea respectivă. După consumarea ciclului instrucțiune (Fetch+Execuție) se inițiază ciclul instrucțiune pentru următoarea instrucțiune. Fiecare subciclu este realizat pe parcursul a una sau mai multe etape/faze, fiecare etapă consumând mai multe tacte de ceas (stări). Subciclu Fetch este identic pentru toate instrucțiunile din ISA, pe când subciclu Execuție diferă de la instrucțiune la instrucțiune.

Intepretarea unei instrucțiuni se reprezintă grafic printr-o diagramă de stare. În această diagramă de stare există un singur traseu pentru subciclu Fetch, care consumă una sau mai multe stări (tacte de ceas) în schimb, după DECODIFICARE, pentru subciclu Execuție există mai multe trasee fiecare desfășurat pe multe etape, iar o etapă consumând mai multe stări (tacte). Aceste etape din subciclu Execuție pot fi: citirea operanzilor (din registre sau din memorie), efectuarea/execuția operației conform OPCODE, citirea/înscrierea operanzilor în memorie, înscrierea rezultatului la (registru) destinație.

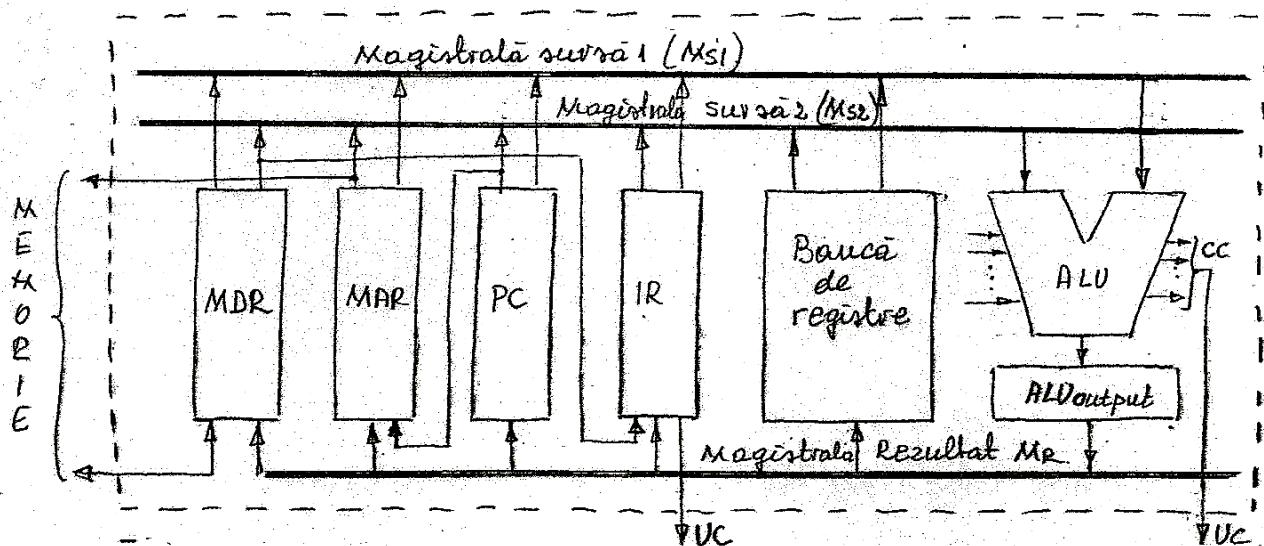


Pentru asigurarea succesiunii etapelor în interpretarea instrucțiunilor, trebuie realizată o funcție de secvențiere și funcție de execuție. Aceste două funcții determină o delimitare în organizarea procesorului într-o unitate de control (CALEA DE CONTROL) și una de execuție (CALEA DE DATE). Prin această delimitare se ușurează atât proiectarea procesorului cât și implementarea sa. Fiecăre din cele două unități au semnale de intrare și de ieșire precum și semnale de interacțiune între ele. Calea de date execută operația specificată de codul instrucțiunii (după decodificare). Realizarea secvențierii tuturor etapelor în interpretarea instrucțiunii, începând cu extragerea din memorie, execuția și până la completare (înscrierea rezultatului) este dirijată de unitatea de control prin generarea de semnale de control. În concluzie interpretarea instrucțiunii cuprinde:

1. (funcția de) secvențiere - realizată în CALEA DE CONTROL;
2. (funcția de) execuție - realizată în CALEA DE DATE.



- Structurarea de principiu pentru calea de date (cu trei magistrale: magistrala sursă1(Ms1), magistrala sursă(Ms2), magistrala rezultat, Mr)

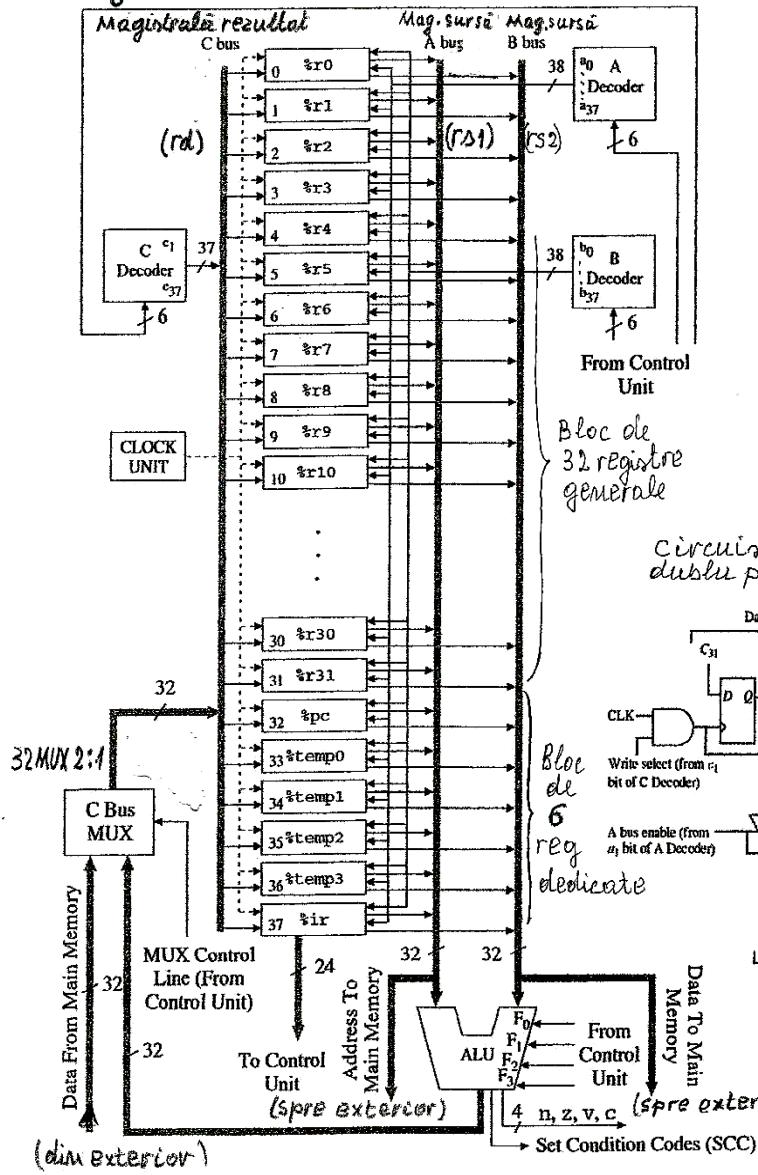


Memory Address Registre (MAR) și Memory Data Registre (MDR) sunt registre buffer care acomodează "interiorul" procesorului cu exteriorul (memoria), primul pentru cuvântul Adresă trimis către memorie, iar al doilea pentru cuvântul Data trimise și primite către și de la memorie. Aceste registre nu au funcție logică ci numai funcție electrică de adaptare cu exteriorul. Uneori, Registrul de Instrucțiuni (IR) și Program Counter (PC) sunt considerate ca fiind componente din calea de control și nu din calea de date.

EXEMPLUL 3.1. [5] Se va prezenta o organizarea de calea de date pentru un microprocesor și setul (redus) de instrucțiuni.

– Calea de date este organizată pe baza a trei magistrale fiecare de 32 biți: Abus – magistrala sursă1 (pe care se vehiculează registrul sursă1, rs1); Bbus – magistrala sursă2 (pe care se vehiculează registrul sursă2, rs2); Cbus – magistrala rezultat (pe care se vehiculează registrul destinație, rd).

Organizarea căii de date

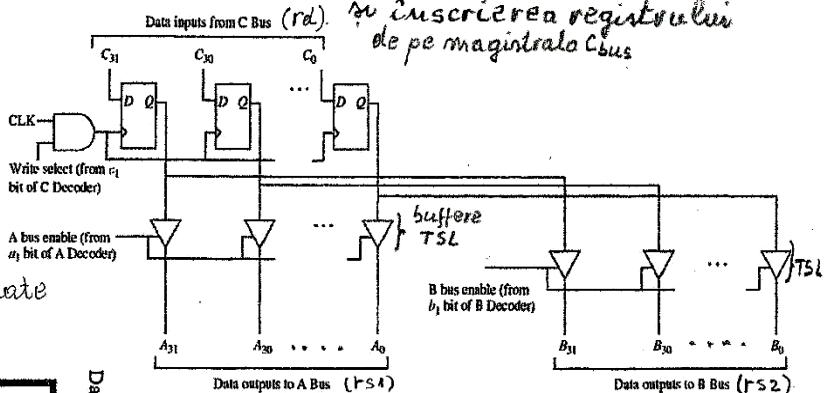


coolerile aplicate la ALU pentru realizarea instrucțiunilor ANDCC și ORCC

F_3	F_2	F_1	F_0	$Carry$ In	a_i	b_i	z_i	$Carry$ Out
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	0	0
0	0	0	0	0	1	1	1	0
0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	1	0	0
0	0	0	0	1	1	0	0	0
0	0	0	0	1	1	1	1	0
0	0	0	1	0	0	0	0	0
0	0	0	1	0	0	1	1	0
0	0	0	1	0	1	0	1	0
0	0	0	1	0	1	1	1	0
0	0	0	1	1	0	0	0	0
0	0	0	1	1	0	1	1	0
0	0	0	1

Circuitistica pentru selectarea unui regisztr
dublu port la cele două magistrale (Abus, Bbus)

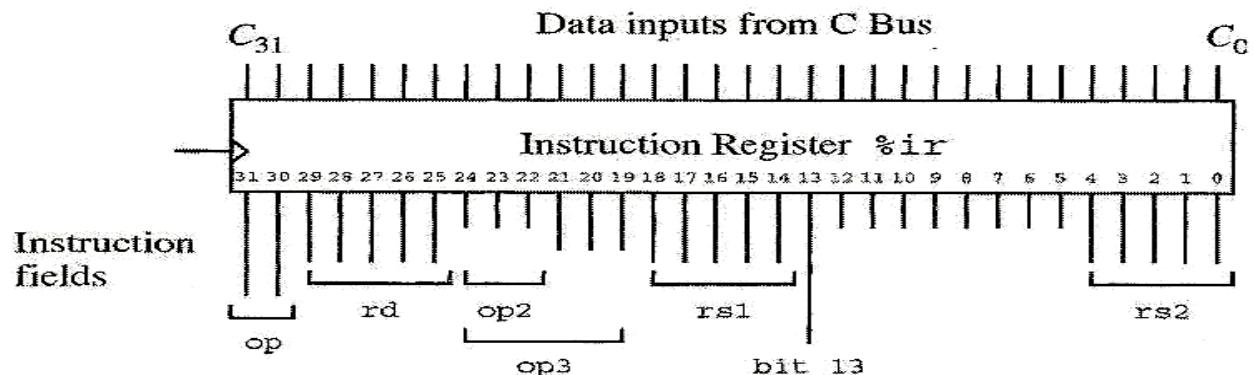
*Su înscrisarea reședinței
de pe magistrata Cibus*



Registrele interne sunt în număr de 37 din care:

- Banca de 32 registre generale de lucru, %r0 – %r31, dublu port, vizibile pentru programator, %r0 (nu este legat la Cbus, deci poate fi numai citit, este cablat la zero pentru a se citi numai valoarea zero); Selectarea unui registru pentru înscriere se realizează prin conjuncția dintre semnalul de la ieșirea decodificatorului C-decoder (la intrarea căruia s-a aplicat numărul registrului (șase biți) în care se face înscrierea) cu semnalul de ceas, iar cuvântul care se înscrise se aplică pe magistrale C bus. Cuvântul care se înscrise poate fi obținut ca un rezultat de la ALU sau este adus din memorie, selectarea între cele două surse se face cu multiplexorul C bus MUX (32MUX 2:1). Citirea unui registru dublu port se obține prin aplicarea ieșirii sale pe A bus sau pe B bus prin comanda bufferul de ieșire TSL, comanda bufferului se obține prin conjuncția dintre semnalul de la decodificatorul A-decoder sau B-decoder cu semnalul de ceas. Cuvintele de pe magistrale A bus sau pe B bus pot fi aplicate direct la ALU sau la memorie externă.
 - Registrul program counter, pc (%r32), are ultimii doi biți cablați la zero pentru a se accesa în memorie doar adrese multiplu de patru, adică (ADRESA)modulo 4 = 0 (cuvânt în memorie de 32 biți, patru bytes).
 - Registrul de instrucționi, ir (%r37), în care se depune instrucționea la sfârșitul ciclului Fetch are următoarele câmpuri, reprezentate în figura următoare:
 - op – codul operației
 - op2, op3 – extensii pentru codul operației
 - rs1, rs1, rd – respectiv registrele sursă unu și doi și destinație

bit13 – (i) specifică existențe unui imediat, imm, în corpul instrucțiunii



- Patru registre temporare, %temp₀ (33) – %temp₃ (36), utilizabile doar de către procesor în execuția instrucțiunilor ca registre temporare.

Unitatea aritmetică și logică, ALU, la care se aplică doi operanzi de pe cele două magistrale sursă, A bus și pe B bus și generează un rezultat. Rezultatul obținut, prin selectarea multiplexorul C bus MUX se aplică pe magistrala C bus și se înscrive în unul din cele 32 de registre ale băncii de registre interne. Operația realizată de ALU prin aplicarea semnalelor F_0, F_1, F_2, F_3 , generate în calea de control, corespunde codului instrucțiunii decodificat. ALU la execuția instrucțiunilor aritmetice și logice produce înscrierea umătorilor biți de condiții din registrul (cc):

z – zero (rezultatul operației este valoarea zero);

n – negativ (rezultatul operației este negativ), bitul 31, MSB, din cuvântul rezultat este 1, exprimare în complement de doi;

c – carry bit, există un transport de la MSB (bitul 31, din poziția 32) spre poziția 33 (care nu există în registrul de rezultat) pentru adunare sau există un împrumut înspre MSB (bitul 31) de la bitul din poziția 33 (care nu există în registrul de rezultat) pentru scădere; b

v – depășire (overflow), rezultatul depășește capacitatea de 32 de biți.

Setul de instrucții. Toate instrucțiunile sunt cu lungimea de 32 biți. Arhitectura este de tipul load-store și este inspirată după procesorul SPARC. Setul redus de instrucții pentru acest procesor este redat în figura următoare (sufixul cc la mnemonicul unei instrucțiuni indică faptul că instrucțiunea înscrie în registrul codurilor de condiții).

SETUL DE INSTRUCȚIUNI

	Mnemonic	Meaning
Memory	ld	Load a register from memory
	st	Store a register into memory
Logic	sethi	Load the 22 most significant bits of a register
	andcc	Bitwise logical AND (<i>înserie cc</i>)
	orcc	Bitwise logical OR (<i>înserie cc</i>)
	orncc	Bitwise logical NOR (<i>înserie cc</i>)
Arithmetic	srl	Shift right (logical)
	addec	Add (<i>înserie cc</i>)
	call	Call subroutine (<i>PC ← Subroutine, %P15 ← PC</i>)
	jmpl	Jump and link (return from subroutine call; <i>jmpl %R15+4, %R0</i>)
Control	be	Branch if equal
	bneg	Branch if negative
	bcs	Branch on carry
	bvs	Branch on overflow
	ba	Branch always

FORMATUL SI CODIFICAREA INSTRUCȚIUNILOR

op (OPCODE)

SETHI Format	31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
	0 0 rd op2 imm22
Branch Format	31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
	0 0 0 cond op2 disp22
CALL format	31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
	0 1 disp30

i

Arithmetic Formats	31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
	1 0 rd op3 rs1 0 0 0 0 0 0 0 0 rs2
	1 0 rd op3 rs1 1 simml3

simml3 - imedial
cu semnal.

sign extended

Calculul adresei din memorie:

$$rd \leftarrow M[rs1 + i] \quad \xrightarrow{\text{adresa}}$$

Memory Formats

$$rd \leftarrow M[rs1 + simm] \quad \xrightarrow{\text{adresa}}$$

CĂMPURILE OPERATION CODE (OPCODE) PENTRU EXTENSIE

op	Format	op2	Inst.	op3 (op=10)	op3 (op=11)	cond	branch
00	SETHI/Branch	010	branch	010000 addcc	000000 ld	0001	be
01	CALL	100	sethi	010001 andcc	000100 st	0101	bcs
10	Arithmetic	100	orcc	010010 orcc	001001 ldsb	0110	bneg
11	Memory	100	orncc	010110 orncc	001010 ldsh	0111	bvs
				100110 srl	000001 ldub	1000	ba
				111000 jmpl	000010 lduh		
					000101 stb		
					000110 sth		

PSR (Rezervul de
stare al procesorului)

PSR
(cc)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
n z v c pil t

condiții inserire de ALU

Pentru rezolvarea exceptiilor întrerupențială

enable trap

Processor Interrupt Level

3.2 REPREZENTAREA NUMERELOR ÎN CALCULATOR- OVERFLOW (DEPĂŞIRE)

Un număr format din n biți este reprezentat în calculator pe un cuvânt, $x_{n-1} x_{n-2} x_{n-3} \dots x_2 x_1 x_0$, având ca suport de implementare un registru sau o locație de memorie cu lungime de n biți. Pe cei n biți se pot reprezenta maximum 2^n valori de numere; uneori se concatenează două registre obținându-se reprezentarea în dublă precizie, ceea ce mărește posibilitatea de reprezentare la 2^{2n} valori de numere.

Numerelor întregi reprezentate în calculator pot fi numere pozitive, referite ca numere fără semn (unsigned number, unsigned int) și numere cu semn (signed int.). Procesorul MIPS are pentru anumite operații instrucțiuni speciale, acestea au în mnemonic litere u pentru numere fără semn, dar aceeași instrucțiune fără sufîxul u este pentru operații cu semn.

Reprezentarea în calculator a numerelor întregi cu semn, în general, se face sub formă codificată în complement de doi $[x]_2$. Un număr cu semn în complement de doi se reprezintă pe un registru/locație cu bitul cel mai semnificativ, MSB, cu valoarea zero dacă este pozitiv ($x_{n-1} = 0$) și același bit are valoare 1 dacă numărul este negativ ($x_{n-1} = 1$), în felul următor:

$$\begin{aligned} \text{pt } x > 0 \quad [x]_2 = 0 x_{n-2} x_{n-3} \dots x_2 x_1 x_0 \rightarrow x = 0 \cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i \\ \text{pt } x < 0 \quad [x]_2 = 1 x_{n-2} x_{n-3} \dots x_2 x_1 x_0 \rightarrow x = (-1) \cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i \end{aligned}$$

valoare numărului fiind în intervalul $(-2^{n-1}) \leq x \leq (2^{n-1} - 1)$,

pentru $n = 32 \rightarrow [-2^{31}, 2^{31} - 1]$, iar pentru $n = 8 \rightarrow [-128, 127]$

Cu un cuvânt de n biți, în complement de doi, se pot reprezenta 2^n numere: 2^{n-1} numere negative, $2^{n-1} - 1$ numere pozitive plus cifra zero (un sir de zerouri). Uneori valoarea numerelor reprezentate în calculator se scalează cu valoarea absolută cea mai mare, 2^{n-1} , astfel că toate numerele se aduc în intervalul $[-1, (1 - 2^{1-n})]$

$$-1 \leq \frac{x}{2^{n-1}} \leq (1 - 2^{1-n})$$

Astfel, toate numerele reprezentate se aduc în intervalul $[-1, (1 - 2^{1-n})]$.

EXEMPLU 3.2. Fie numerele conținute în registrele \$s0 și \$s1

$$\$s0 = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111 \rightarrow [-1]_2 \text{ sau fără semn este } 4\ 249\ 967\ 293$$

$$\$s1 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001 \rightarrow 1$$

Aplicând instrucțiunile slt, sltu, care este rezultatul procesării?

Soluție. Aplicând instrucțiunea relațională pentru numere cu semn

$$\text{slt } \$to \$s0 \$s1 ; \$to \leftarrow 1 \text{ deoarece } [-1]_2 < 1$$

iar aplicând instrucțiunea pentru numere fără semn (u)

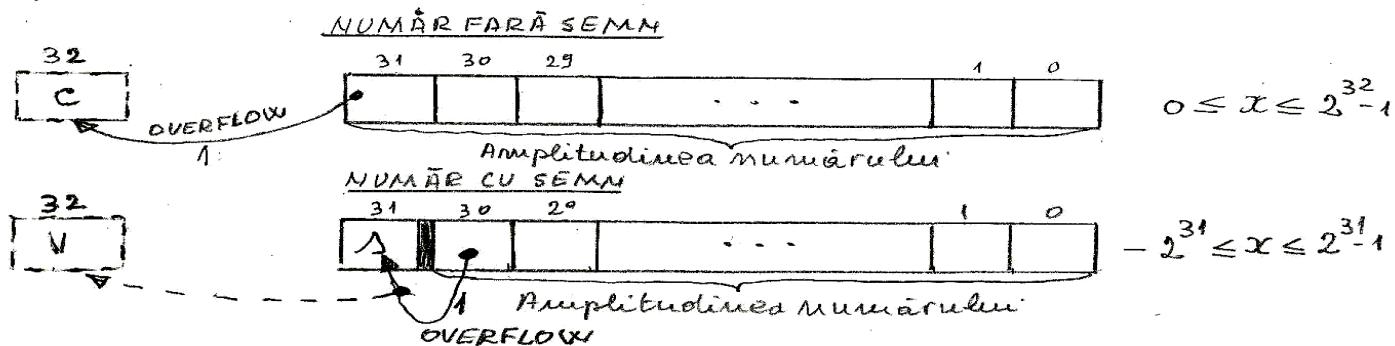
$$\text{sltu } \$to \$s0 \$s1 ; \$to \leftarrow 0 \text{ deoarece } 4\ 249\ 967\ 293 > 1$$

- **DEPĂŞIREA (OVERFLOW).** În urma unor operații aritmetice poate apărea operația de depășire, adică numărul rezultat nu mai poate fi reprezentat pe cei n biți care sunt disponibili într-un registru/locație. Depășirea se manifestă, de exemplu pentru un cuvânt de 32 biți, $x_{31}x_{30}x_{29} \dots x_2x_1x_0$, printr-un transfer 1 de la bitul din poziția 32 (x_{31}) la bitul din poziția 33 care de fapt nu există pentru numerele fără semn, iar depășirea pentru numerele cu semn este un transfer 1 de la bitul din poziția 31 (x_{30}) la bitul de semn s, din poziția 32, (x_{31}).

La apariția unei depășiri, pe arhitecturile actuale soluțiile pot fi:

- cazul 1. se generează EXCEPTIE atât pentru numerele cu semn cât și pentru cele fără semn;
- cazul 2. se înscrie un bit **C** (CARRY) pentru numerele fără semn și se înscrie un bit **V** pentru numerele cu semn (acești biți, C și V, sunt în afara lungimii registrului, de fapt ar fi în poziția 33);

- cazul 3. nu se semnalează în nici un fel, problema trebuie rezolvată de către programator (după fiecare instrucțiune care poate genera depășire, dacă aceea depășire contează).



Cazul 1 și 2

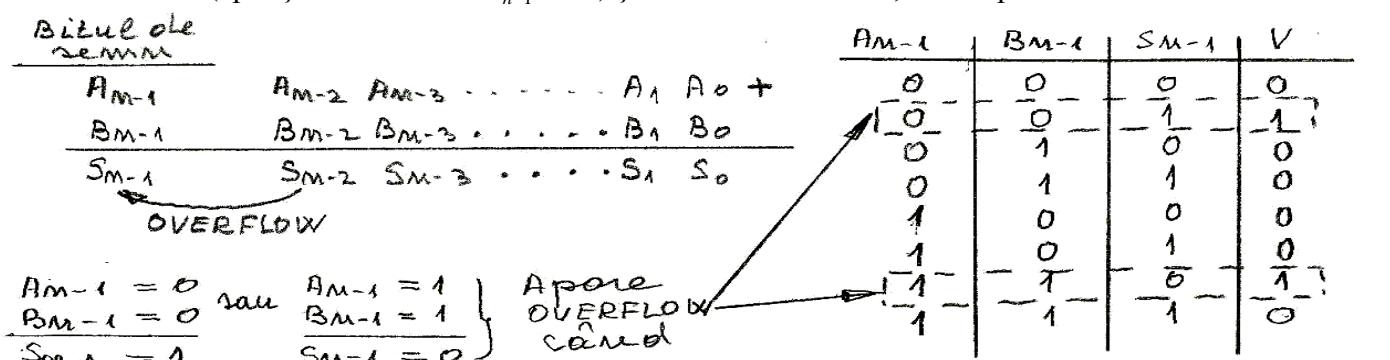
Multe din arhitecturi implementează combinat soluțiile 1 și 2 prin introducerea unui bit **EB** (Enable Bit). Când se programează prin sistemul de operare EB = 1, atunci la apariția depășirii procesorul va genera un eveniment de EXCEPTIE (care în fond se reduce la apelarea unei subroutines), iar dacă procesorul se programează cu EB = 0 atunci la apariția depășirii se va înscrie bitul C dacă numerele sunt fără semn pozitive sau se va înscrie bitul V dacă operanții sunt tratați ca numere cu semn. Acest mod de abordare este reflectat în tabelul următor pentru unele din procesoarele cunoscute

PRIN SISTEMUL DE OPERARE SE PROGRAMEAZĂ EB:

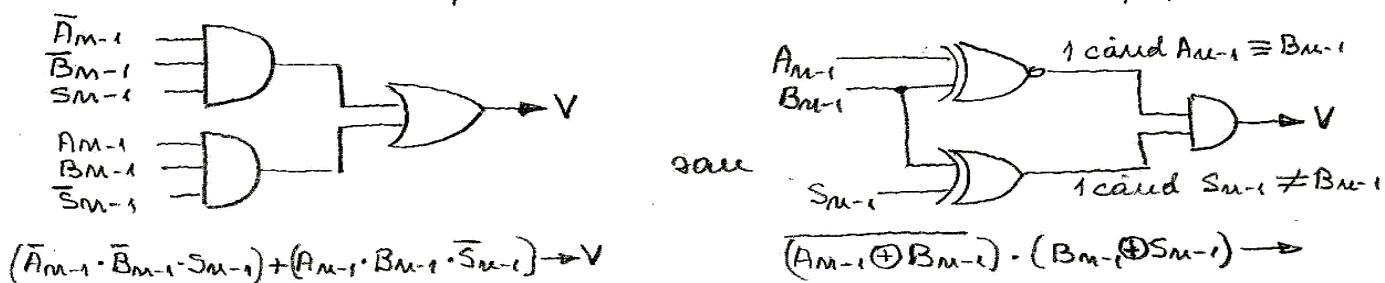
PROCESORUL	GENERAREA EXCEPTIE PENTRU:		V, pentru numere cu semn	C, pentru numere fără semn	EB=0
	NUMERE CU SEMN	PENTRU NUMERE FĂRĂ SEMN			
VAX	DA	NU	Hu se generează EXCEPTIE pentru că numerele fără semn sunt utilizate în calculul de adiere	DA, Add înserie V=1	DA, Add înserie C=1
IBM 370	DA	NU	nu se generează EXCEPTIE pentru că numerele fără semn sunt utilizate în calculul de adiere	DA, Add înserie bitul din reg. cc	DA, Add înserie bitul din reg. cc
INTEL 8086	DA	NU	nu se generează EXCEPTIE pentru că numerele fără semn sunt utilizate în calculul de adiere	DA, Add înserie V=1	DA, Add înserie C=1
SPARC	NU	NU	nu se generează EXCEPTIE pentru că numerele fără semn sunt utilizate în calculul de adiere	Addcc înserie V=1 Add înserie V=0	Addcc înserie C=1 Add nu înserie

- Modalitate de detectare (prin hard) a depășirii la numerele:

- cu semn (apariția transferului $S_{n-1} \leftarrow 1$) și înscrierea bitului V, este explicată în continuare:

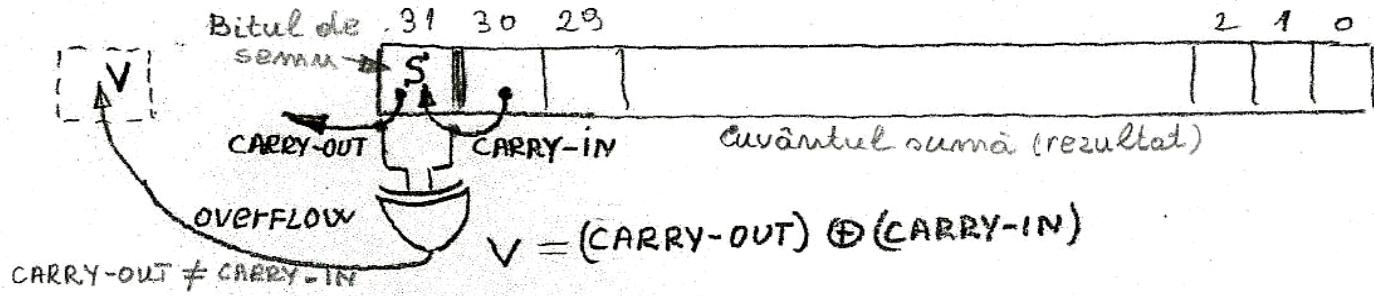


Pentru care corespund următoarele două implementări



La adunarea a două numere cu semn dacă cele două numere au același semn iar bitului de semn rezultă de semn opus evident că a fost a fost o depășire, ceea ce corespunde liniei 2 și 7 din tabelul de adevăr din figura anterioară, deci se poate deduce funcția logică pentru bitul V și implementa circuitul combinațional de detecție.

O altă modalitate de implementare a circuitului detector se bazează pe faptul că totdeauna există overflow când transferul de la bitul de semn S_{n-1} spre exterior (carry-out) este diferit de transferul de la MSB (S_{n-2}) la bitul de semn S_{n-1} , ceea ce se poate implementa ușor cu o poartă SAU EXCLUSIV ca în figura următoare.



- fără semn suportul este foarte simplu: transferul de la bitul 31 (poziția 32) se înscrie ca fanion de Carry în poziția 33 (din afara registrului).

Cazul 3. Detectarea apariției depășirii la numerele cu semn, deoarece nu există un mecanism hard, trebuie realizată în soft de către programator, deoarece instrucțiunile nu generează Excepție și nici nu înscrie biții de CARRY (C) sau de overflow (V), ca la cazurile 1 și 2. Programatorul din analiza semnului celor doi operanzi și a semnului operandului rezultat poate deduce apariția de overflow, conform celor patru situații expuse în tabelul următor

OPERATIЯ	SEMNUЛ OPERAНZILOR Am-1 Bm-1	SEMNUЛ RE ZVLTATOLUI Sm-1	
A D U H	$\begin{cases} A + B \\ A + B \end{cases}$	+	Adunarea a două numere de semne contrare nu poate genera OVERFLOW
A R E	$\begin{cases} A + B \\ A + B \end{cases}$	—	Adunarea a două numere de același semn poate genera OVERFLOW
S C A D E Q E	$\begin{cases} A - B \\ A - B \\ A - B \\ A - B \end{cases}$	$\begin{cases} + \\ — \\ + \\ — \end{cases}$	$\begin{cases} Scăderea a două numere de același semn nu poate genera OVERFLOW \\ Scăderea a două numere de semne contrare poate genera OVERFLOW \end{cases}$
			OVERFLOW

Din acest tabel rezultă că adunarea a două numere de semne opuse și scăderea numerelor de același semn nu poate genera overflow. În consecință, trebuie testat doar semnul rezultatului adunării a două numere de același semn, dacă are semn opus față de cel al celor doi operanzi indică overflow. La scădere poate apărea overflow numai dacă dacă cei doi operanzi au semne contrare (aceasta se reduce, de fapt, tot la adunarea de operanzi de același semn).

- Cum se rezolvă depășirea la MIPS? Există două tipuri de instrucțiuni:
 - instrucțiunile utilizate pentru numerele cu semn (add, addi, sub, subi) care la producerea depășirii generează un eveniment de Excepție (controlul este preluat de sistemul de operare, care printr-o subrutină specifică rezolvă depășirea și apoi se face reîntoarcerea la instrucțiunea care a generat depășirea).
 - Instrucțiuni utilizate pentru numere fără semn (addu, addiu, subu) care nu generează un eveniment de Excepție dacă s-a produs o depășire, este sarcina programatorului să se asigure că nu s-a produs depășirea! Aceste instrucțiuni sunt folosite pentru calculul de adrese de acces la memorie care nu sunt

numere cu semn. Dacă totuși aceste instrucțiuni sunt utilizate pentru numere cu semn, neapărat trebuie să se determine (prin soft, conform tabelului anterior) dacă s-a produs overflow.

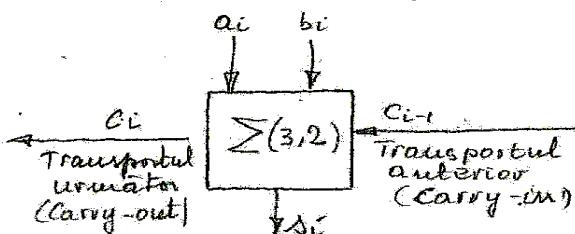
3.3. UNITATEA ARITMETICĂ ȘI LOGICĂ, ALU (ARITHMETIC AND LOGIC UNIT)

3.3.1. Sumatoare.

3.3.1.1. Sumatorul cu Transport Progresiv, STP

- Celula sumator complet, $(\sum_{(3,2)})$. Tabelul de adevăr pentru celula sumator complet (cu trei intrări: a_i, b_i și transportul anterior C_{i-1} , generând două ieșiri: suma și și transportul următor C_i) cu reprezentarea, schemă bloc și implementarea cu porți reprezentate în figura următoare

de intrare		de ieșire		intermediare	
a_i	b_i	\bar{c}_{i-1}	s_i	c_i	p_i
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	0	1	0
1	1	0	0	1	1
1	1	1	1	1	1

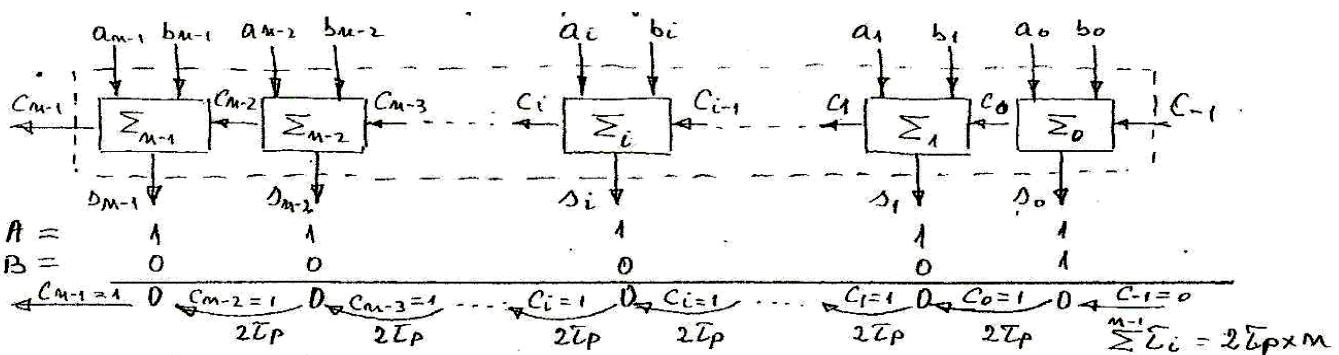
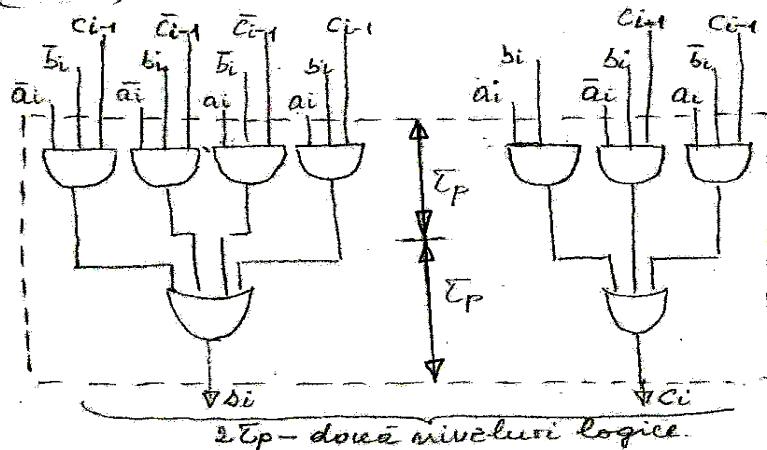
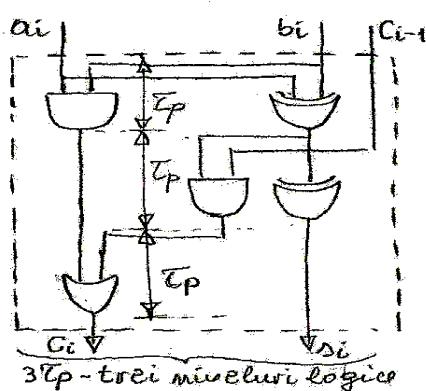


$$s_i = \bar{a}_i \bar{b}_i \bar{c}_{i-1} + \bar{a}_i b_i \bar{c}_{i-1} + a_i \bar{b}_i \bar{c}_{i-1} + a_i b_i c_{i-1} = (a_i \oplus b_i) \oplus c_{i-1}$$

$$C_i = \bar{a}_i b_i c_{i-1} + a_i \bar{b}_i c_{i-1} + a_i b_i \bar{c}_{i-1} + a_i b_i c_{i-1} = a_i b_i + (a_i \oplus b_i) C_{i-1} = a_i b_i + (a_i + b_i) C_{i-1}$$

(pentru că există identitatea: $a_i b_i + (a_i \oplus b_i) C_{i-1} \equiv a_i b_i + (a_i + b_i) C_{i-1}$)

• Implementare



În exemplul din figură pentru sumarea cuvintelor A și B, de n biți, sumarea nu este terminată până când transportul se propagă prin toate cele n celule sumator, deci timpul de sumare $T = n \cdot 2 T_p$.

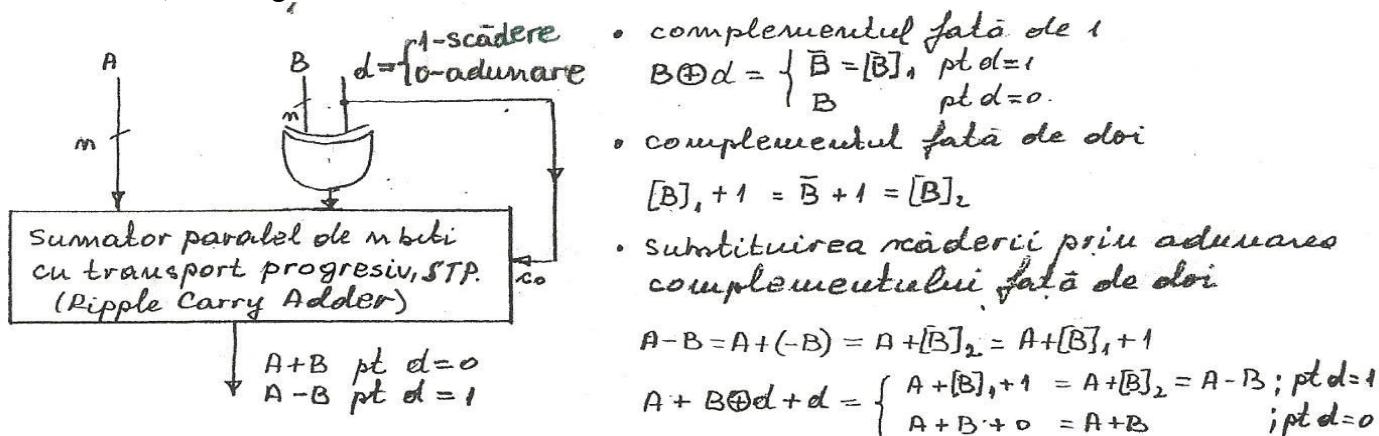
Caracteristicile sumatorului cu transport progresiv:

- structură regulată, ușor de implementat în siliciu (predomină porțile și nu conexiunile);
- suprafața consumată în siliciu O(n);
- timpul de propagare ridicat O(n), de exemplu: pentru $n = 32$, $\tau_p = 10\text{ns}$, timpul de calcul $T = n \cdot 2 \tau_p = 640\text{ns}$, frecvența maximă $f_{max} \leq 1/640\text{ns} = 1,56\text{MHz}$.

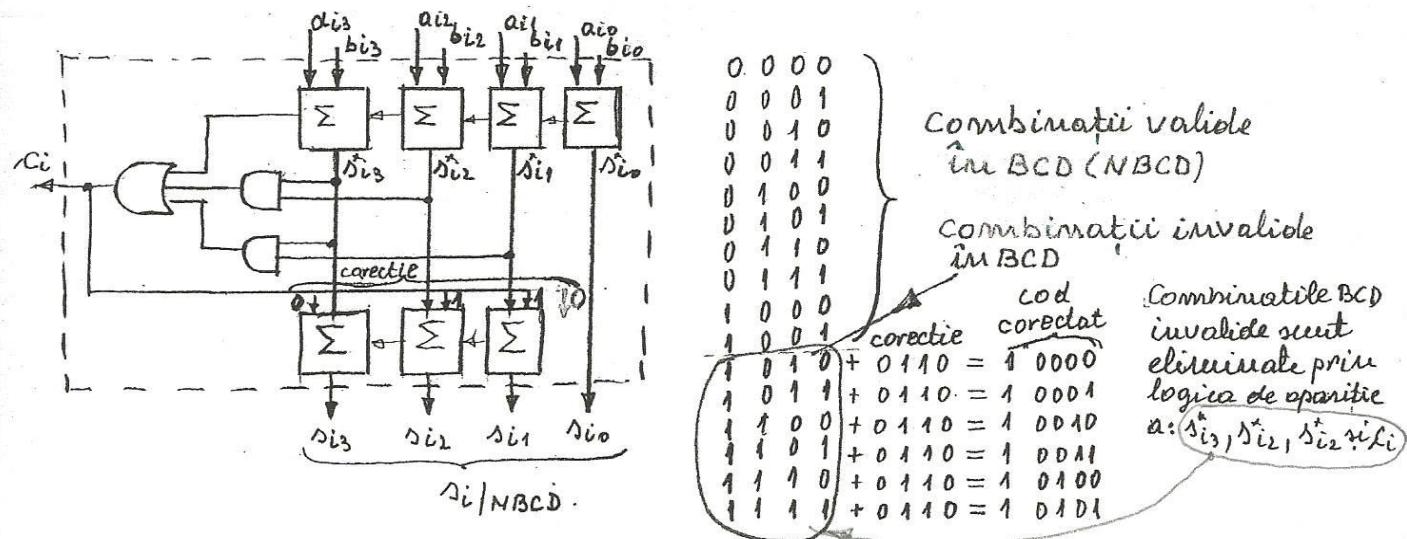
- Sumatorul ($A+B$)/ Scăzătorul ($A-B$). Scăderea a două numere se substituie prin adunarea scăzătorului exprimat în complement de doi

$$A - B = A + (-B) = A + [B]_2$$

deci nu este nevoie să existe pe lângă sumator și un scăzător. Sumatorul, prin intermediul unei variabile de control d va realiza operația de adunare, când $d = 0$, iar pentru $d = 1$ va realiza operația de scădere (prin calculul lui $[B]_2 = \bar{B} + 1$) ca în figura următoare



- Sumator în cod BCD. Sumarea a două cifre zecimale codificate în BCD generează o cifră corectă dacă suma este mai mică decât $9|10$ (1001|BCD), dar dacă suma este mai mare sau egală cu $10|10$ (1010|BCD) atunci rezultatul este incorect (nu mai este un cod BCD) și se corectează prin sumarea corecției 6|10 (0110|BCD), de exemplu $9+8=17$ ($1001 + 1000 = 0001$ (suma) \rightarrow (și transportul) $1|10$, dacă se sumează (corecția) $0110 + 0001 = 0111 \rightarrow 7|10$. Aplicarea corecției este necesară fie când suma rezultată este mai mare de $15|10$ (1111), deci apare un transfer, fie când suma rezultată este între $10|10$ și $15|10$ care se poate identifica prin faptul că doi biți dintre cei trei mai semnificativi au simultan valoarea 1.



3.3.1.2. Sumatorul cu transport anticipat, CLA (Carry-Look-Ahead adder)

Timpul mare $O(n)$ de sumare la sumatorul cu transport progresiv se datorează timpului mare pentru propagarea transportului prin cele n celule. Sumatorul cu transport anticipat reduce timpul de sumare la o valoare constantă $O(1)$.

În tabelul de adevăr pentru celula sumator complet ($\Sigma_{(3,2)}$) s-au introdus și două variabile intremediare g_i și p_i , definite în modul următor:

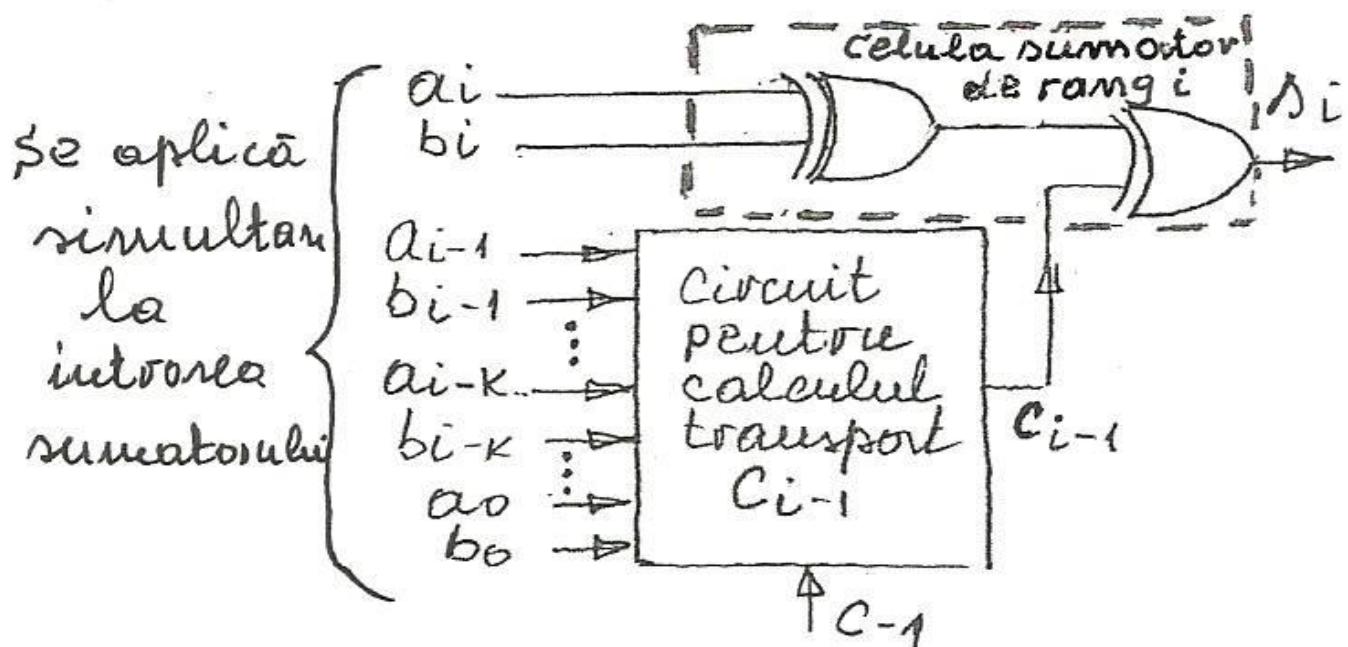
- funcția de generare, g_i , care are valoarea 1 când celula i generează un transport următor și care se exprimă prin relația logică, $g_i = a_i \cdot b_i$, deci implementare pe o poartă AND;
- funcția de propagare, p_i , care are valoarea 1 când prin celula i se propagă transport anterior și care se exprimă prin relația logică, $p_i = a_i + b_i$, deci implementare pe o poartă OR.

Cu ajutorul variabilelor intremediare, g_i , p_i , expresiile deduse anterior pentru C_i și s_i pot avea următoare exprimare

$$\begin{aligned} p_i &= a_i + b_i \\ s_i &= a_i \oplus b_i \oplus C_{i-1} \Rightarrow g_i = a_i \cdot b_i \\ C_i &= a_i \cdot b_i + (a_i + b_i)C_{i-1} \quad s_i = a_i \oplus b_i \oplus C_{i-1} \\ &\qquad\qquad\qquad C_i = g_i + p_i \cdot C_{i-1} \end{aligned}$$

Celula de ordin i va genera transport următor C_i fie când $g_i = 1$, fie când $p_i = 1$ și $C_{i-1} = 1$

Pe baza expresiei $C_i = g_i + p_i \cdot C_{i-1}$ printr-o dezvoltare recurrentă pentru $i = 1, 2, \dots, n-1$ se obține expresia logică care calculează transportul anterior C_{i-1} pentru fiecare celulă, deci fiecare celulă își poate calcula suma și imediat ce se aplică biții de intrare a_i și b_i fără să mai aștepte ca transportul anterior să străbată lanțul de la C_1 până la intrarea sa. La toate celulele sumatorului se aplică simultan, la început, cele doi operanzi A , B iar transportul anterior la intrarea, C_{i-1} , se poate calcula pentru celula sumatoare de rang i ($=0, 1, 2, \dots, n-1$) numai pe baza biților anteriori: $a_{i-1}, b_{i-1}; a_{i-2}, b_{i-2}; \dots; a_1, b_1; a_0, b_0$ și a bitului de intrare C_{i-1} doar pe două niveluri logice. În acest mod de abordare celula sumator are structura prezentată în figura următoare, pe lângă calculul sumei, $s_i = a_i \oplus b_i \oplus C_{i-1}$, pe două porți SAU EXCLUSIV, se mai adaugă circuitul care generează transportul anterior C_{i-1} .



$$\begin{cases} S_0 = a_0 \oplus b_0 \oplus C_{-1}, \\ C_0 = g_0 + p_0 C_{-1} \end{cases}$$

$$\begin{cases} S_1 = a_1 \oplus b_1 \oplus C_0 \\ C_1 = g_1 + p_1 C_0 = g_1 + g_0 \cdot p_1 + p_1 p_2 C_{-1} \end{cases}$$

$$\begin{cases} S_2 = a_2 \oplus b_2 \oplus C_1 \\ C_2 = g_2 + p_2 C_1 = g_2 + g_1 p_2 + g_0 p_2 p_1 + p_2 p_1 p_0 C_{-1} \end{cases}$$

$$\begin{cases} S_3 = a_3 \oplus b_3 \oplus C_2 \\ C_3 = g_3 + p_3 C_2 = g_3 + g_2 p_3 + g_1 p_3 p_2 + g_0 p_3 p_2 p_1 + p_3 p_2 p_1 p_0 C_{-1} \end{cases}$$

$$\begin{cases} S_i = a_i \oplus b_i \oplus C_{i-1} \\ C_i = g_i + p_i C_{i-1} = g_i + g_{i-1} p_i + g_{i-2} p_i p_{i-1} + g_{i-3} p_i p_{i-1} p_{i-2} + \dots + p_i p_{i-1} \dots p_1 p_0 C_{-1} \end{cases}$$

$$\begin{cases} S_{n-1} = a_{n-1} \oplus b_{n-1} \oplus C_{n-2} \\ C_{n-1} = g_{n-1} + p_{n-1} C_{n-2} = g_{n-1} + g_{n-2} p_{n-1} + g_{n-3} p_{n-1} p_{n-2} + \dots + g_i p_{n-1} p_{n-2} \dots p_{i+2} p_{i+1} + \dots \end{cases}$$

$$a_3 - a_0 = A$$

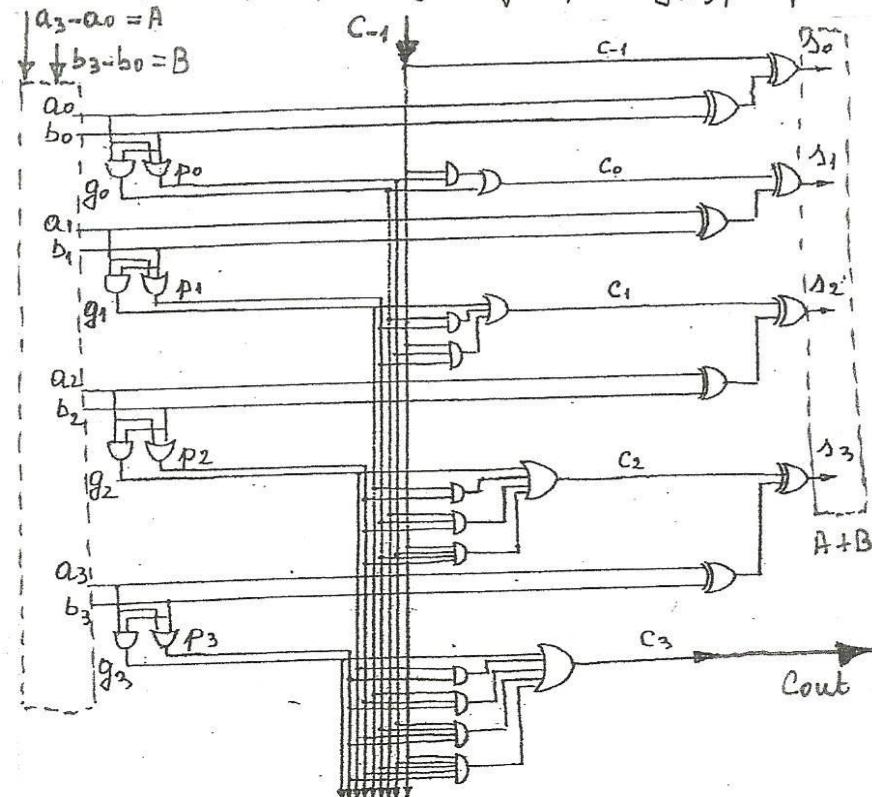
$$b_3 - b_0 = B$$

$$a_0, b_0$$

$$a_1, b_1$$

$$a_2, b_2$$

$$a_3, b_3$$



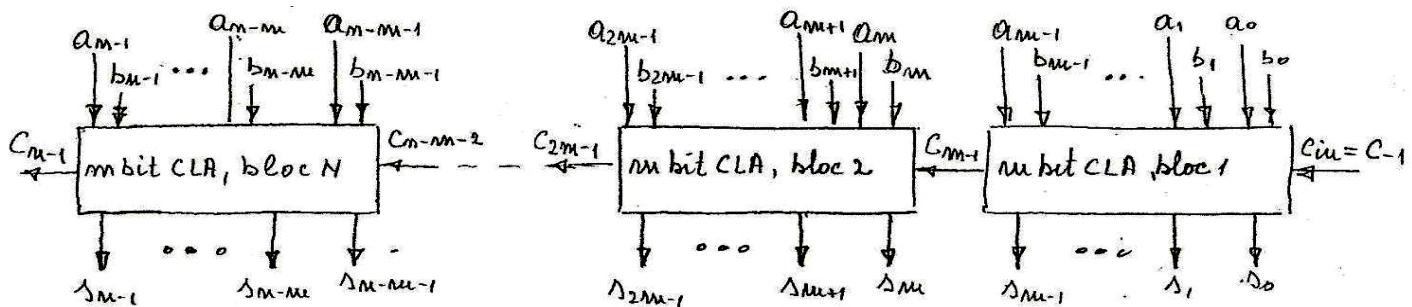
Caracteristici:

- Tiup de propagare
tranzistor: $4\tau_p$; $O(1)$

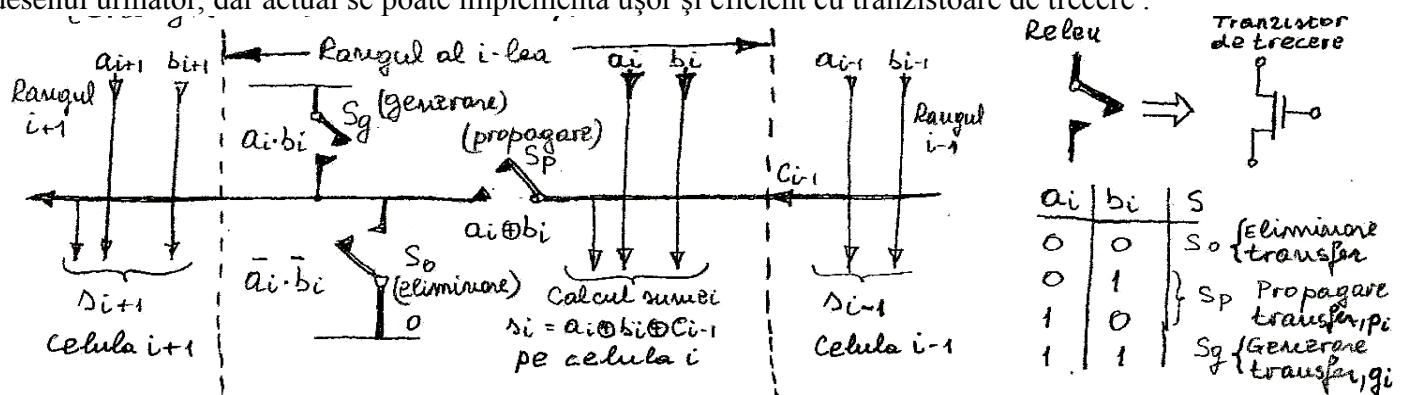
- Necesitatea de a asigura valori mai mari pentru fan-in și fan-out și face practic neutilizabil pentru $n > 8$ (conține porti AND și OR cu $(n+1)$ intrari!!!)
- Pentru lay-out proiectă o rețea ne-regulată (predominanță conexiunile

Sumator cu transport progresiv (compus) din blocuri cu transport anticipat (Ripple- Block Carry Look-Ahead Adder, RCLA).

Deoarece un sumator cu transport anticipat, CLA, pentru un număr n mare de ranguri de sumare este practic greu de realizat, dar se pot utiliza astfel de sumatoare cu un număr mai mic de biți (m), iar aceste sumatoare sunt inseriate sub forma unui sumator cu propagarea transportului, pentru un sumator de n biți fiind necesară inserierea a n/m sumatoare CLA. În funcție de numărul sumatoarelor CLA utilizate și de numărul de biți m, pentru implementarea unui sumator de n biți se pot obține structuri care realizează timp de sumare cuprins între cel al unui al unui sumator complet CLA de n biți și cel al unui sumator cu transport progresiv de n biți. O organizare de principiu pentru un astfel de numărător combinat este prezentată în figura următoare (cu N=n/m blocuri CLA fiecare de m biți).



- Circuitul/lanțul Manchester pentru propagarea transportului (Manchester chain, Kilburn adder). Structura de lanț Manchester a fost concepută în perioada de început a calculatoarelor pentru a crea, în paralel cu celulele sumatorului, o cale rapidă de transfer pentru transport. Inițial a fost implementat cu relee, cum este figurat în desenul următor, dar actual se poate implementa ușor și eficient cu tranzistoare de trecere.



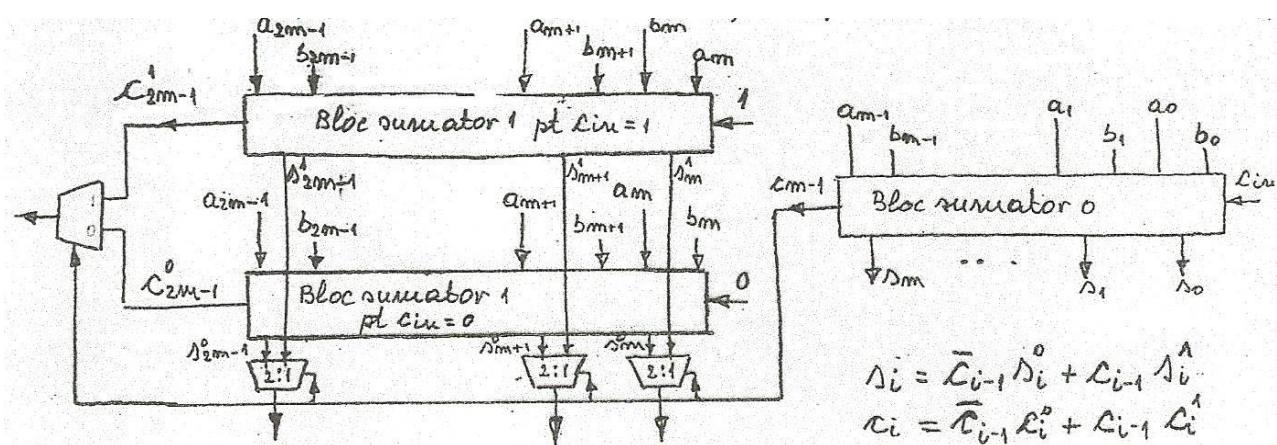
$$C_i = S_g + S_p \cdot C_{i-1}$$

$$S_i = a_i \oplus b_i \oplus C_{i-1}$$

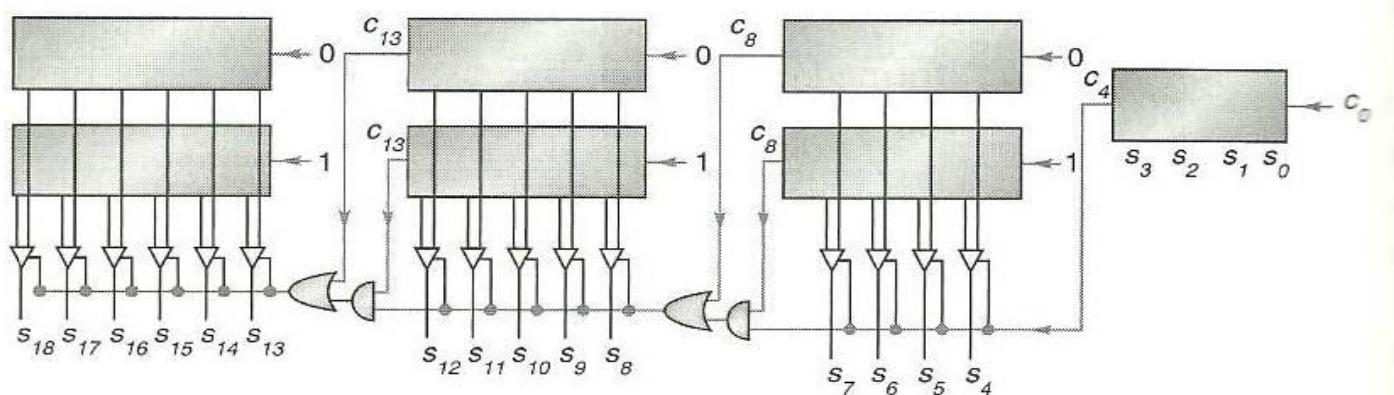
$$S_i = a_i \oplus b_i \oplus C_{i-1}$$

3.3.1.3 Sumatorul cu selectarea transportului, CSA (Carry Select Adder)

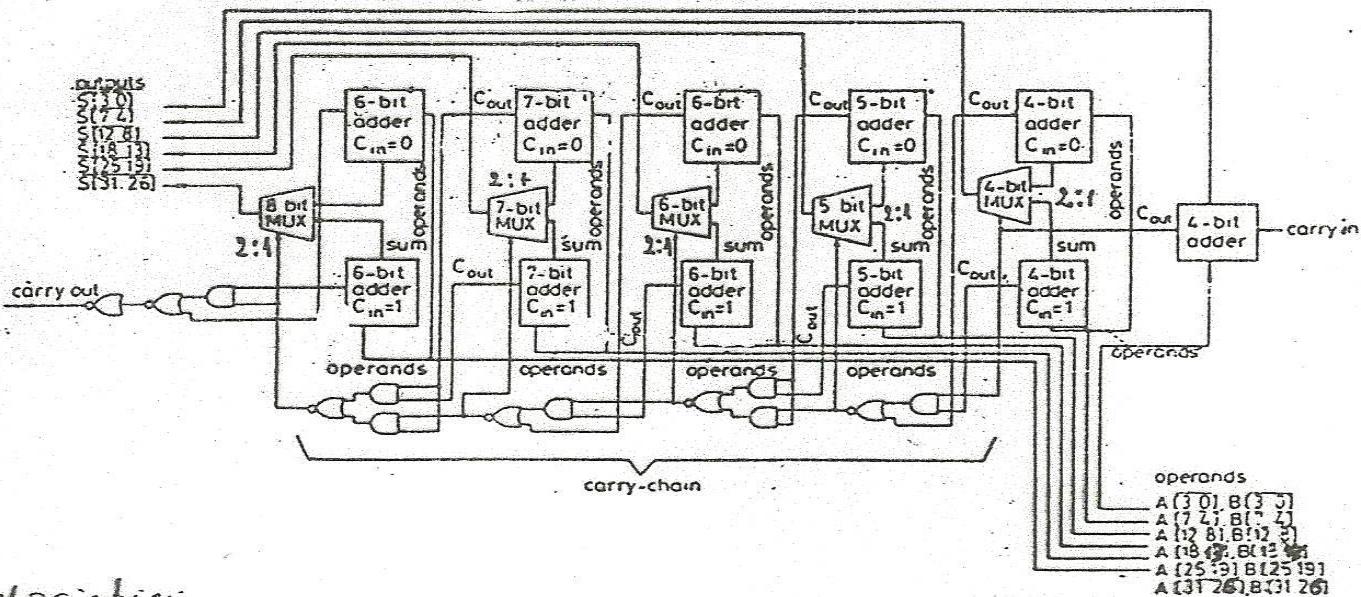
Funcționarea sumatorului cu selectarea transportului se bazează pe asertivitatea: sumarea completă pe o celulă sau pe un modul sumator se poate efectua doar când este disponibil transportul anterior, C_{in} , care poate avea numai două valori: 1 sau 0. În concluzie, se poate porni procesul de sumare pe respectiva celulă sau modul realizând simultan sumarea atât pentru transportul anterior $C_{in} = 1$ cât și pentru $C_{in} = 0$, fără a se mai aștepta sosirea acestuia, iar când acesta sosete, din cele două sumări, deja efectuate, se selectează doar suma care s-a calculat pentru transportul anterior care are valoarea egală cu cea a transportului anterior care a sosit, cum se prezintă în figura următoare. Când semnalul C_{in} sosete acesta este utilizat pentru selectarea unui grup de multiplexoare (2:1) care alege sumarea corectă cât și pentru selectarea unui multiplexor (2:1) care selectează transportul următor corect.



Un sumator CSA de n biți se structurează din module de m biți, câte două astfel module pentru fiecare subgrup de m biți din cuvintele de sumare, cu excepția celui mai puțin semnificativ subgrup de m biți pentru care este utilizat doar un singur modul, cum este prezentat în figura următoare. Biții cuvintelor de sumare se aplică simultan pe toate modulele de m biți ai sumatorului. Pe fiecare modul de m biți, cu excepția modulului cel mai puțin semnificativ, se calculează atât suma și transportul următor cu $C_{i-1} = 1$ cât și cu $C_{i-1} = 0$, iar când sosete transportul anterior, calculat de modulul anterior, se alege suma și transportul următor calculate corespunzător valorii bitului C_{i-1} sosit.



Structurarea sumatorului nu se face în module de un număr egal de biți ci un număr crescător. Această neuniformitate se datorează faptului că la o aplicare simultană a biților cuvintelor de sumat, când semnalul de (transport) selectare a sosit de la modulul anterior acesta a trebuit să se propage și printr-un multiplexor, deci cel puțin două nivele logice de parcurs în plus, în consecință modulul de selectat poate avea cu cel puțin o celulă (rang) de sumare în plus față de modulul anterior.



Caracteristicile sumatorului CSA sunt:

- blocurile componente pot avea structuri de : transport progresiv, transport anticipat etc;
- sunt recomandate organizările cu blocuri inegale, blocul următor fiind cu un rang mai lung ;
- fun-out pentru semnalul de selectare al multiplexoarelor poate deveni de valoare ridicată ;
- timpul de sumare este $O(\sqrt{n})$.

3.3.2. Multiplicatorul - pentru numere fără semn (Multiply unsigned - multu)

Înmulțirea a două numere, $B \times A$, fără semn

$$A = a_{n-1} a_{n-2} \dots a_2 a_1 a_0$$

Inmultitor (Multiplier)

$$B = b_{n-1} b_{n-2} \dots b_2 b_1 b_0$$

Deinmultitor (Multiplicand)

se poate realiza : 1. cu deplasarea deînmulțitorului spre stânga ; 2. deplasarea produselor parțiale spre dreapta

1. Multiplicarea cu deplasarea deînmulțitorului spre stânga

$$B = 11_{10} \rightarrow 1011_2 ; A = 13_{10} \rightarrow 1101_2$$

$$\begin{array}{r}
 1011 \times 1101 \\
 1011 \\
 0000 \\
 1011 \\
 1011 \\
 \hline
 10001111_2 \rightarrow 143_{10}
 \end{array}$$

2. Multiplicarea cu deplasarea produselor parțiale spre dreapta (deînmulțitorul rămâne în aceeași poziție), operația de deplasare dreapta cu o poziție se poate realiza ușor pe un tact.

$$\begin{array}{r}
 \begin{array}{r} 1011 \times 1101 \\ \hline \end{array}
 \end{array}$$

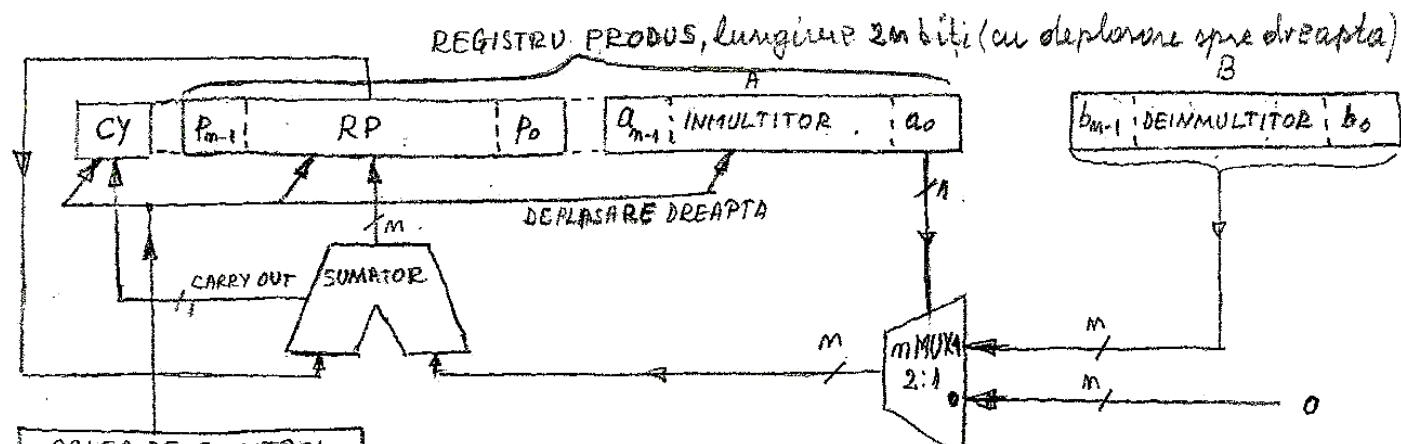
Add { $\begin{array}{r} 0000 \\ 1011 \end{array}$ } Prinul produs parțial P_0
 Shift { $\begin{array}{r} 1011 \\ 01011 \end{array}$ } $P_0 \times 2^{-1}$ (deplasare dreapta cu un rang)
 Add { $\begin{array}{r} 0000 \\ 01011 \end{array}$ } Al doilea produs parțial P_1
 Shift { $\begin{array}{r} 001011 \\ 1011 \end{array}$ } $P_1 \times 2^{-1}$ (deplasare dreapta cu un rang)
 Add { $\begin{array}{r} 001011 \\ 110111 \end{array}$ } Al treilea produs parțial P_2
 Shift { $\begin{array}{r} 0110111 \\ 1011 \end{array}$ } $P_2 \times 2^{-1}$ (deplasare dreapta cu un rang)
 Add { $\begin{array}{r} 0001111 \\ 1011 \end{array}$ } Al patrulea produs parțial P_3
 Cy = 1 $P_3 \times 2^{-1}$ (deplasare dreapta cu un rang)
 Shift { $\begin{array}{r} 10001111 \\ 1011 \end{array}$ } Produsul final $10001111_2 = 143_{10}$

Adunarea și deplasarea spre dreapta pot fi realizate în două tacte de ceas. Aceste două operații, adunare + deplasare pot constitui o primitivă

$$\begin{aligned}
 P_i &\leftarrow P_{i-1} + B \cdot a_i & B \cdot a_i \\
 P_{i+1} &\leftarrow P_i \cdot 2^{-1}
 \end{aligned}$$

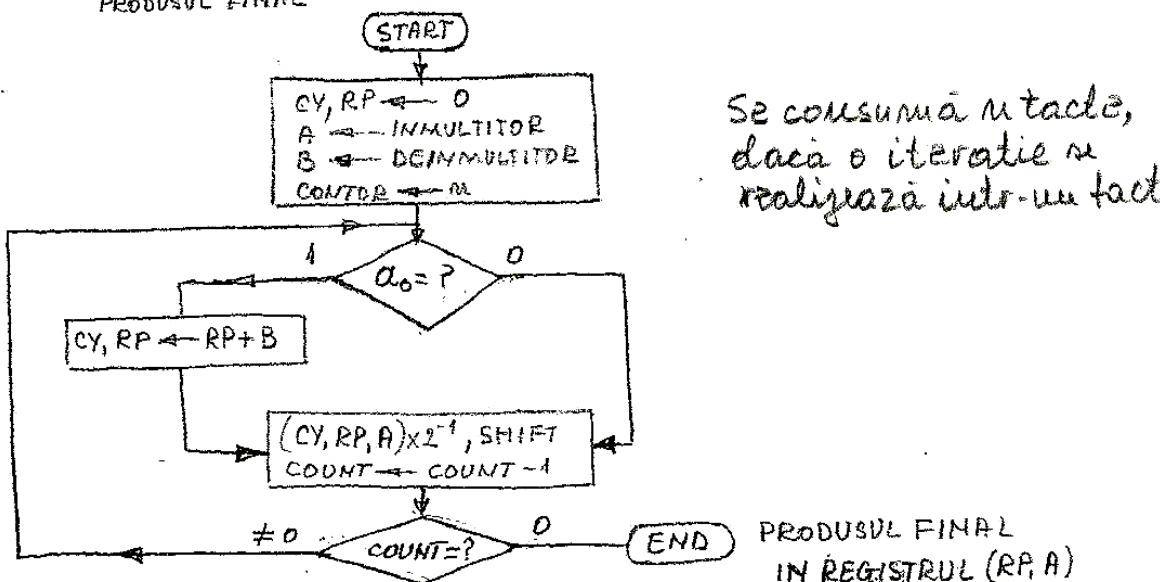
care poate fi integrată sub forma unei instrucțiuni AND and SHIFT în ISA, în consecință realizarea unei multiplicări în soft a două cuvinte de n biți se realizează prin aplicarea repetitivă, de n ori, a instrucțiunii AND and SHIFT.

Structurarea hard a unui circuit multiplicator, care realizează multiplicarea prin regula sumare și deplasare spre dreapta a produselor parțiale, conform algoritmului prezentat anterior, este prezentată în figura următoare. Deînmulțitorul B este înscris în registrul B iar înmulțitorul în registrul A , produsele parțiale P_i se obțin în registrul RP , fiecare dintre aceste registre având lungimea de n biți, iar bitul de transport (CARRY) se obține în registrul de un bit CY , plasat înaintea registrului RP . Registrele $CY + RP + A$ prin concatenare formează un registru de deplasare, în care după înscrierea produsului parțial P_i în registrul RP (ca rezultat de la sumator) și a bitului de transport în registrul CY se realizează o deplasare spre dreapta cu o poziție (2^{-1}), cel mai puțin semnificativ (LSB) din registrul A (înmulțitor) se pierde. Prin selectarea multiplexoarelor $n \times MUX$ 2:1, cu bitul LSB din registrul A , se realizează pe sumator, fie sumarea cu B , fie sumarea cu 0 după cum LSB este 1 sau 0, adică se aplică produsul $B \cdot a_i$. Sumatorul realizează adunarea operandului $B \cdot a_i$, care este ieșirea de la multiplexor, cu operandul $P_{i-1} \cdot 2^{-1}$, ca ieșire din registrul RP , și generează conținutul în registrul RP și în CY dacă există transfer. Registrul de deplasare, $RP + A$, are lungimea de $2n$ biți, deci are capacitatea ca produsul final cu lungimea de $2n$ biți să încapă. Multiplicarea cuvintelor de n biți necesită n pași, număr care se înscrise într-un contor, odată cu înscrierea registrelor A și B , care apoi se decrementeză după fiecare pas, ajungând la zero după n pași (pe un pas se realizează o adunare și o deplasare spre dreapta).



C	Reg. RP	Reg. A	Reg. B	CONTOR	VALORI ÎNITIALE
0	0 0 0 0	1 1 0 1	1 0 1 1	4	ADD } PRIMUL PAS
0	1 0 1 1	1 1 0 1	1 0 1 1	3	SHIFT }
0	0 1 0 1	1 1 1 0	1 0 1 1	2	ADD } AL DOILEA PAS
0	0 1 0 1	1 1 1 0	1 0 1 1	1	SHIFT }
0	0 0 1 0	1 1 1 1	1 0 1 1	0	ADD } ALTREILEA PAS
0	1 1 0 1	1 1 1 1	1 0 1 1		SHIFT }
0	0 1 1 0	1 1 1 1	1 0 1 1		AL PATRULĂ PAS
1	0 0 0 1	1 1 1 1	1 0 1 1		
0	1 0 0 0	1 1 1 1	1 0 1 1		

PRODUSUL FINAL



Există și alte structuri de multiplicare succesivă care prin diferite modalități pot duce la micșorarea numărului de pași de execuție pentru multiplicarea a doi operanzi cu lungimea de n biți. De asemenea, pentru lungimi mai reduse de operanzi se poate realiza multiplicarea nu succesiv ci combinațional, folosind pentru aceasta look-up table.

3.3.3 Unitatea de procesare în virgulă flotantă

3.3.3.1. Reprezentarea în virgulă flotantă

Exprimarea unui număr real N în virgulă flotantă (reprezentarea stiințifică) este

$$N = (-1)^s \cdot M \cdot B^e$$

în care: s – este semnul numărului, care se reprezintă pe un bit și este: 1 pentru negativ și 0 pentru pozitiv;
 M – matisa numărului;
 B – baza sistemului de numerație în care este considerată exprimarea numărului (uzual 2, 10, 16);
 e – exponentul.

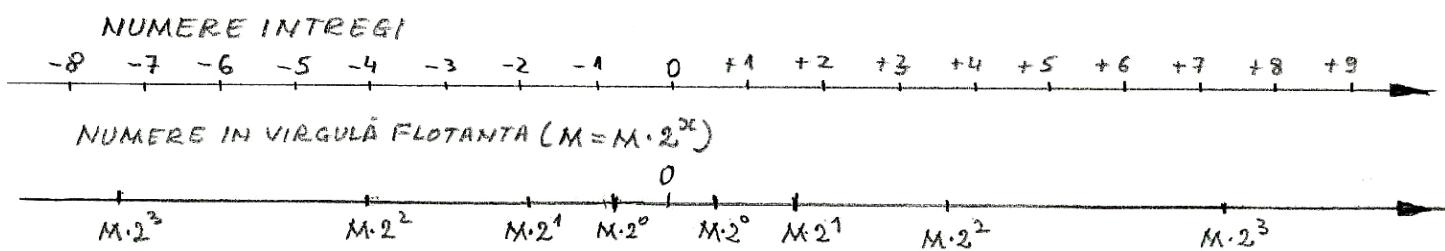
Pentru un număr real N în virgulă flotantă există o infinitate de forme scrise (de unde și sintagma de virgulă flotantă, punctul zecimal (radix point) poate fi fixat ca poziție oriunde în cadrul numărului!) de exemplu, câteva din aceste forme de scris pentru numerele 3.14_{10} și 10.11_2 (uzual baza oricărui sistem de numerație se scrie 10, aici s-a scris 2 (și nu sub forma 10) pentru o mai usoară înțelegere)

$$3.14 \cdot 10^0 = 31.4 \cdot 10^{-1} = 0.0314 \cdot 10^2 = 0.314 \cdot 10^1$$

$$10.11 \cdot 2^0 = 101.1 \cdot 2^{-1} = 1.011 \cdot 2^1 = 0.1011 \cdot 2^2$$

Din această multitudine de forme de scris, uzual, se alege **forma normalizată** care corespunde formei când prima cifră după punctul (virgula) zecimal este o cifră semnificativă (adică nu este zero), ceea ce pentru numerele anterioare corespund: $0.314 \cdot 10^1$, $0.1011 \cdot 2^2$.

Domeniul de adresare și domeniul de reprezentare a numerelor sunt direct determinate de lungimea cuvintelor (registrelor) din procesor. Reprezentarea în virgulă flotantă a unui număr este caracterizată de tripletul (s, M, e) . Deoarece numărul este exprimat în procesor pe un cuvânt cu lungimea (fixă) de n biți, trebuie ca lungimile (în biți) repartizate celor trei parametri să satisfacă relația $n = s + M + e$. Lungimea în biți a mantisei (deci numărul de biți pe care se reprezintă în procesor) determină precizia de exprimare a valorii numărului, iar exponentul (deci numărul de biți pe care se reprezintă în procesor) determină rangul/domeniul de exprimare al numărului. Numărul total de biți alocați pentru exprimarea mantisei și pentru exprimarea exponentului este $n-s$ (s – semnul totdeauna necesită 1 bit). Între numărul de biți repartizați pentru mantisă și numărul de biți repartizat pentru exponent se face un balans, ceea ce se reflectă printr-un balans între precizia și domeniul numărului, cu creșterea preciziei (numărul de biți pentru mantisă) scăderea preciziei crește domeniul. Dar, indiferent căți biți se repartizează pentru exponent și căți biți se repartizează pentru mantisă numărul total de numere în virgulă flotantă care se pot reprezenta pe un cuvânt de n biți este 2^n (la fel ca și pentru numere întregi). Diferența în reprezentarea pe o axă a numerelor constă în faptul că numerele întregi sunt repartizate uniform (distanța absolută dintre două numere consecutive este 1) pe când la cele în virgulă flotantă distanța absolută între două numere consecutive crește cu cât domeniul/rangul (exponentul) în care se situează valoarea numerelor devine mai mare.



De exemplu, în intervalele $(2^1, 2^2)$, $(2^2, 2^3)$ sunt același valori ale mantisei, dar distanța între două valori consecutive ale mantisei din al doilea interval este de două ori mai mare decât între aceleași valori ale mantisei din primul interval.

EXEMPLUL 3.2. Un număr în virgulă flotantă cu trei digiti repartizați pentru mantisă și doi digiti pentru exponent are forma $N = M \cdot 10^e = (m_2 m_1 m_0) \cdot 10^{e_1 e_0}$; rezultă că valorile pentru mantisa (pe trei digiti) se situeză în intervalul $(0.001, 0.999)$, iar valorile pentru exponent (pe doi digiti) în intervalul $(00, 99)$. Să se determine distanța

absolută și relativă între valorile cele mai mari ale mantiselor numerelor (0,998, 0999) la domeniul cel mai mic (00) și la domeniul cel mai mare (99).

La domeniul cel mai mic (10^0) :

$$\text{Diferenta absoluta } 0.999 \cdot 10^0 - 0.998 \cdot 10^0 = 0.001 \cdot 10^0$$

$$\text{Diferenta relativa } \frac{0.001 \cdot 10^0}{0.998 \cdot 10^0} = 0.001$$

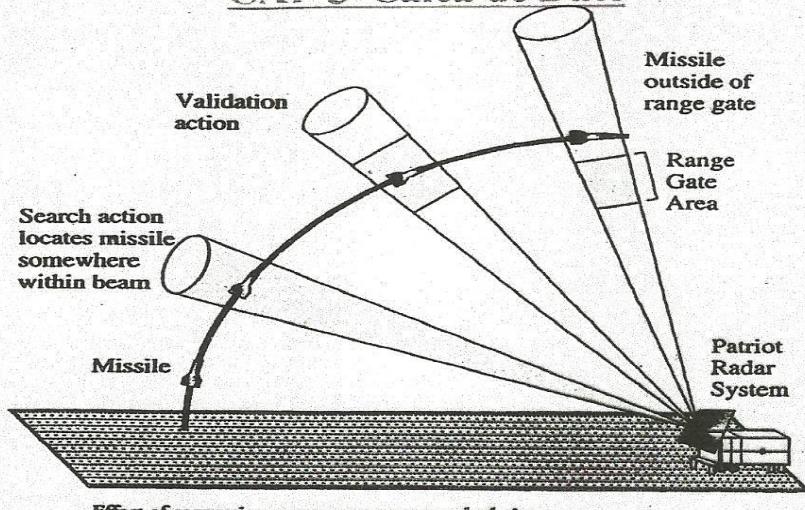
La domeniul cel mai mare (10^{99}) :

$$\text{Diferenta absoluta } 0.999 \cdot 10^{99} - 0.998 \cdot 10^{99} = 0.001 \cdot 10^{99}$$

$$\text{Diferenta relativa } \frac{0.001 \cdot 10^{99}}{0.998 \cdot 10^{99}} = 0.001$$

Diferența absolută depinde de domeniu pe când diferența relativă este aproximativ aceeași indiferent de domeniu

CAP 3 Calea de Date

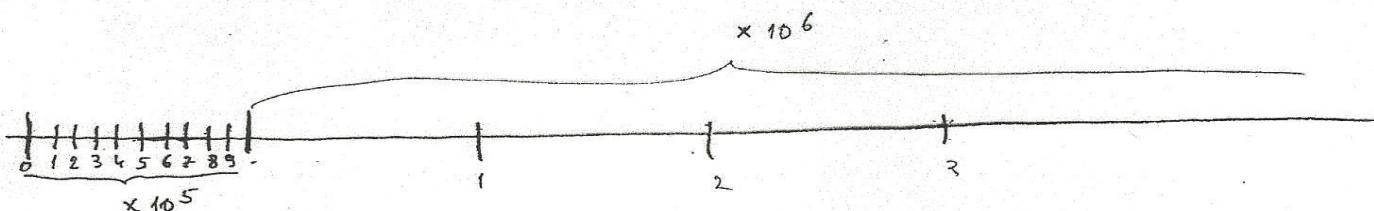


Effect of conversion error on range gate calculation.

- Dhahran, Saudi Arabia, 5th February 1991
- Timpul se măsoară cu un următoare cu 24 biți
cu $T_{clock} = 100 \mu s$, deci rezulta un contineur în următorul (valoarea timpului) în numere întregi, care trebuie aproximat cu cea mai apropiată valoare exprimată ca un număr după puținile lui 10
- $v = \frac{\Delta s}{\Delta t}$ calculat în floating point pe 24 biți
- În 100 ore de funcționare a stației PATRIOT valoare contineur este

$$100 \text{ ore} \times 3600 \text{ sec} \times 10 = 3600000$$

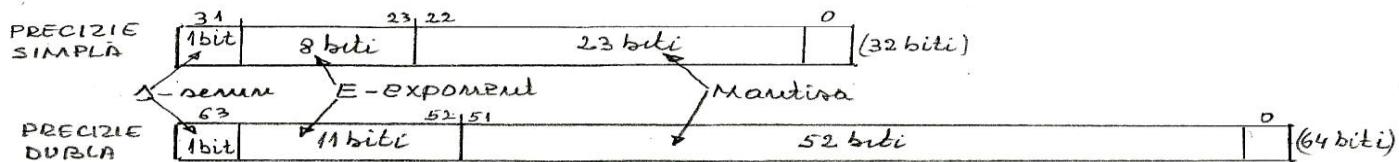
$$4 \text{ ore} \times 3600 \text{ sec} \times 10 = 144000$$
- Eroarea de calcul pentru range gate se face de 687 m



3.3.3.2. Reprezentarea numerelor conform standardului IEEE-754

(Fundamentarea matematică a fost realizată de Prof. W. Kahan, Berkeley University)

- Formatul de cuvânt: simplă și dublă precizie



- Semnul numărului. Pentru semn este rezervat bitul cel mai semnificativ din cuvânt, 0-pentru numere pozitive și 1-pentru numere pozitive.
- Reprezentarea exponentului. Pentru reprezentarea exponentului sunt repartizați 8 biți (23-30), în simplă precizie, și 11 biți (52-62), în dublă precizie, iar codificarea pe acești biți este în mărime și semn la care se aplică apoi o deplasare (cod deplasat). Pentru simplă precizie deplasarea pe codul mărime și semn se face cu +127 rezultând codul deplasat cu 127, sau codul Excess 127.
 - complement de 2: $[2^{M-1}, 2^{-1}] \rightarrow [128, 127]; 0 \rightarrow 0000\ 0000$ (8 reprezentări)
 - mărime și semn: $[2^{M-1}, 2^{-1}] \rightarrow [127, 127]; [-1023, 1023] \xrightarrow{-0} 1000\ 0000$ (2 reprezentări)
 - numere întregi: $[0, 2^{M-1}] \rightarrow [0, 255]; [0, 1023]; 0 \rightarrow 0000\ 0000$ (0 reprezentări)

- Numărul zero. Un număr $N = M \cdot B^e = 0$ când $M = 0$ și/sau $B^e = 0$; pentru $B^e = 0 \rightarrow e = -\infty$ (adică cel mai mic exponent negativ (-127), reprezentat în mărime și semn pe opt biți, este considerat ca $-\infty$). Dar, dacă pentru a introduce o identitate cu reprezentarea numerelor intregi, la care zero este un sir de 32 biți zero, atunci la reprezentarea în virgulă flotantă pe lângă cei 23 de biți ai mantisei de valoare zero, care sunt un sir de zerouri, trebuie ca și cei opt biți ai exponentului în mărime și semn să fie tot un sir de zerouri. Ca numărul (exponentul cel mai mic) -127, în mărime și semn (1111 1111), să fie reprezentat ca un sir de opt zero-uri (0000 0000) trebuie deplasat (sumat) cu 127, adică utilizând un cod deplasat cu 127, rezultând pentru exponentul e reprezentarea E referită cu exponentul deplasat sau Excess 127 ($E = e + 127$).

- Mantisa, M - este parte fracționară a numărului și se aduce totdeauna la forma normalizată: 0.**1**bb...bbbb (23 de biți pentru precizie simplă); 0.**1**bb.....bbbb (52 de biți pentru precizie dublă) cu valori situate în intervalele (valoarea mantisei este considerată totdeauna pozitivă și se exprimă în binar natural)

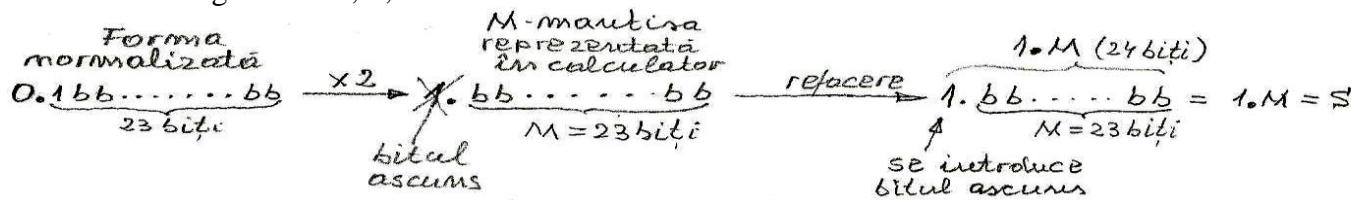
$$0.\underbrace{1000\dots 00}_{23 \text{ biți}} = 0.5_{10} \leq 0.\underbrace{1bbb\dots bb}_M \leq 0.\underbrace{111\dots 11}_{23 \text{ biți}} = (1 - 2^{-23})_{10} \text{ pt precizie simplă}$$

$$0.\underbrace{1000\dots 00}_{23 \text{ biți}} = 0.5_{10} \leq 0.\underbrace{1bbb\dots bb}_M \leq 0.\underbrace{111\dots 11}_{52 \text{ biți}} = (1 - 2^{-52})_{10} \text{ pt precizie dublă}$$

Deoarece la reprezentarea normalizată bitul după punctul zecimal este totdeauna 1 acest bit poate să nu mai fie reprezentat în formatul IEEE-754 (este bitul ascuns, reprezentat implicit, deci se face economie de un bit în reprezentarea mantisei), rezultă că mantisa, de fapt, este de 24 de biți (respectiv pe 53 de biți); în consecință, totdeauna când se extrage mantisa din formatul IEEE754 prima operație în obținerea valorii exacte a mantisei este cea de adăugare a bitului (ascuns, considerat implicit 1) înaintea punctului zecimal, obținându-se significantul, S.

- Significantul, S = 1. M, este o exprimare a valorii mantisei numărului obținută din reprezentarea mantisei în formatul 754 la care se introduce ca parte întreagă bitul ascuns, bit care este totdeauna de valoare 1. Altfel spus, significantul se obține din forma normalizată a numărului (0.1xxx...) deplasată cu o poziție spre stânga (1.xxx...).

Domeniul în care significantul, S, are valori este:



Domeniul în care significantul, S, are valori este:

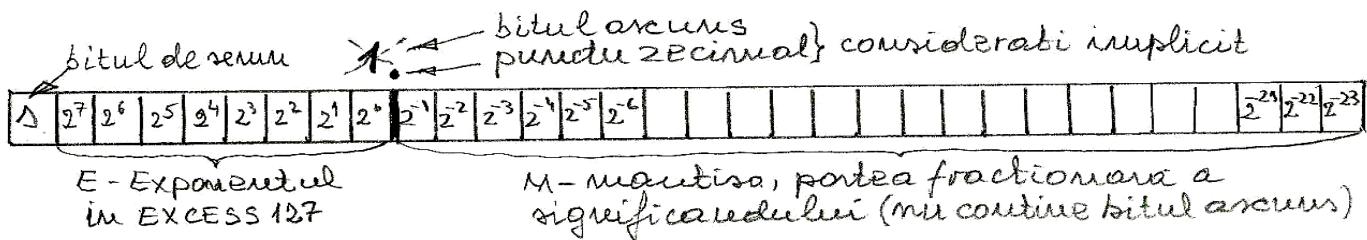
$$S_{\min} = 1.\underbrace{000\dots 000}_{23 \text{ biti}} = 1.0|_{10} \leq \underbrace{1.bbb\dots bbb}_S \leq 1.\underbrace{111\dots 111}_{23 \text{ biti}} = (2-2^{-23})|_{10} = S_{\max}, \text{ pentru o simptă precizie}$$

$$S_{\min} = 1.\underbrace{000\dots 000}_{52 \text{ biti}} = 1.0|_{10} \leq \underbrace{1.bbb\dots bbb}_S \leq 1.\underbrace{111\dots 111}_{52 \text{ biti}} = (2-2^{-52})|_{10} = S_{\max}, \text{ pentru o dulă precizie}$$

În standardul IEEE 754 un număr se reprezintă sub forma:

$$N = (-1)^s \cdot 1.M \cdot 2^e = (-1)^s \cdot S \cdot 2^{E-127}; 0 \leq E \leq 255,$$

din aceste numere se exclud valorile de exponent deplasat: $E = 0$ (obtinut din $-127 + 127$ (deplasarea)) și $E = 255$ (obtinut din $127 + 127$ (deplasarea) + 1) care corespund respectiv reprezentelor pentru $-\infty$ și $+\infty$



Exponentul este reprezentat în cod Excess 127, iar mantisa nu este codificată (număr binar natural pozitiv)

EXEMPLUL 3.3. Pentru numerele 3,5 și -11,375 să se scrie codul în formatul IEEE754

$$\text{Numărul: } N = 3.5|_{10} = 11.1|_2 \times 2^0 = (-1)^0 \times 1.11 \times 2^1$$

Bitul de semn: $S = 0$

Exponentul: $e = 1$

$$\text{Exponentul în excess 127: } E = e + 127 = 128|_2 = 1000\ 0000|_2$$

$$\text{Significant: } S = 1.M = 1.\underbrace{1100\dots 000}_{23 \text{ biti}}$$

$$\text{Mantisa: } M = \cancel{1}.\underbrace{1100\dots 000}_{23 \text{ biti}}$$

Formatul ole cuvânt conform IEEE754

S	E	M
0	1000 0000	110,000,000,000,000,000,000

$$-11.375|_{10} = -1011.011|_2 \times 2^0 = (-1)^1 \times 1.011011 \times 2^3$$

$$S = 1$$

$$E = 3$$

$$E = e + 127 = 130|_{10} = 1000\ 0010|_2$$

$$S = 1.\underbrace{01101100\dots 0}_{23 \text{ biti}}$$

$$M = \cancel{1}.\underbrace{01101100\dots 0}_{23 \text{ biti}}$$

1	1000	0010	011	0110	0000	0000	0000	0	0	0	H
C	1	3	6	0	0	0	0	0	0	0	H

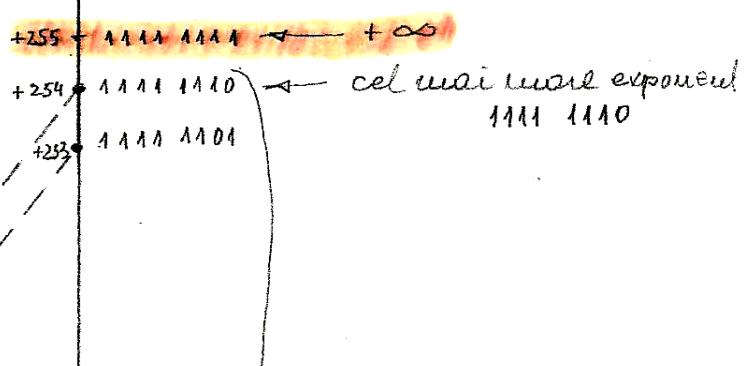
(Conversia unui număr real din zecimal în binare se obține prin:

- partea întreagă se împarte succesiv cu doi și se consideră, în sens invers, sirul de biți ai resturilor;
- partea zecimală se înmulțește succesiv cu doi și se consideră sirul de biți partea întreagă rezultată).

- Reprezentarea exponentului (e) în cod desprinsat +127. ($E = e + 127$)

$$E = e + 127 \text{ (exponentul desprinsat +127)}$$

La reprezentarea exponentului în IEEE 754 se alege codul de mărime și semn. Dar pe urmă că exponentul cel mai negativ (-127) care corespunde lui $e = -\infty$, în limita de bini alocată și produce valoarea $2^{-\infty} = 0$, se alege pentru $-127 = -\infty$ reprezentarea 0000 0000 care împreună cu semnul său cod desprinsat cu +127 ($E = e + 127$)



Intervalul de variație a exponentului (E) în cod desprinsat +127 pentru reprezentările normalize

EXPOENTUL (e)
în cod mărime
și semn

SEMNA	0 111 1111	+127	1000 0000
0 111 1110	+126	0111 1111	0111 1110

Intervalul de variație a exponentului (e) în cod mărime și semn pentru reprezentările normalize

0 : 000 0010	+2	0000 0010	cel mai mic exponent 0000 0001
0 000 0001	+1	0000 0001	
0 000 0000	+0	0000 0000	
0 { 000 0000	-0		
1 000 0001	-1		
1 000 0010	-2		

1 111 1101
1 111 1110
1 111 1111

-125

-126

-127

Valorile limite pentru exponentul desprinsat E

$$e = -126 \rightarrow E = (-126 + 127) = 1 \rightarrow 0000 0001$$

$$e = +127 \rightarrow E = (127 + 127) = 254 \rightarrow 1111 1110$$

- Valorile limite pentru significantul denormalizat

$$\begin{aligned} S_{\min} &= 0.000\ 0000\ 0000\ 0000\ 0001 \\ S_{\max} &= 0.111\ 1111\ 1111\ 1111\ 1111 \end{aligned}$$

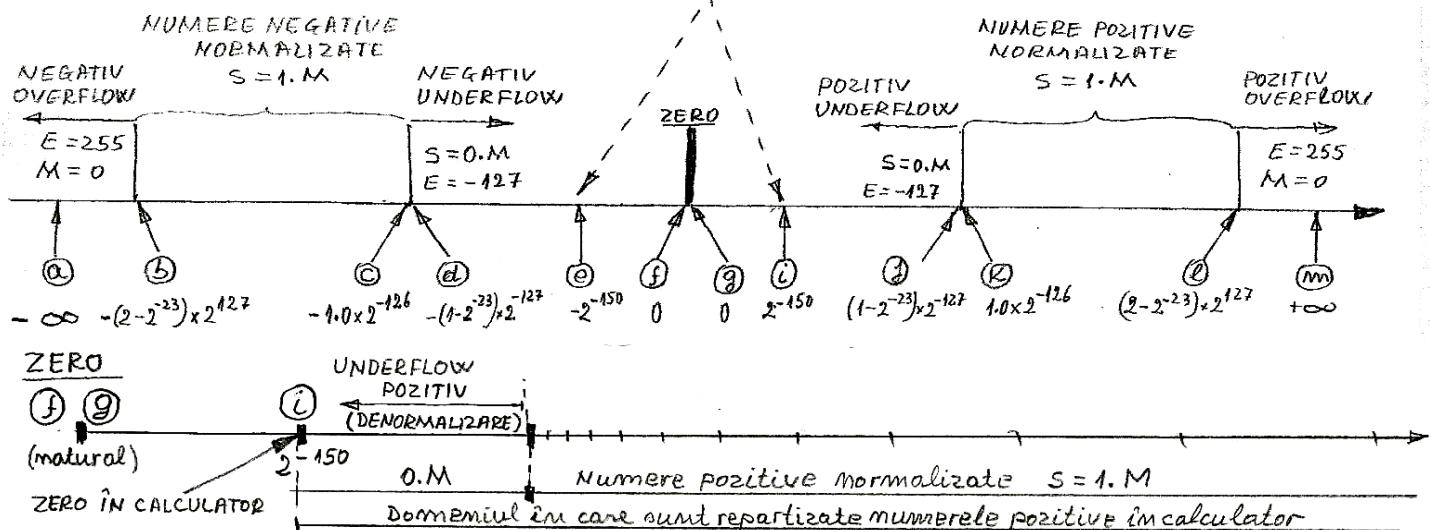
INTERVALELE DE REPREZENTARE A NUMERELOR IN STANDARDUL IEEE-754 (scimătă precizie)

	A	E	M
① $-\infty$	1	1111 1111	000 0000 0000 0000 0000
⑥ Cel mai mic număr negativ normalizat $S = 1 \cdot M$ $N = -(2 - 2^{-23}) \times 2^{127}; E = 127, E = 254$	1	1111 1110	111 1111 1111 1111 1111
⑦ Cel mai mare număr negativ normalizat $S = 1 \cdot M$ $N = -1 \cdot 0 \times 2^{-126}; E = -126, E = 1$	1	0000 0001	000 0000 0000 0000 0000
⑧ Cel mai mic număr negativ denormalizat $S = 0 \cdot M$ $N = -(1 - 2^{-23}) \times 2^{-127}; E = -127, E = 0$	1	0000 0000	111 1111 1111 1111 1111
⑨ Cel mai mare număr negativ denormalizat $S = 0 \cdot M$ $N = -2^{-23} \times 2^{-127} = -2^{-150}; E = -127, E = 0$	1	0000 0000	000 0000 0000 0000 0001
⑩ ZERO NEGATIV $= 0$	1	0000 0000	000 0000 0000 0000 0000
⑪ ZERO POZITIV $+0$	0	0000 0000	000 0000 0000 0000 0000
⑫ Cel mai mic număr pozitiv denormalizat $S = 0 \cdot M$ $N = 2^{-23} \times 2^{-127} = 2^{-150}; E = -127, E = 0$	0	0000 0000	000 0000 0000 0000 0001
⑬ Cel mai mare număr pozitiv denormalizat $S = 0 \cdot M$ $N = (1 - 2^{-23}) \times 2^{-127}; E = -127, E = 0$	0	0000 0000	111 1111 1111 1111 1111
⑭ Cel mai mic număr pozitiv normalizat $S = 1 \cdot M_{\min} = 1 \cdot 0$ $N = 1 \cdot 0 \times 2^{-126}; E = -126, E = 1$	0	0000 0001	000 0000 0000 0000 0000
⑮ Cel mai mare număr pozitiv normalizat $S = 1 \cdot M_{\max}$ $N = (2 - 2^{-23}) \times 2^{127}; E = 127, E = 254$	0	1111 1110	111 1111 1111 1111 1111
⑯ $+\infty$	0	1111 1111	000 0000 0000 0000 0000
⑰ Not a Number NaN; $E = 255, M \neq 0$	0	1111 1111	XXX XXXX XXXX XXXX XXXX XXXX

Trecere graduale spre zero

- NaN, acest format nu este interpretat de procesor ca un număr ci ca un semnal care apare în urma unei operații sărate sau ($V=1, 0 \times \infty, 0/0$). Se generează NaN și când intr-o operație se utilizează un NaN. Există o întreagă "familie" de formate definite ca NaN ($M \neq 0$) de exemplu: NaN stabilizat general de procesor în urma unei operații invalide; NaN de renormalizare generat când se aplică un operand care este un NaN

$$0.000 \dots 001 \times 2^{-127} = 1 \times 2^{-23} \times 2^{-127} = 2^{-150} \text{ (zero în calculator)}$$



3.3.3.3 Operații în virgulă flotantă (FLOPS, MFLOPS)

Exemplificarea operațiilor se va face cu numerele: $X = X_s \cdot B^{X_E} = 0.3 \cdot 10^2$; $Y = Y_s \cdot B^{Y_E} = 0.2 \cdot 10^3$

- Adunarea/scăderea

$$\begin{aligned} X + Y &= X_s \cdot B^{X_E} + Y_s \cdot B^{Y_E} = (X_s \cdot B^{X_E - Y_E} + Y_s) B^{Y_E} \\ X - Y &= X_s \cdot B^{X_E} - Y_s \cdot B^{Y_E} = (X_s \cdot B^{X_E - Y_E} - Y_s) B^{Y_E} \end{aligned} \quad \left. \begin{array}{l} X_E \leq Y_E \\ X_E \geq Y_E \end{array} \right\}$$

$$\begin{aligned} X + Y &= (0.3 \times 10^{2-3} + 0.2) 10^3 = 0.23 \times 10^3 = 230 \\ X - Y &= (0.3 \times 10^{2-3} - 0.2) 10^3 = (-0.17) 10^3 = -170 \end{aligned}$$

1. Mantisa numărului mai mic (X_s) se deplasează spre dreapta cu ($Y_E - X_E$) poziții (aliniere) crescând simultan exponentul X_E până se ajunge la valoarea Y_E ;
2. Se obține mantisa rezultată prin adunarea/scăderea celor două mantine;
3. Mantisa rezultată se normalizează prin deplasări stânga/dreapta simultan cu scăderea/creșterea exponentului Y_E ;
4. Se rotunjește/aproximează mantisa rezultată.

- Înmulțirea/împărțirea

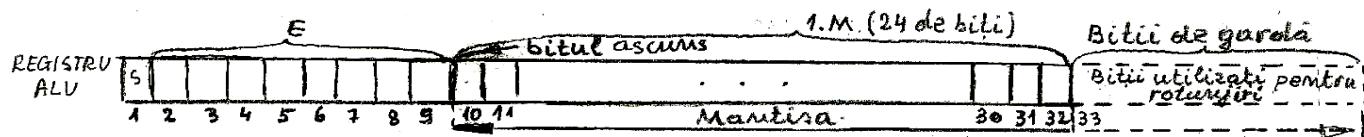
$$\begin{aligned} X \times Y &= X_s \cdot B^{X_E} \times Y_s \cdot B^{Y_E} = (X_s \times Y_s) B^{X_E + Y_E} \\ X : Y &= X_s \cdot B^{X_E} / Y_s \cdot B^{Y_E} = \frac{X_s}{Y_s} \times B^{X_E - Y_E} \end{aligned} \quad \begin{aligned} X \times Y &= (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000 \\ X : Y &= (0.3 / 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15 \end{aligned}$$

1. Mantinele se înmulțesc/împart;
2. Se sumează exponentii pentru înmulțire și se scad pentru scădere;
3. Din exponentul rezultat la înmulțire se scade valoarea excesului (127), iar din cel rezultat la scădere se adună excesul (127);
4. Se normalizează mantinele prin deplasări stânga/dreapta simultan cu scăderea/creșterea exponentului Y și se rotunjesc/aproximează.

- Depășiri.

1. Depășire exponent (Exponent overflow). Valoarea exponentului rezultat depășește valoarea maximă pentru numere normale (+127), adică are valoarea $+\infty$.
2. Subdepășirile (Exponent underflow). Exponentul negativ rezultat este mai mic decât -126. Aceasta înseamnă că numărul rezultat este prea mic pentru a fi reprezentat și poate fi raportat ca fiind zero.
3. Subdepășire significant (Significant underflow). În procesul de aliniere a matiselor biții pot fi deplasati la dreapta peste poziția 2^{-23} .
4. Depășire significant (Significant overderflow). În procesul de adunare a matiselor poate apărea un transfer dela bitul cel mai semnificativ (bitul ascuns). Se poate realiniua cu mărirea exponentului.

- Biții de gardă. Registrele ALU (care aplică operanții și care colectează rezultatul de la ALU) ce conțin o dată sau o dată rezultat în virgulă flotantă au lungimi pentru reprezentarea valorii mantisei ce depășesc 24 de biți (s-a considerat că s-a introdus și bitul ascuns) cu un anumit număr de biți, denumiți biți de gardă (ascunși) cum este prezentat în figura următoare.



Exemplul 3.4 Fie următoarele două numere pozitive: $X = 1.000\dots00 \times 2^1$ și $Y = 1.111\dots11 \times 2^0$. Să se realizeze operația $X-Y$ pentru cazul când matisa este reprezentată fără biți de gardă și pentru cazul când pentru reprezentarea mantisei există și patru biți de gardă

$$x = \underbrace{1.000\dots00}_{24 \text{ biți}} \underbrace{0000 \times 2^1}_{\text{gardă}}; y = \underbrace{1.111\dots11}_{24 \text{ biți}} \underbrace{0000 \times 2^0}_{\text{gardă}} \xrightarrow{\text{primă atragere}} y = \underbrace{0.111\dots11}_{24 \text{ biți}} \underbrace{1000 \times 2^1}_{\text{gardă}}$$

scădere fără biți de gardă

$$\begin{aligned} x &= 1.000\dots00 \times 2^1 \\ -y &= 0.111\dots11 \times 2^1 \\ z_1 &= \underbrace{0.000\dots01}_{24 \text{ biți}} \times 2^{-22} = \underbrace{1.000\dots00}_{24 \text{ biți}} \times 2^{-22} \end{aligned}$$

scădere cu biți de gardă

$$\begin{aligned} x &= 1.000\dots00 \quad 0000 \times 2^1 \\ -y &= 0.111\dots11 \quad 1000 \times 2^1 \\ z_2 &= \underbrace{0.000\dots00}_{24 \text{ biți}} \quad 1000 \times 2^1 = \underbrace{1.000\dots00}_{24 \text{ biți}} \quad 0000 \times 2^{-23} \\ z_1 &= \frac{1.000\dots00 \times 2^{-22}}{1.000\dots00 \quad 0000 \times 2^{-23}} = \frac{1}{2} \end{aligned}$$

- Rotunjirea/approximarea. Din banca de registre ale procesorului sau din memorie, unde numerele sunt stocate în format IEEE-754, sunt extrase: matisa 23 biți (0-22), exponentul 8 biți (23-30) și bitul de semn (31). Mantisa, M, este completată cu bitul ascuns obținându-se significantul, 1.M, (24 biți), iar acesta este depus în registrele care aplică operanții la unitatea aritmetică și logică în virgulă flotantă. Aceste registre care aplică operanții precum și registrele în care se obțin rezultatele de la unitatea aritmetică și logică pe lângă cei 24 de biți au în completare 4-6 biți, numiți *biți de gardă*. Biți de gardă sunt necesari pentru obținerea unor rezultate cât mai corecte în urma rotunjirii sau trunchierii rezultatelor la 23 biți (mantisă) și apoi înscriverea acestora sub forma standard IEEE-754 în banca de registre ale procesorului. Modul în care sunt utilizării biți de gardă este prezentat în exemplele următoare:

– Rotunjire la valoarea cea mai apropiată, exemple:

$1.\underline{xxx\dotsxxx}_{23 \text{ biți}}$	Rotunjire $\begin{array}{c} 1010 \\ \text{gardă} \end{array} \rightarrow \underbrace{xxx\dotsxx}_{23 \text{ biți}} \underbrace{(x+1)}_{\text{rotunjire cu valoarea } 1 \text{ (bit de gardă)}}$
$1.\underline{xxx\dotsxxx}_{23 \text{ biți}}$	$0111 \rightarrow \underbrace{xxx\dotsxx}_{23 \text{ biți}} \underbrace{(x+0)}_{\text{rotunjire la } 0 \text{ cu valoarea } 0 \text{ (bit de gardă)}}$
$1.\underline{xxx\dotsxxx} \quad 1000$	$\rightarrow \underbrace{xxx\dotsxx}_{23 \text{ biți}} \underbrace{(x+0 \text{ sau } x+1)}_{\text{pentru că } 1000 = 2^{-24}}$

– Trunchiere, se consideră doarcei 23 de biți ai mantisei, se negljează biții de gardă

$$1.\underline{xxx\dotsxxx}_{23 \text{ biți}} \quad \underbrace{x \ x \ x \ x}_{\text{gardă}} \rightarrow \underbrace{xxx\dotsxx}_{23 \text{ biți}}; \text{biți de gardă se negligează}$$

Prin operația de rotunjire sau de trunchiere numărul se exprină cu o anumită eroare, iar pe o succesiune de operații erorile introduse prin rotunjire/trunchiere se pot cumula!

- Adunarea în virgulă flotantă

în zecimal

$$x = 9.999|_{10} \times 10^1$$

$$y = 1.610|_{10} \times 10^{-1}$$

1. Se comparați cele două expoziții. Se deplasăza opereaza dreapta numărul mai mic până când cele două expoziții au aceeași valoare

$$x = 9.999 \times 10^1, y = 0.016 \times 10^1$$

în binar

$$x = 0.5|_{10} = 1/2|_{10} = 1 \times 2^{-1} = 1.000 \times 2^{-1}$$

$$y = -0.4375|_{10} = -7/16|_{10} = -111 \times 2^{-4} = -1.110 \times 2^{-2}$$

2. Se adună cele două operaanzi

$$s = (9.999 + 0.016) \times 10^1 = 10.015 \times 10^1 \quad | \quad s = (1.000 - 0.111) \times 2^{-1} = 0.001 \times 2^{-1}$$

3. Se normalizează suma obținută, prin deplasare stânga/dreapta și decrementare/incrementare a expoziției, astfel încât prima cifră înainte de punctul zecimal să fie o cifră diferită de zero.

$$s = 10.015 \times 10^1 = 1.0015 \times 10^2$$

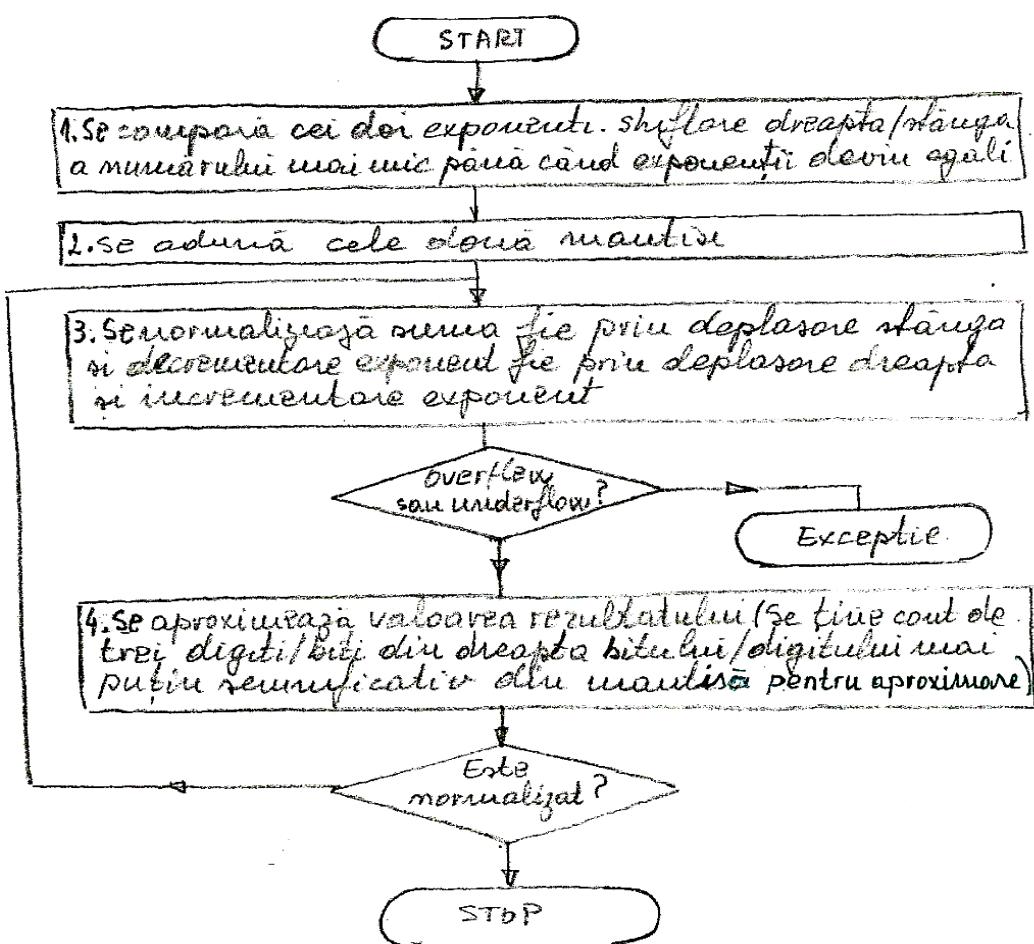
$$s = 0.001 \times 2^{-1} = 1.000 \times 2^{-4}$$

Dacă expoziția se întinde de la -127 < -4 < +127 nu există nici undeoverflow sau underflow. $E = -4 + 127 = 123$

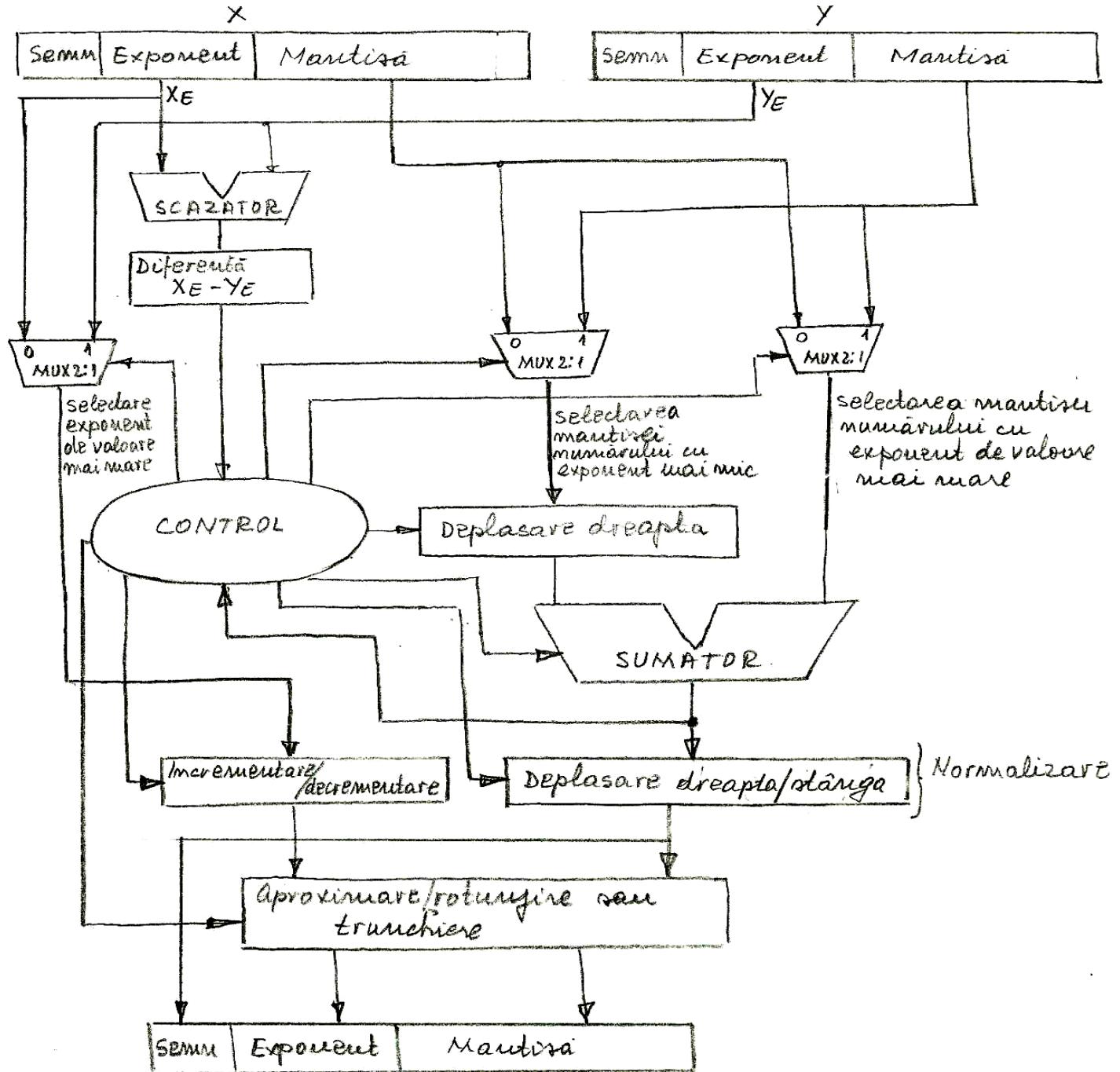
4. Rotunjirea rezultatului. Se impune ca aproximarea significativii să se facă numai cu patru digiti/biti

$$s = 1.001 \times 10^2$$

$$s = 1.000 \times 2^{-4}$$



- Structurarea unității de adunare în virgulă flotantă



Exponentul Y_E al operandului Y este scăzut din exponentul X_E al operandului X și în funcție de semnul diferenței ($Y_E - X_E$) se determină operandul cu exponentul mai mare, această operație este efectuată pe un sumator/scăzător de dimensiune redusă (exponenții sunt exprimați pe 8 biți în simplă precizie). În funcție de semnul acestei diferențe unitatea de control comandă trei multiplexoare care (în ordinea de la stânga la dreapta în figură) selectează: exponentul de valoarea mai mare, mantisa operandului cu exponentul mai mic, mantisa operandului cu exponentul mai mare. Mantisa operandului cu exponentul mai mic este deplasată la dreapta cu un număr de poziții egal cu diferența exponenților $|Y_E - X_E|$, după care pe sumatorul de dimensiune mai mare (24 biți fără biții de gardă) este sumată cu cealaltă mantisă. În etapa de normalizare a rezultatului acesta este deplasat stânga/dreapta cu decrementarea/incrementarea exponentului (cel de valoare mai mare), iar după rotunjire/trunchiere se obține rezultatul.

- Înmulțirea în virgulă flotantă.

În decimal

$$\begin{aligned} X &= 1.110_{10} \times 10^{10} \\ Y &= 9.200_{10} \times 10^{-5} \\ X_E &= 10 + 127 = 137; Y_E = -5 + 127 = 122 \end{aligned}$$

În binar

$$\begin{aligned} X &= 0.5_{10} = 1.000 \times 2^{-1} \\ Y &= -0.4375_{10} = -1.110 \times 2^{-2} \\ X_E &= -1 + 127 = 126; Y_E = -2 + 127 = 125 \end{aligned}$$

1. Se adună cei doi expoziți deplasatori și apoi din suma rezultată se extrage valoarea de deplasare 127

$$X_E + Y_E = (10 + 127) + (-5 + 127) - 127 = 132$$

$$X_E + Y_E = (-1 + 127) + (-2 + 127) - 127 = 124$$

2. Se înmulțesc măntisile

$$\begin{array}{r} 1.110 \times 9.200 \\ \hline 0000 \\ 0000 \\ 2220 \\ 9999 \\ \hline 10.212000 \end{array}$$

$$P = 10.212 \times 10^{132-127} = 10.212 \times 10^5$$

$$\begin{array}{r} 1.000 \times 1.110 \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1.110000 \end{array}$$

$$P = 1.110 \times 2^{124}$$

3. Se normalizează produsul dacă este necesar prin deplasare stânga/dreapta cu decrementarea/fiecrea unor unități a expozițului

$$P = 1.0212 \times 10^6$$

$$P = 1.110 \times 2^{124}$$

$0 < 124 < 255$ nu există overflow sau underflow

5. Se rotunjesc măntisa la 4 biți (digiți)

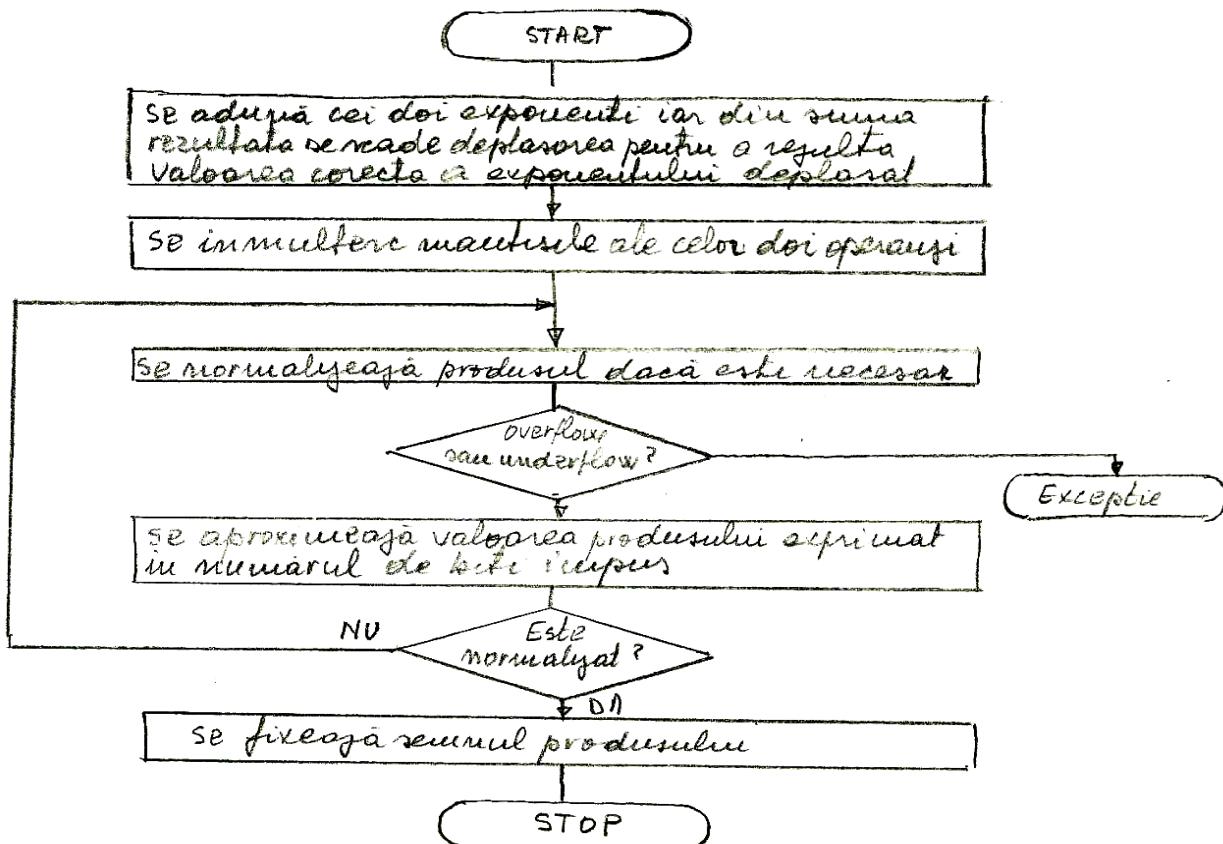
$$P = 1.021 \times 10^6$$

$$P = 1.110 \times 2^{124}$$

6. Se atasează semnul

$$P = 1.021 \times 10^6$$

$$P = -1.110 \times 2^{124-127} = -1.110 \times 2^{-3}$$



3.4. REGISTRE

Registrele constituie memorie (internă) din calea de date, se spune că registrele sunt “zestrea” microprocesorului. Registrele ca memorie internă în raport cu memoria externă prezintă avantajele: 1- timp de acces cel puțin cu un ordin de mărime mai mic; 2- cuvânt de adresare de maximum 4-7 biți (în corpul unei instrucțiuni); 3. Registrele fiind o structură uniformă prezintă un layout-ul de o mare regularitate.

Modul de organizare al regastrelor este un atribut architectural al procesorului. *Organizarea regastrelor împreună cu arhitectura setului de instrucțiuni (ISA), cu tipurile de date și cu modurile de adresare sunt suportul pentru programator, pentru codul generat de compilator și pentru sistemul de operare.*

- *Pentru programator* organizarea regastrelor este de dorit să aibă o regularitate și flexibilitate cât mai ridicată, adică să fie o organizare simetrică. Uneori, pentru organizarea regastrelor se cere și păstrarea unei compatibilități cu implemetările anterioare ale procesorului.
- *Pentru compilator* o **organizare simetrică** a regastrelor (**regularitate + flexibilitate**) constituie suportul pentru generarea unui cod eficient. Un set simetric de registre oferă posibilitatea ca oricare registru să poată fi utilizat:

- de către oricare instrucțiune (OPCODE);
- ca acumulator (în operații);
- ca sursă/destinație în operații;
- ca pointer, registru index.

Un set simetric de registre și un număr suficient de registre formează un suport pentru un set ortogonal de instrucțiuni. O arhitectură cu suport pentru compilator este referită ca arhitectură îndreptată spre compilator (Compiler-oriented-architecture).

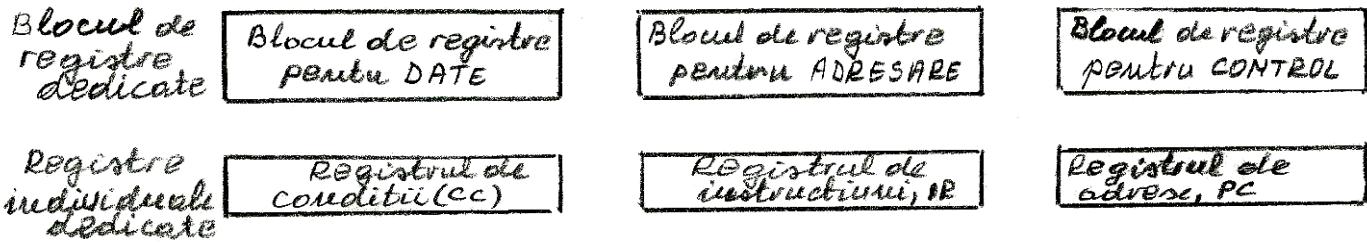
- *Pentru sistemul de operare* o organizare potrivită a regastrelor (Operating-system-oriented-architecture) poate constitui suport pentru :
 - comutarea proceselor;
 - răspunsul la întreruperi;
 - adresarea spațiului memoriei și asigurarea protecției;
 - sincronizare în lucrul multiprocesor și depanare.

În concluzie, o organizare simetrică (regularitate + flexibilitate) pentru setul de registre interne plus un ISA ortogonal constituie un suport pentru compilator și pentru sistemul de operare.

3.4.1 Structurarea regastrelor într-un microprocesor.

Registrele - resurse ale procesorului – sunt structurate în trei bănci/blocuri (fiecare bancă conținând, ușual, 16, 32, 64, 128) plus câteva registre specializate.

1. Banca regastrelor de date, conține acele registre care sunt utilizate în lucrul cu date (procesare și transfer);
2. Banca regastrelor de adresare, conține acele registre care sunt utilizate în determinarea adreselor;
3. Banca regastrelor de control, conține acele registre pentru controlul funcționării procesorului și pentru funcțiunile sistemului de operare.



Ușual, registrele individuale dedicate sunt:

– *Registrul de adrese, PC* (Program Counter). Conține adresa instrucțiuni care se extrage din memorie pe durata ciclului FETCH, iar la sfârșitul acestui ciclu se incrementează pentru a indica adresa următoarei instrucțiuni care se

va extrage, $PC \leftarrow PC + 1$, iar dacă instrucțiunea următoare se obține printr-un salt la o adresă ADRESA atunci $PC \leftarrow ADRESA$.

– *Registrul de instrucțiuni, IR* (Instruction Register). Este registrul în care se depune instrucțiunea adusă din memorie în ciclul de FETCH și care se păstrează în acest registru până la terminarea ciclului EXECUTE.

Câmpurile instrucțiunii din acest regisztr se aplică la unitatea de control pentru a se genera semnalale de control necesare ciclului de execuție, totodată unele câmpuri se aplică la registrele de date/adrese pentru a se extrage date și/sau a se calcula adrese.

– *Registrul codurilor de condiții CC* (Condition Codes). În acest regisztr se colectează biții corespunzători condițiilor realizate/sau în urma efectuării unei instrucțiuni (Z-zero, N-negativ, P/ pozitiv, C-Carry, V-overflow/underflow, <, >, = etc). În acest regisztr pot fi cuprinși și biții care indică starea sistemului: validare întreruperilor, validare timere interne, user/sistem etc. Acest regisztr împreună cu PC și cu altele, definind starea sistemului, sunt referite prin PROGRAM STAUS WORD, PSW. La schimbarea contextului se salvează PSW care, apoi, se reface la restabilirea contextului.

- La procesoarele actuale organizarea anterioară a regisztrelor se reduce doar la două bânci:
 1. Banca regisztrelor generale GPR (General Purpose Register)
 2. Banca regisztrelor de control.

Față de organizarea anterioară banca de registre de adresare a fost inclusă în regisztrele de date și în regisztrele de control.

*Blocuri
de regisztri
dedicati*

Regisztre generale, GPR

Regisztre de control

Regisztrele (de utilizare) generale, GPR. Pentru conținuturile acestor regisztre generale sunt permise aceleași operații, fie că sunt cuvinte DATA, fie că sunt cuvinte ADRESĂ. Regisztrele generale sunt vizibile pentru programator. Dar și în cadrul bâncii de regisztre generale pot exista anumite regisztre care sunt dedicate acestea sunt fie asignate prin convenție, fie dedicate prin cablare. De exemplu la MIPS prin convenție sunt următoarele asignări: \$gp $\leftarrow \$28$; \$sp $\leftarrow \$29$; \$fp $\leftarrow \$30$; \$ra $\leftarrow \$31$. Operațiile cărora li se pot asigna prin cablare (hard) regiszre sunt: Call Return (regisztrul Stack Pointer, SP); user/program; realizarea unor adresări frecvente, realizarea salturilor, operator zero (\$zero) etc.

Regisztrele de control. Aceste regisztre sunt destinate ca suport pentru controlul anumitor componente ale procesorului sau regimuri de funcționare, deoarece acestea sunt foarte numeroase rezultă că și regisztrele de control corespunzătoare sunt în număr mare dar și foarte diferite. Sumar, astfel de regisztre de control ar putea fi pentru: controlul și protecția lucrului cu memoria, cu memoria cache, cu TLB (Translation-Look-aside-Buffer), regimul de întreruperi, depanare și diagnosticare etc. Un regisztr de control cu lungimea doar de câțiva biți este regisztrul pentru configurarea sistemului, biții acestuia, prin înscrriere la început configurației sistemului, indică prezența în sistem a unor dispozitive externe: coprocesor matematic, unitate pentru managementul memorie (MMU), circuit pentru întreruperi vectorizate, DMA etc. În general, se zice că arhitectul pentru fiecare nouă funcțiune introdusă mai adaugă un regisztr de control!

Regisztre auxiliare. Aceste regisztre auxiliare, uneori vizibile alteori transparente pentru programator, sunt suport temporar pentru operațiile realizate de procesor, pentru compilator sau pentru sistemul de operare.

– Multe operații din procesor, între două stări, necesită o stocare temporară care are mai mult rol electric sau de temporizare decât logică. De exemplu, la prezentarea schemei de principiu a căii de date sunt prezente regiszrele MAR (Memory Address Register) și MDR (Memory Data Register) care au rolul electric, de buffer, între calea de date și exterior.

– Uneori compilatorul sau mai ales sistemul de operare (regimul kernel) necesită anumite stocări pentru faze intermediare, în consecință pentru aceste situații sunt necesare regiszre suport. De exemplu, la procesorul MIPS, prin convenție sunt dedicate sistemului de operare \$k0 $\leftarrow \$26$, \$k1 $\leftarrow \$27$, iar pentru compilator/asamblor \$at $\leftarrow \$1$.

EXEMPLUL 3.5.

A. Organizarea tip fereastră a registrelor interne. Acest mod de organizare, introdus inițial la procesoarele RISC1 și RISC 2 (Berkeley University, prof. Patterson), iar apoi continuat în procesoarele SPARC și IA-64, țineste spre creșterea vitezei la schimbarea contextului procesorului. Situațiile de schimbarea contextului sunt : Call-Return, intreruperi interne/externe, comutarea proceselor; se estimează ca schimbarea contextelor încarcă procesorul (regia- timpul consumat de procesor pentru schimbarea contextelor) cu până la 40%.

O problemă care trebuie rezolvată la schimbarea cotextului, pentru Call-Return, este transferul parametrilor, iar rezolvarea depinde de numărul parametrilor transferați; număr care variază în funcție de tipul de program după cum este prezentat în tabelul următor

Procedura apelată prezintă:	Frecvența în programe de tipul:	
	Compilator, Interpretator, Procesare de texte.	Programe numerice de dimensiune mică.
> 3 argumente (parametrii)	0-7%	0-5%
> 5 argumente (parametrii)	0-3%	0%
> 8 cuvinte care sunt argumente și scalari locali	1-20 %	0-6%
>12 cuvinte care sunt argumente și scalari locali	1-6%	0-3%

Organizarea de tip fereastră necesită un număr mare de registre (registre multiple) în setul de registre interioare ale procesorului, în cazul din acest exemplu sunt în total 138. Pentru a explica organizarea și funcționarea registrelor multiple din procesor se consideră o înlănțuire de subroutines prin apelare A → B → C → D → ... → Y, apoi prin reîntoarcere Y → ...D → C → B → A. Fiecărei subroutine i se repartizează o fereastră (bancă) de 32 registre, din care 10 registre R0-R9 sunt comune pentru toate ferestrele succesive – registre globale – conține date globale pentru program, iar celelalte 22 de registre sunt asignate în modul următor:

1. 6 registre de intrare (superioare), R26-R31, aceste registre sunt comune cu cele 6 registre de ieșire (inferioare) ale subroutinei apelante, prin acestea se face transferul direct al parametrilor de la apelant la apelat fără a se schimba fizic registrele (datele rămân în același regisztr), deci fără consum de timp la schimbarea de context;
2. 10 registre locale, R16-R15, proprii ale subroutinei. Aceste registre sunt utilizate în exclusivitate de subroutine, nu sunt multiplexate cu registrele de ieșire (inferioare) sau cu registrele de intrare (superioare) de la cele două subroutine adiacente;
3. 6 registre de ieșire (inferioare), R10-R15, comune cu cele 6 registre de intrare (superioare) ale subroutinei apelate și căruia i se transmit parametrii necesari, fără consum de timp, prin aceste registre comune.

O subroutine utilizează pentru procesarea proprie cele 10 registre locale, dar înainte de apelarea următoarei subroutine înscrie parametrii transmiși în cele șase registre de ieșire (inferioare), la fel, înainte de reîntoarcere în subroutine apelantă înscrie valorile calculate în cele șase registre de intrare (superioare). Prin această ferestruire, când registrele de ieșire ale subroutinei apelante sunt comune cu registrele de intrare de la subroutine apelată, transferul de parametrii se face fără consum de timp, similar se realizează și reîntoarcerea valorilor calculate de subroutine apelată către subroutine apelantă.

Pentru organizarea de tip fereastră apar unele dezavantaje:

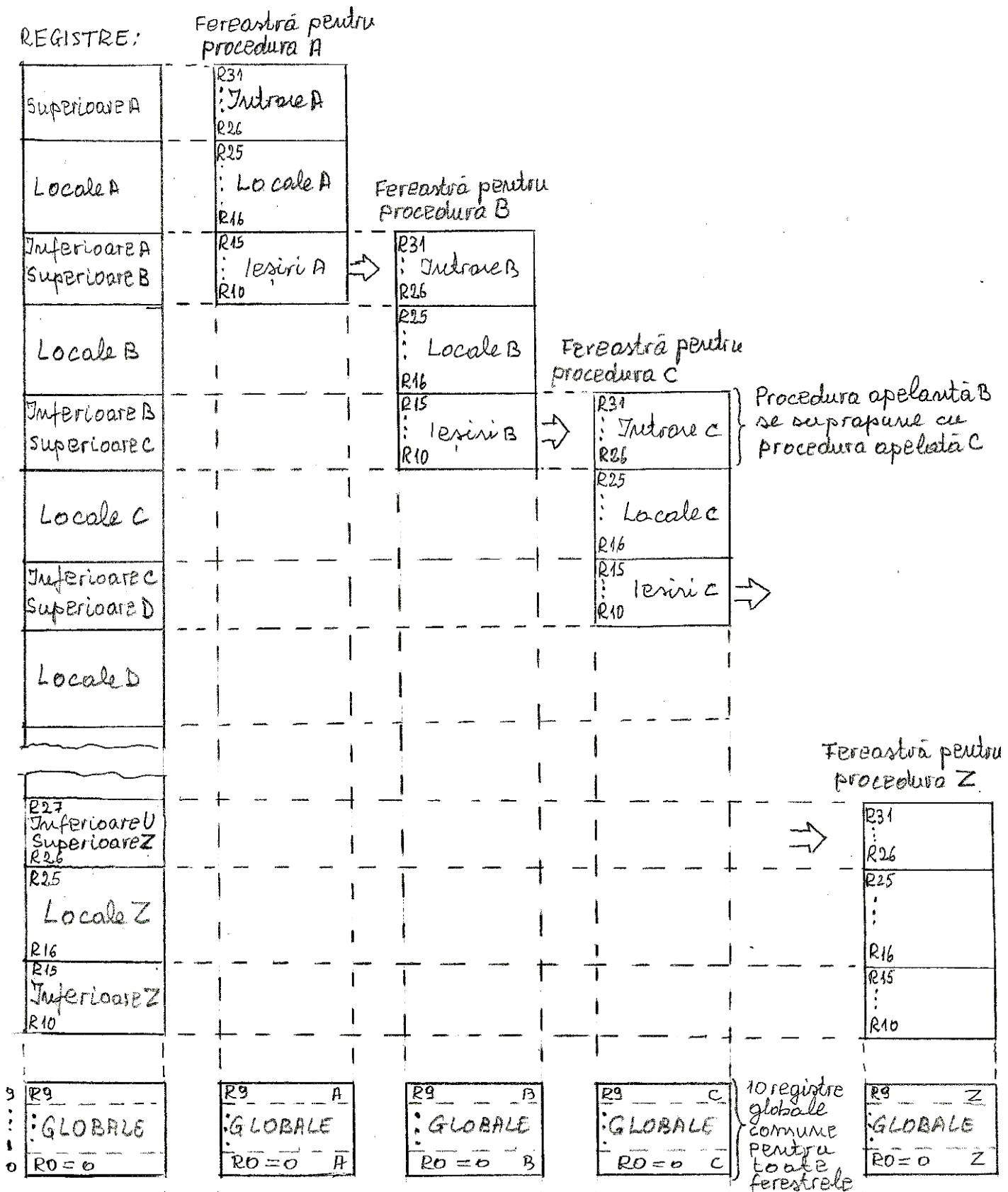
- timp de acces mai lung la registre (datorită faptului că sunt mai multe registre realizate, de exemplu 138, decât în cazul unei singure bânci de 32);
- necesitatea unui decodificator suplimentar pentru ferestre;
- creșterea complexității căii de control
- consum mai mare pe suprafață de Si.

La organizarea de tip fereastră trebuie analizate:

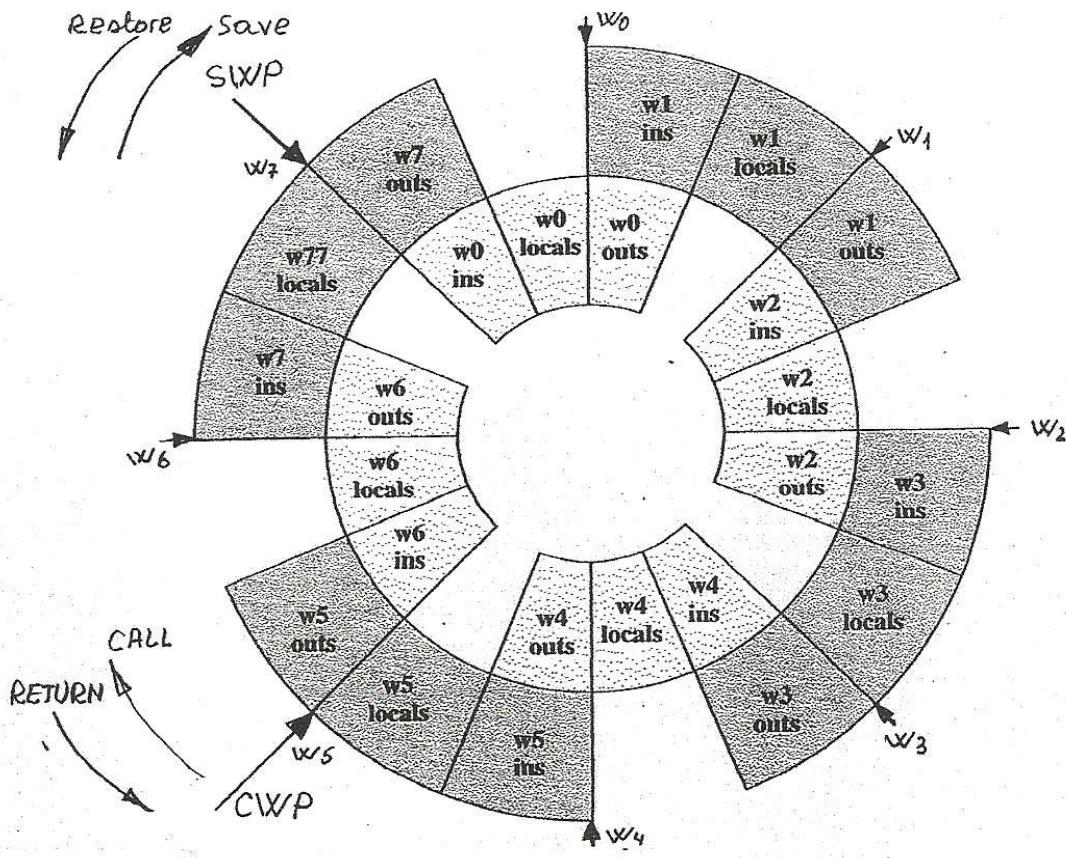
- care este numărul de registre pentru o fereastră, respectiv alegerea optimă între numerele de registre din cadrul unei ferestre în : registre de intrare, registre locale și registre de ieșire;
- determinarea numărului total de ferestre;

- ferestre cu un număr fix, variabil sau combinat de număr de registre pe fereastră.

Organizarea de tip fereastră a regisztrilor interne.



B. Organizarea circulară a ferestrelor. Această organizare este un mod particular al organizării anterioare, obținut prin conectarea ultimei ferestre cu prima fereastră, formând astfel un cerc ca în figura următoare (fereastra de început, w_0 , este alipită cu fereastra de sfârșit, w_7).



Statistic s-a constatat că o organizare fereastră a registrelor cu un număr relativ mic de ferestre satisfac marea majoritate a apelărilor înlănțuite Call-Return, doar 1% de astfel de apelări necesită o înlănțuire mai mare de opt ferestre. Dacă sunt mai multe apelări decât numărul total de ferestre, când apar astfel de situații, ferestrele mai vechi se înscriu în stivă.

Administrarea accesării ferestrelor circulare are ca suport două registre:

1. CWP – Current Window Pointer, care conține adresa ferestrei cu care se lucrează (currentă);
2. SWP – Save Window Pointer, care conține adresa ultimei ferestre (care trebuie salvată în stivă).

Să presupunem că înlănțuirea până în prezent de subroutines a fost de cinci, pornind de la fereastra w_0 până la fereastra w_5 și acum se rulează subroutinea curentă care are alocată fereastra w_5 . Registrele din fereastra w_5 sunt adresate relativ la conținutul pointerului CWP la care se adaugă un offset. Pointerul SWP indică fereastra w_7 . Dacă se apelează în continuare o altă subroutine, căruia îi corespunde fereastra w_6 , atunci subroutinea curentă încarcă parametrii în registrele de ieșire, w_5 outs, care corespund cu registrele de intrare, w_6 ins, ale ferestrei w_6 , iar pointerul CWP va fi înscris cu adresa ferestrei w_6 , $CWP \leftarrow w_6$. Dacă de la w_6 se trece în continuare (la o următoare apelare de subroutine) la w_7 , atunci conținuturile celor doi pointeri ajung să coincidă, $CWP = SWP$, ceea ce generează o excepție. Generarea de excepție este motivată de faptul că dacă w_7 , apelează în continuare o altă subroutine (fereastra w_0) atunci când se înscriv parametrii de transmis în w_7 outs aceștia se vor supraînscrie peste w_0 ins. În acest caz excepția va rezolva: $SWP \leftarrow w_0$; $CWP \leftarrow w_7$, iar conținuturile w_0 ins și w_0 locals se înscriv în stivă.

Conisderând acum revenirea (RETURN), succesiv, de la w_7 până la w_0 când din nou $CWP = SWP$ se generează o excepție; de această dată excepția va rezolva $SWP \leftarrow w_7$; $CWP \leftarrow w_0$, iar conținuturile w_0 ins și w_0 locals vor fi refăcute din stivă.

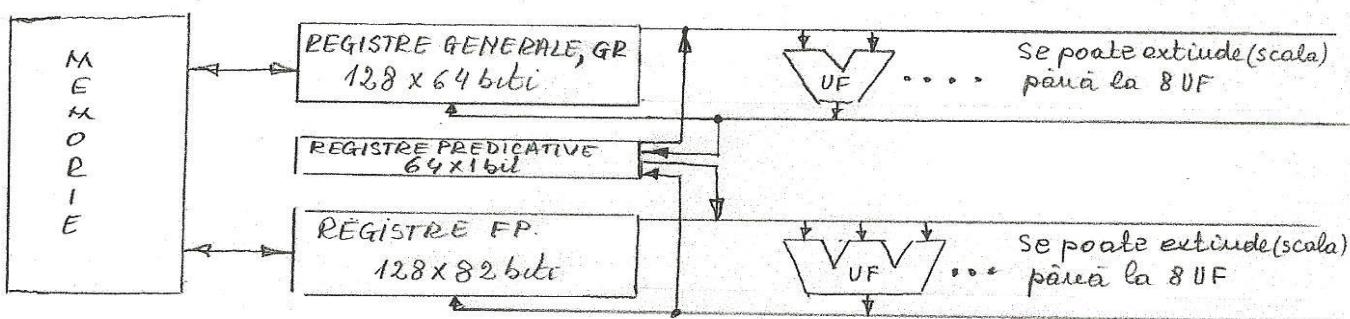
Rezultă că o organizare circulară de N ferestre poate conține un număr de N-1 subroutine activate.

C. Organizarea regiszrelor în arhitectura IA-64

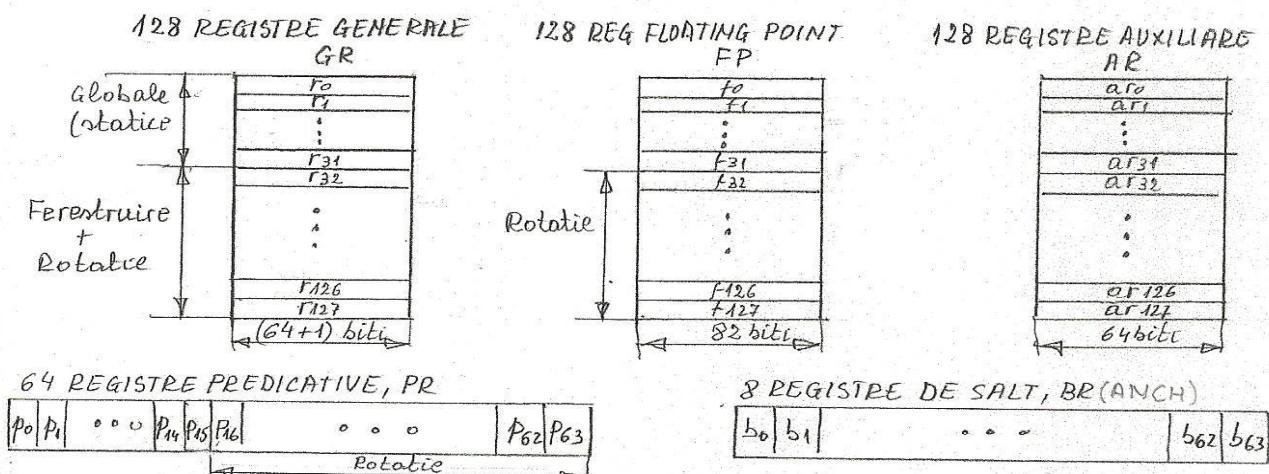
- Caracteristici

PROCESOR	Max. de instrucții lăsate pe tact	Unități Funcționale	Maximul de operații pe tact	FCLK max [GHz]	Nr. de tranzistori 10^6	Potență consumată [Watt]	SPEC Int 2000	SPEC FP 2000
ITANIUM1	6	4 Integr/MMU 2 Acces memorie 3 Realizare branch 2 Floating Point TOTAL = 11	9	0,8	25	130	379	701
ITANIUM2.	6	6 Integr/MMU 4 Acces memorie 3 Realizare branch 2 Floating Point TOTAL = 15	11	1,5	221	130	810	1427

- Arhitectura IA-64



- Organizarea regiszrelor



- 128 GR. Banca de regiszre generale, pentru operații sau adrese; primele 32 regiszre sunt pentru acces global iar restul, începând de la r32 pot fi organizate sub formă de ferestre (circulare). Cașul regiszrele, începând cu r32 sunt folosite independent față de ROTIRE, după ce secesorul regiszrelor și automat la următoarea acțiune va fi $r_i + 1$.
- 128 FP. Banca de regiszre pentru operații în virgula flotantă; regiszrele începând cu f32 pot fi utilizate prin ROTIRE.
- 128 AR. Banca de regiszre de aplicații (Auxiliare); utilizate de procesor pentru a realiza o stivă în hard, pentru realizarea cumuluior ieserităciunii, pentru anumite cerințe ale sistemului de operare.
- 64 PR. Banca de regiszre predicative, pentru execuția instrucțiunilor cu execuție condicioană (predicative), vezi pg 87. (toate construcțiile în IA-64 sunt predicative)
- 8 BR. Banca de regiszre pentru salt, utile în procesul de adresare la memorie (la instrucțiunile de salt). Deoarece citire din memorie se face simultan către 16 byte: ADRESA(modulo 8)=0 trebuie să ultimii 4 biti din adresa=0000

CAP 4. CALEA DE CONTROL

4.1 FUNCȚIA UNITĂȚII DE CONTROL

Divizarea procesorului în două părți distincte: *Calea de Date* și *Calea de Control* (Unitatea de Control) nu este o opțiune numai sub aspect didactic ci este și o opțiune din punct de vedere al proiectării. La un procesor cu set de instrucțiuni, din punct de vedere al organizării și funcționării, apar două acțiuni distincte, care trebuie privite separat: secvențierea instrucțiunii și execuția instrucțiunii.

- *Secvențierea instrucțiunilor* este acțiunea realizată de Unitatea de Control (UC), prin care se selectează instrucțiunea următoare ce trebuie executată, sau altfel spus, cum se trece la instrucțiunea următoare: 1. fie în ordinea de creștere a adreselor în sensul de creștere al numerelor naturale (regulă implicită, deci nu se mai specifică în instrucțiune); 2. fie după o altă regulă, care nu păstrează succesiunea de creștere a numerelor naturale în accesarea adreselor și care trebuie specificată în instrucțiune împreună cu modul următor:

1. $PC \leftarrow PC + k$, k fiind pasul de aliniere a instrucțiunilor în memorie, (ADRESA) modulo $k = 0$;
2. $PC \leftarrow$ Adresă:
 - fixă
 - calculată
 - introdusă(vector)
 =>
 - salt necondiționat
 - salt necondiționat
 - Call/Return
 - Exceptii (hard, soft)

Dar secvențialitatea pentru realizarea traseului prin memorie (zona text), prin extragerea instrucțiunilor (subciclul Fetch), supraveghetă de unitatea de control se continuă apoi și prin secvențierea etapelor în executarea instrucțiunilor (subciclul Execuție) în calea de date.

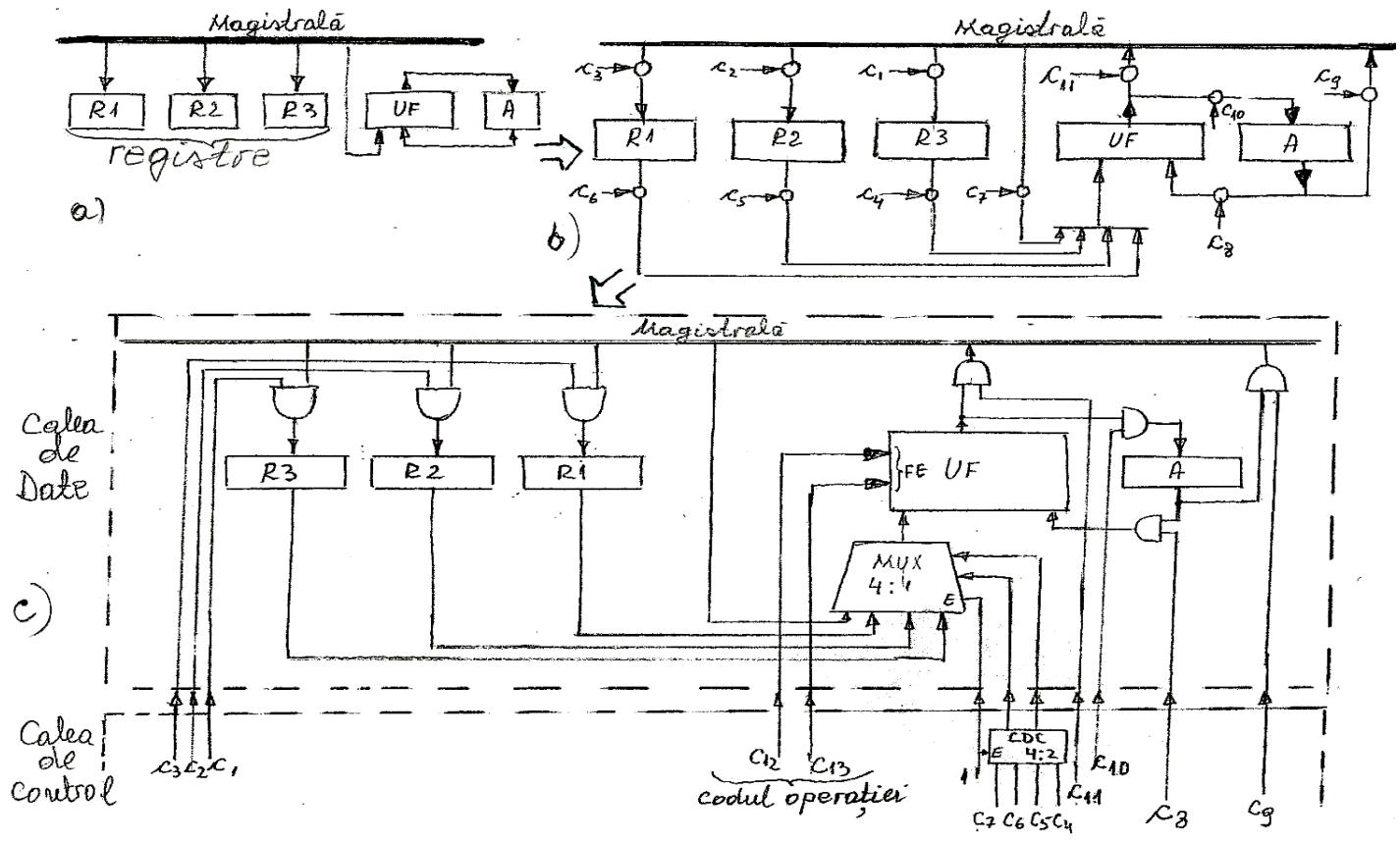
- *Execuția* este acțiunea efectuată în calea de date sub controlul căii de control pentru realizarea funcției/operațiile specificate în codul instrucțiunii. Deoarece calea de date cu un repertoriu limitat de componente (unități funcționale (UF), bloc de registre, magistrale, buffere, shiftere) trebuie să execute întreg setul de instrucțiuni, calea de control pentru execuția fiecărui instrucțiuni alege:

1. componentele necesare din calea de date;
2. traseele dintre aceste componente;
3. succesiunea transferurilor pe traseele dintre componente încât să realizeze în hardware execuția instrucțiunii.

Aceste selectări de componente, configurații de trasee și succesiuni de transferuri în calea de date se realizează prin semnalele generate de unitatea de control (în urma decodificării OPCODE și pe baza semnalelor primite din calea de date).

EXEMPLUL 4.1 Pentru o cale de date în care sunt trei registre (R1, R2, R3), o unitate funcțională, UF și un registru acumulator, A, cu transfer pe o singură magistrală (Figura a) să se stabilească configurația traseelor și punctele de aplicare a semnalelor de control.

Traseele între elementele componente și punctele de control corespunzătoare pentru efectuarea transferurilor sunt desenate în figura b. Traseele sau ales să înțină cont că fiecare registru trebuie să comunice ("să aibă ieșire") pe magistrală și cu unitatea funcțională (evident, cu două sau trei magistrale transferurile se pot face mai ușor și în paralel). De asemenea, la unitatea funcțională trebuie să poată fi adusă informația din fiecare element, iar rezultatul să poată fi înscris în oricare element. Registrul acumulator trebuie să înscrive și să fie înscris din UF, precum și de pe magistrală. Rezultă traseele și punctele de control din figura b. În fiecare punct de control, figura c, semnalul de transmis este aplicat pe o poartă AND împreună cu semnalul de control, care trebuie generat de unitatea de control. Deoarece la UF se aplică informație de la mai multe surse, selectarea uneia dintre aceste surse se realizează prin multiplexor. S-au mai introdus două semnale de control c₁₂ și c₁₂' pentru programarea operațiilor pe UF (în acest caz pentru patru operații).



Semnalele de control generate de calea de control pentru coordonarea acțiunilor din calea de date se supun tripletului de actiuni : "unde" , "când" și "cum".

Prin acțiune de "cum" semnalele de control trebuie să configureze/selecteze cu elementele (magistrale, registre, unități funcționale) din calea de date un traseu încât să se poată modela hardware starea respectivă în execuția instrucțiunii.

Prin acțiune de "unde" semnalul de control trebuie să realizeze comanda într-un anumit punct din calea de date.

Prin acțiune de "când" (în ce moment) semnalul de control trebuie să devină activ într-un anumit moment și să fie activ pe o anumită durată.

Semnalele de control, în general, sunt sincrone cu semnal de ceas și pot fi: 1. semnale de strob, care au o durată scurtă (impuls) servind realizarea unei acțiuni, de exemplu o înscriere; 2. semnale active pe palier, care au durată palierului de unu sau mai multe perioade de ceas și servesc în general pentru configurare de trasee.

Sinteza semnalelor de control are la bază o diagrame de stări (este o organigramă de tip ASM - Algoritmic State Machine, diagramă FSM - Finite State Machine), care descrie grafic etapele pentru interpretarea unei instrucțiuni (Fetch + Execuție), un model generic fiind reprezentat în figura următoare. În această diagrame de stări se disting cele două cicluri de Fetch și Execuție, fiecare dintre cicluri fiind constituit dintr-o etapă sau mai multe etape, cel de Fetch având o singură etapă iar cel de Execuție patru etape (Accesare operanzi, realizarea operației, acces la memorie (pentru înscriere sau citire date) și înscrierea rezultatului produs de instrucțiune). Fiecare etapă la rândul său conține una sau mai multe stări, considerând că o stare este pe durata unui tact de ceas, deci o etapă se consumă pe durate unei sau mai multe perioade de ceas (unele stări pot consuma mai multe perioade de ceas).

În fiecare stare se realizează o anumită funcție, F_i , iar pentru *interpretarea instrucțiunii* (Fetch + Execuție) se realizează succesiunea de funcții $F_1, F_2, F_3, \dots, F_i, \dots, F_k$ corespunzătoare la cele k stări din diagrama, pe intervalul de k (sau mai multe) tacte de ceas. O funcție F_i se compune dintr-o serie de **microoperații** (citirea unui registru, înscrierea unui registru, transferul pe magistrală, realizarea unei operații pe UF etc) care într-un limbaj de descriere RTL (Register Transfer Language) se exprimă în felul următor:

$$Z \leftarrow f(X_1, X_2, \dots, X_n)$$

în care: – Z, X_1, X_2, \dots, X_n sunt registre sau valori din aceste registre

- f este microoperația efectuată (încarcă, citește, transferă, incrementează, adună, decodifică etc)
- \leftarrow indică sensul de transfer.

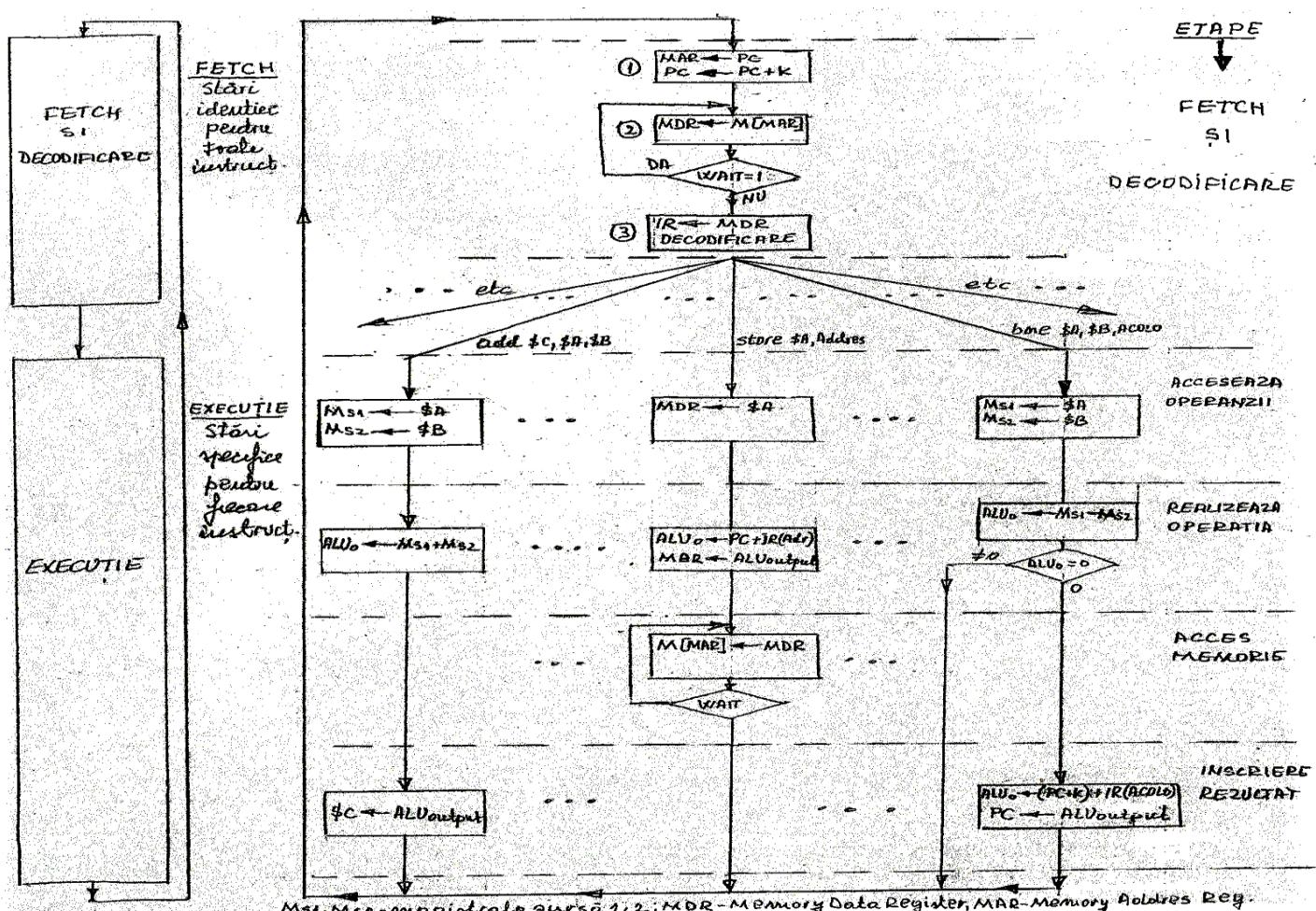
În fiecare stare i (deci pe durata a unui sau mai multor tacte de ceas) dintr-o etapă se analizează care sunt microoperațiile necesare a se efectua în calea de date, respectiv care este setul de semnale de control $\{c_{ij}\}$, $j = 1, 2, 3, \dots$ necesare a fi generate de către calea de control; într-o stare i pot fi realizate în paralel microoperațiile f_1, f_2, f_3, \dots . Folosind descrierea RTL se poate exprima funcția F_i , corespunzătoare stării i cu relația

$$\text{if } \{c_{ij}\} \text{ then } F_i = \{ Z \leftarrow f_1(X_1, X_2, \dots, X_n) \cup f_2(X_1, X_2, \dots, X_n) \cup \dots \}$$

De exemplu, în diagrama de stări prezentată în figura următoare pentru instrucțiunea $\text{add } \$c, \$a, \$b ; (\$c \leftarrow \$a + \$b)$, din etapa Realizare Operație, realizată pe o singură stare (o perioadă de ceas) este necesar a se efectua două microoperații în paralele:

1. citirea primului operand din registrul A și depunerea pe magistrala M_{S1} ;
2. citirea celui de al doilea operand din registrul B și depunerea pe magistrala M_{S2} .

Deci pentru această etapă unitatea de control trebuie să genereze setul de semnale de control $\{c_{ij}\}$ astfel ca să se realizeze funcția accesare operanți $F = \{ M_{S1} \leftarrow \$a \cup M_{S2} \leftarrow \$b \}$.



In figura următoare, ca exemplu, este prezentată diagrama de stări pentru microprocesorul MIPS, atât ca schemă bloc (Fetch + Execuției (cu cele patru ramuri corespunzătoare celor trei tipuri de instrucțiuni: R, I și J, vezi 1.7.4.)) cât și detaliată când sunt reprezentate toate stările pentru interpretarea celor trei tipuri de instrucțiuni. În fiecare stare (reprezentate prin cerculete) sunt specificate semnalele de control generate de unitatea de control.

Diagrama de stări, pentru intrepretarea instrucțiunilor, la microprocesorul MIPS

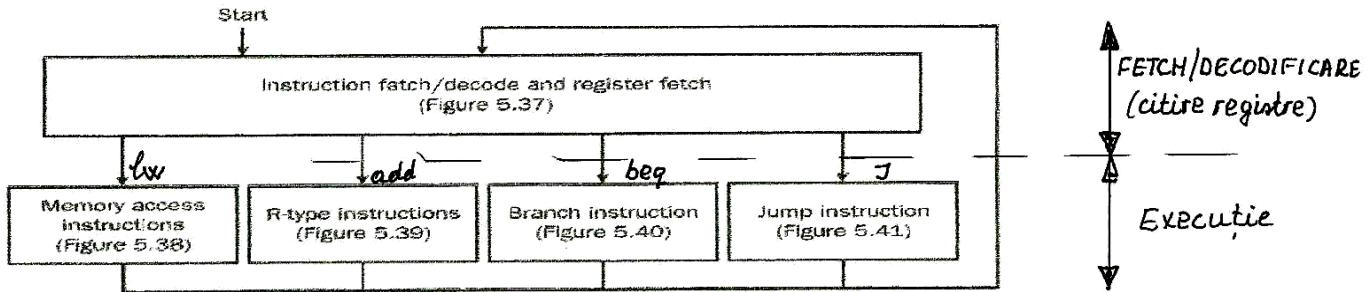
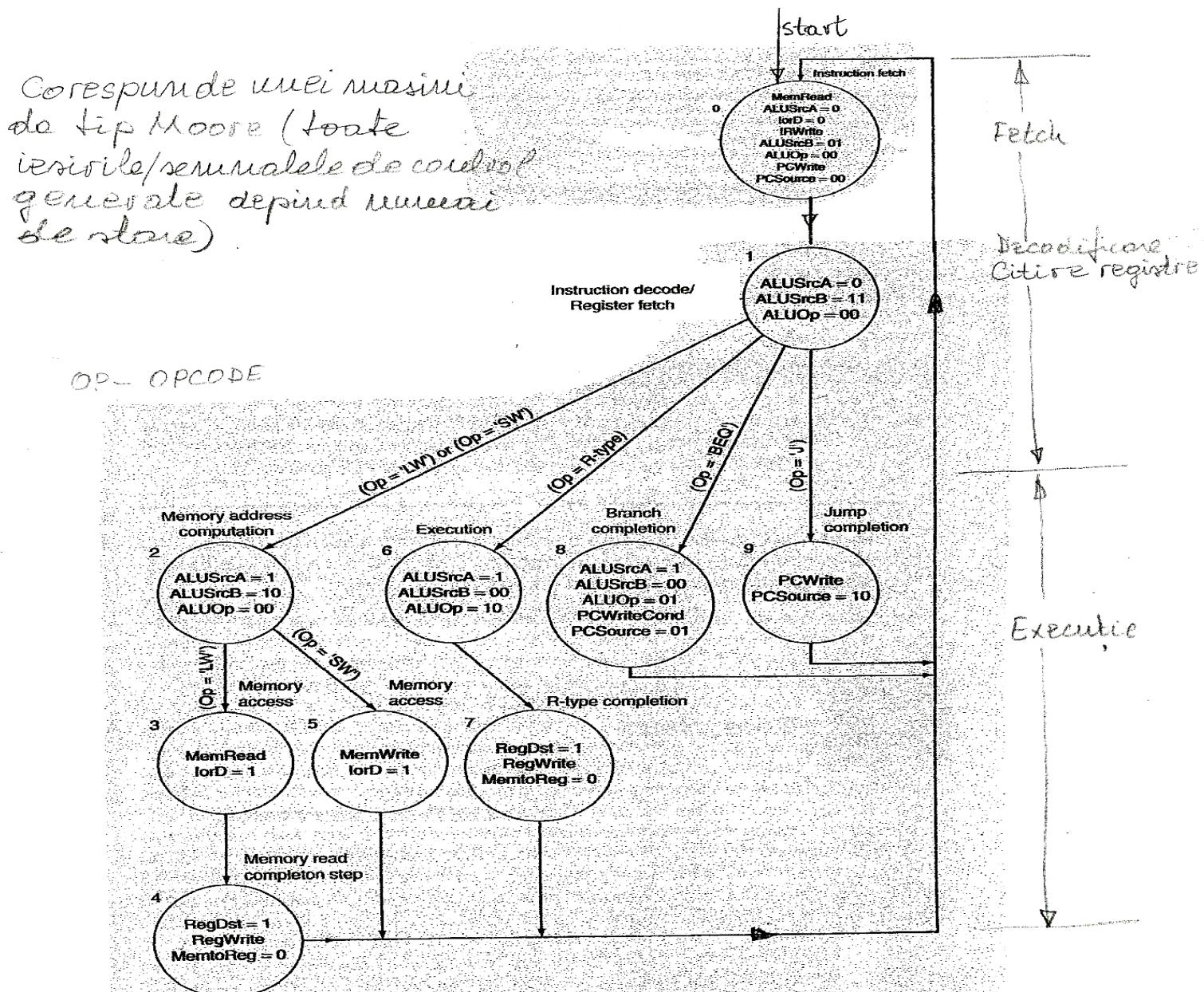


FIGURE 5.36 The high-level view of the finite state machine control. The first steps are independent of the instruction class; then a series of sequences that depend on the instruction opcode are used to complete each instruction class. After completing the actions needed for that instruction class, the control returns to fetch a new instruction. Each box in this figure may represent one to several states. The arc labeled *Start* marks the state in which to begin when the first instruction is to be fetched.

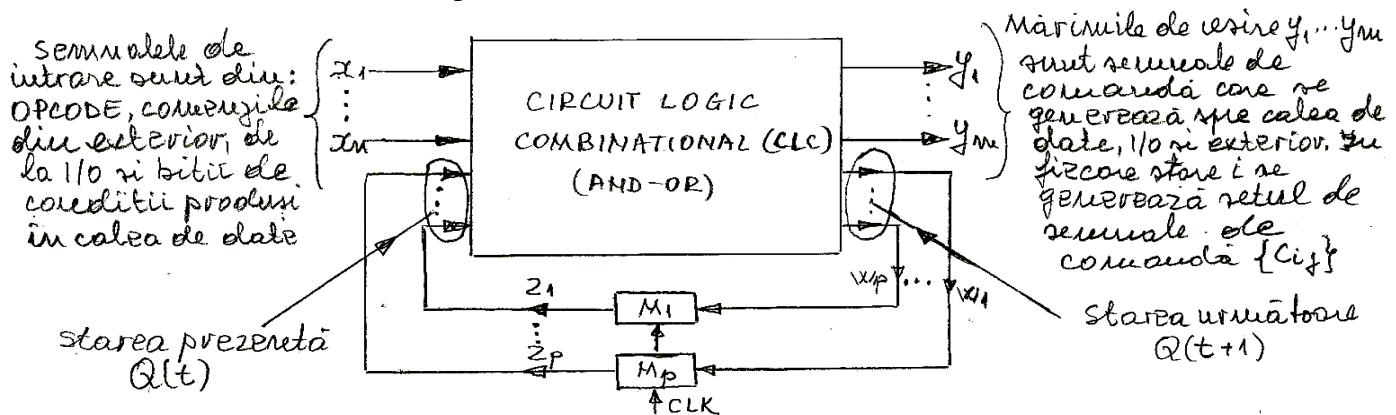


Unitatea de control poate fi implementată în două variante:

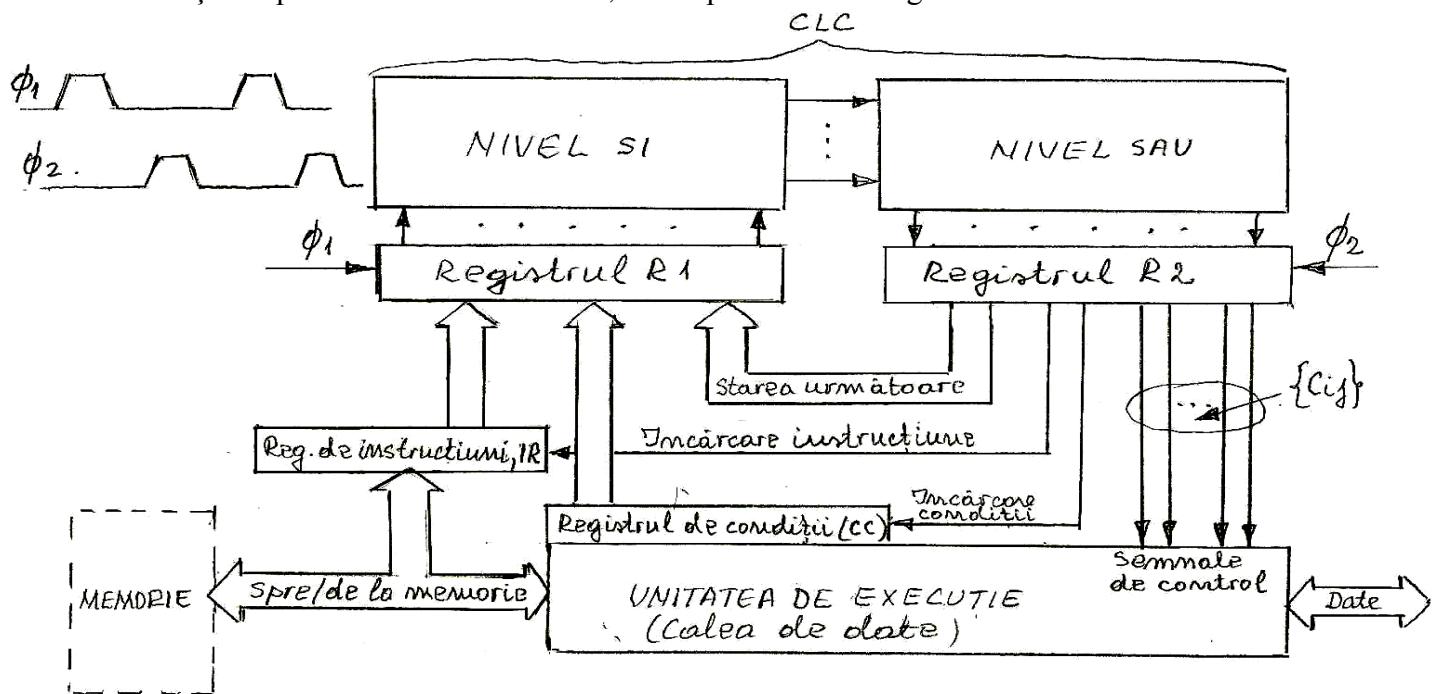
1. Unitate de control cablată;
2. Unitate de control microprogramată.

4.2. UNITATEA DE CONTROL CABLATĂ

Unitatea de control cablată este, în fond, o structură de mașină cu algoritm de stare, ASM, a cărui funcționare realizează în hard (emulează) diagrama de stări a procesorului. Structura de bază a oricărui ASM se reduce la schema Huffman de circuit secvențial prezentată în figura următoare. În această figură, de principiu, s-au introdus text explicativ încât structura de principiu de circuit secvențial să poată fi privită ca o unitate de control care generează semnalele de control $\{C_{ij}\}$ pentru o cale de date.



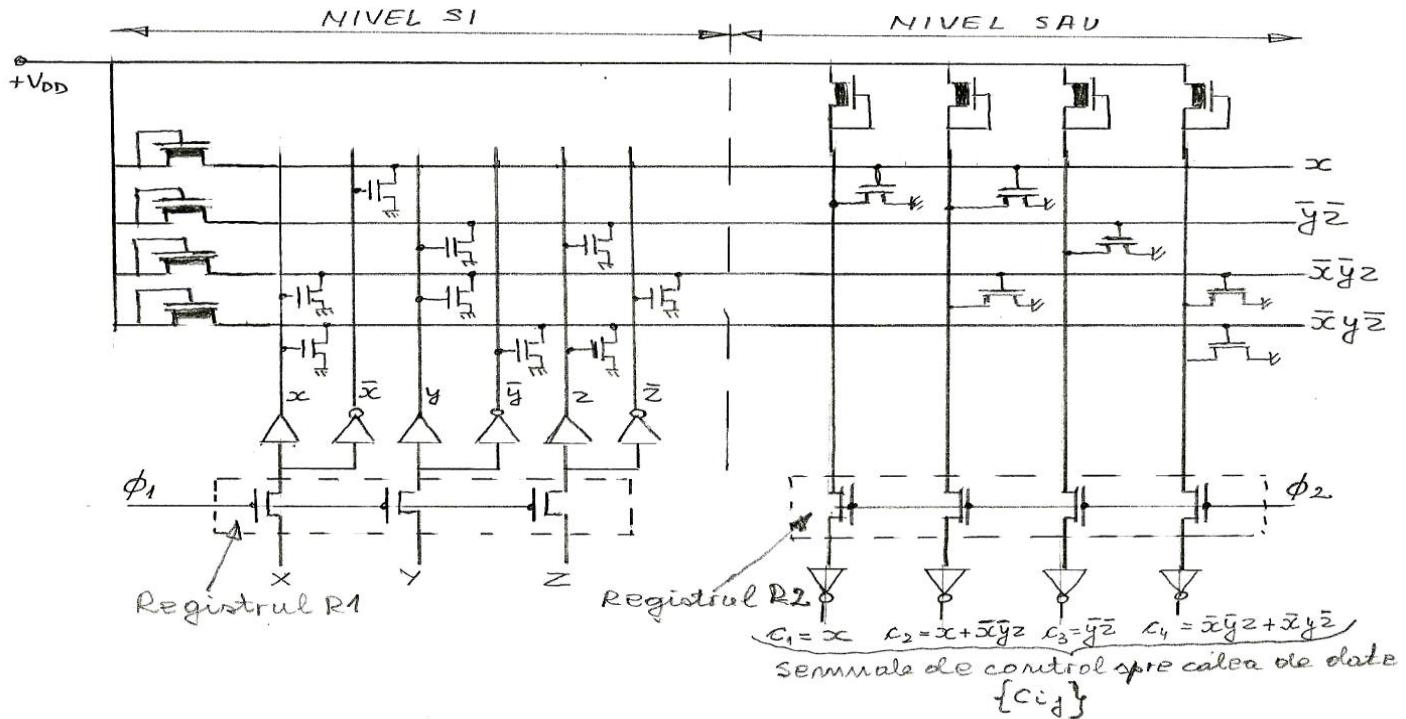
O structură de unitate de control care generează semnale pentru calea de date, realizată ca un ASM cu partea de circuit combinatorial pe bază de o matrice PLA, este reprezentată în figura următoare



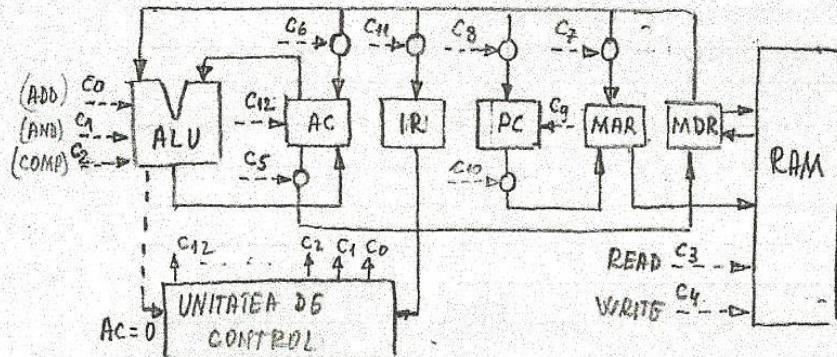
Calculul logic al semnalelor de control din calea de date, $\{C_{ij}\}$, al semnalului de încărcare a instrucțiunii în registrul de instrucții, IR și al semnalului pentru încărcarea codurilor de condiții, în registrul cc, sunt realizate pe cele două niveluri logice din PLA; de asemenea se calculează și starea următoare pentru ASM. „Cablarea” unității de control se face prin programarea matricei PLA atât pe nivelul de SI cât și pe nivelul de SAU. În figura următoare este prezentat, ca exemplu, modul de programare a unei matrice PLA de tipul NOR-NOR pentru generarea a patru semnale de control descrise de următoarele ecuații. (Primul nivel, SI, este realizat pe o matrice

NOR aplicând pe intrări variabilele negate, iar ieșirile acestuia sunt intrările nivelului SAU realizat tot pe o matrice NOR, iar ieșirile sunt apoi negate)

$$\begin{aligned}
 c_1 &= x \rightarrow \overline{\overline{x}} \\
 c_2 &= x + \overline{xyz} \rightarrow \overline{\overline{(x)} + \overline{(x+y+z)}} \\
 c_3 &= \overline{yz} \rightarrow \overline{\overline{(y+z)}} \\
 c_4 &= \overline{x}\overline{yz} + \overline{xyz} \rightarrow \overline{\overline{(x+y+\overline{z})} + \overline{(x+y+\overline{z})}}
 \end{aligned}$$



EXEMPLUL 4.2 Pentru o arhitectură pe bază de accumulator, din figura următoare, pentru care s-au figurat punctele de control c_1, \dots, c_{12} din calea de date, cu explicarea microoperațiilor corespunzătoare precum și setul de instrucțiuni care se pot executa, să se realizeze o unitate de control cablată.



MICROOPERATIILE REALIZATE DE SEMAÑALELE DE CONTROL

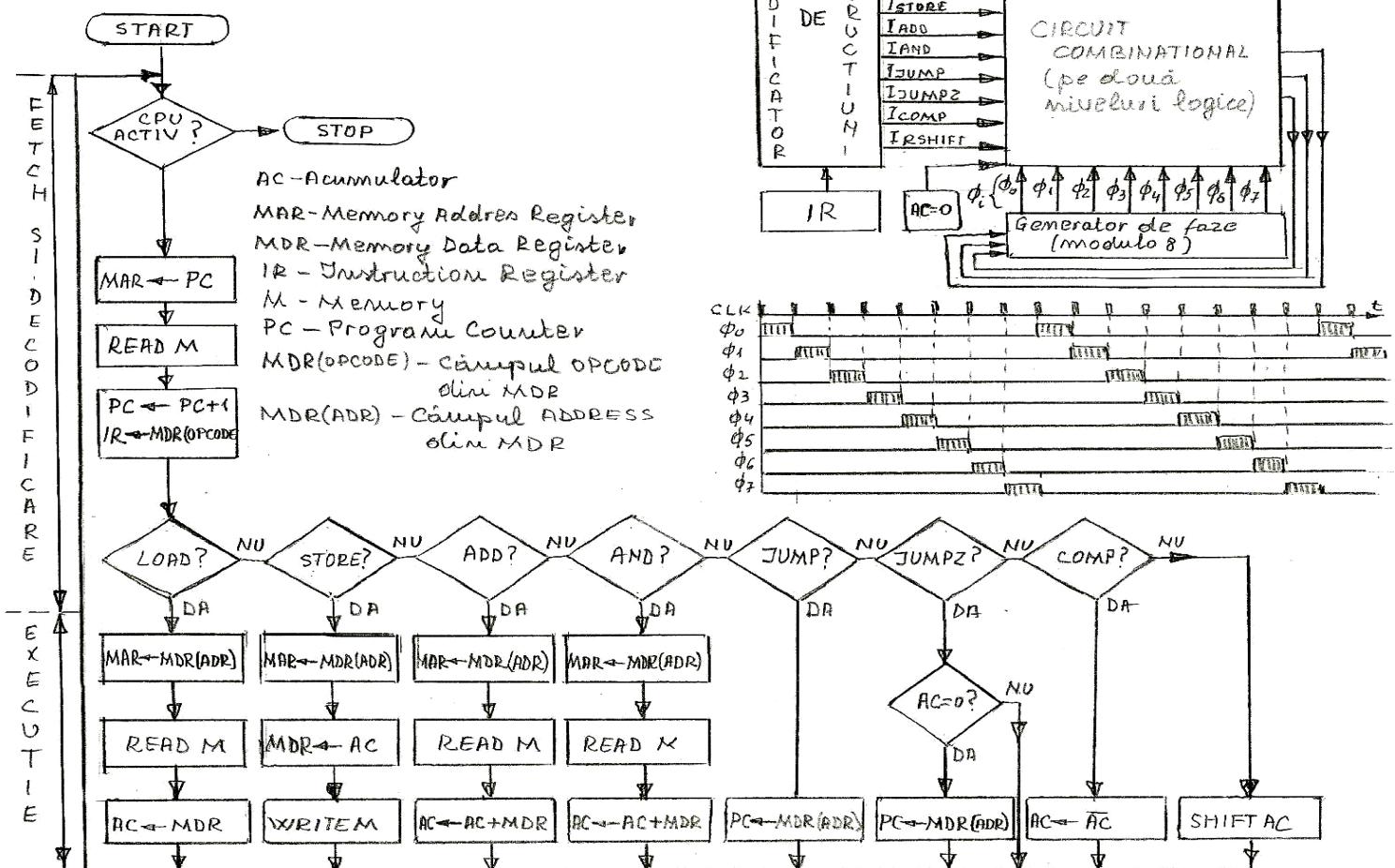
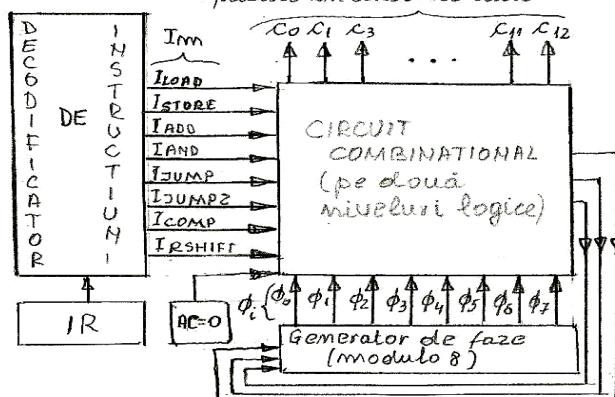
SEÑALUL DE CONTROL	OPERATIA CONTROLATÀ
C0	AC \leftarrow AC + MDR
C1	AC \leftarrow AC \cap MDR
C2	AC \leftarrow AC
C3	MDR \leftarrow M[MDR] (READ M)
C4	M[MDR] \leftarrow MDR (WRITE M)
C5	MDR \leftarrow AC
C6	AC \leftarrow MDR
C7	MAR \leftarrow MDR (ADR)
C8	PC \leftarrow MDR (ADR)
f9	PC \leftarrow PC + 1
f10	MAR \leftarrow PC
f11	IR \leftarrow MDR (OPCODE)
C12	RIGHT-SHIFT AC

SETUL DE INSTRUCȚII MNEMONIC DESCRIEREA

- LOAD X AC \leftarrow M[X] transferul conținutului locației de adresă X în acumulator
- STORE X M[X] \leftarrow AC
- ADD X AC \leftarrow AC + M[X], adunare în complement de 2
- AND X AC \leftarrow AC \cap M[X], produs logic
- JUMP X PC \leftarrow X, salt necondiționat
- JUMPZ X if AC = 0 then PC \leftarrow X, salt condiționat
- COMP AC \leftarrow AC, complementarea acumulatorului
- RSHIFT Deplasare cu un rang spre dreapta a acumulatorului

Structura UC implementată pe baza unui generator de faze

Semañale de control aplicate în calea de date



– Pentru cele opt instrucțiuni alese care formează setul de instrucțiuni al procesorului și pentru elementele componente din calea de date, urmărind toate transferurile necesare intrepretării acestor instrucțiuni, se stabilesc traseele necesare cu punctele de control și semnalele corespunzătoare, în număr de 13, c_0, \dots, c_{12}

– Se construiește diagrama de stări pentru intrepretarea setului de instrucțiuni, care cuprinde un ciclu FETCH (comun tuturor instrucțiunilor) care se realizează pe trei stări, iar în ciclul de execuție se consumă maxim trei stări, deci ar fi necesare 6 tacte de ceas pentru ciclul instrucțiune. Dar, deoarece în ciclul FETCH există totdeauna un acces la memorie iar în cel de execuție pentru unele din instrucțiuni cum ar fi : LOAD, STORE, ADD, AND există totdeauna încă un acces la memorie, accesări care se consumă pe două perioade de ceas, rezultă că intrepretarea instrucțiunilor cele mai "lente" necesită 8 tacte, (3+1) pentru Fetch + (3+1) pentru Execuție, deci în total 8 tacte.

– După această analiză a diagramei de stări pentru intrepretarea instrucțiunilor (ciclicitate în opt tacte de ceas) se alege pentru circuitul unității de control o implementare pe baza unui generator de faze modulo 8 (fazele realizate de un astfel de generator, ϕ_0, \dots, ϕ_7 , sunt reprezentate în figură). Pentru subciclul FETCH: pe ϕ_0 $MAR \leftarrow PC$, pe ϕ_1 și ϕ_2 Read Memory, pe ϕ_3 $IR \leftarrow MDR$ și $PC \leftarrow PC+1$; ϕ_4, ϕ_5, ϕ_6 și ϕ_7 sunt consumate pentru microoperațiile din subciclul de EXECUȚIE. Circuitul care calculează logic setul de semnale de control, c_1, \dots, c_{12} , care se aplică în calea de date are ca intrări:

1. cele opt faze de la generatorul de faze, ϕ_0, \dots, ϕ_7 ;
2. cele opt semnalele I_m , $m = 0, \dots, 7$, obținute de la decodificatorul codului operației din instrucțiune
3. fanionul de condiție, $AC = 0$.

Ecuația logică pentru un semnal de control c_j ($j=1, \dots, 12$) este o sumă logică a produselor dintre semnalul de fază, ϕ_i , când c_j trebuie să fie activ și fiecare dintre semnalele de la decodificator active în acea fază $\phi_i \sum_0^7 I_m$, având expresia

$$c_j = \sum_{i=0}^3 \phi_i + \sum_{i=4}^7 \phi_i \left(\sum_{m=0}^7 I_m \right) = \phi_0 + \phi_1 + \phi_2 + \phi_3 + \phi_4 \sum_{m=0}^7 I_m + \phi_5 \sum_{m=0}^7 I_m + \phi_6 \sum_{m=0}^7 I_m + \phi_7 \sum_{m=0}^7 I_m$$

De exemplu, semnalul c_3 , care este semnalul de READ MEMORY, trebuie să fie activ în starea doi și trei (ϕ_1 și ϕ_2), când se extrage instrucțiunea din memorie și în starea șase și șapte (ϕ_5 și ϕ_6), când se citește operandul din memorie (pentru instrucțiunile LOAD, ADD, AND), s-a considerat că accesul la memorie consumă două perioade de ceas, atunci din relația anterioară rezultă

$$c_3 = \phi_1 + \phi_2 + \phi_5 (I_{Load} + I_{Add} + I_{And}) + \phi_6 (I_{Load} + I_{Add} + I_{And})$$

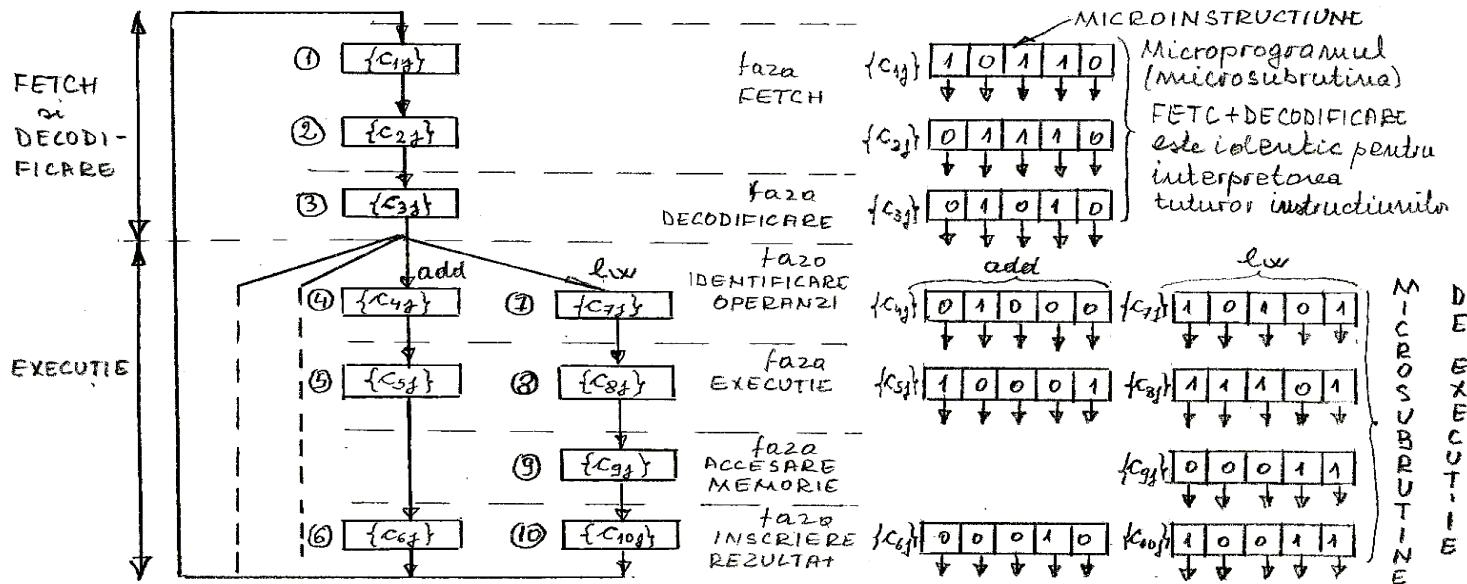
Această expresie logică se poate implementa pe un circuit cu două niveluri logice ca o sumă de produse.

4.3 UNITATEA DE CONTROL MICROPROGRAMATĂ

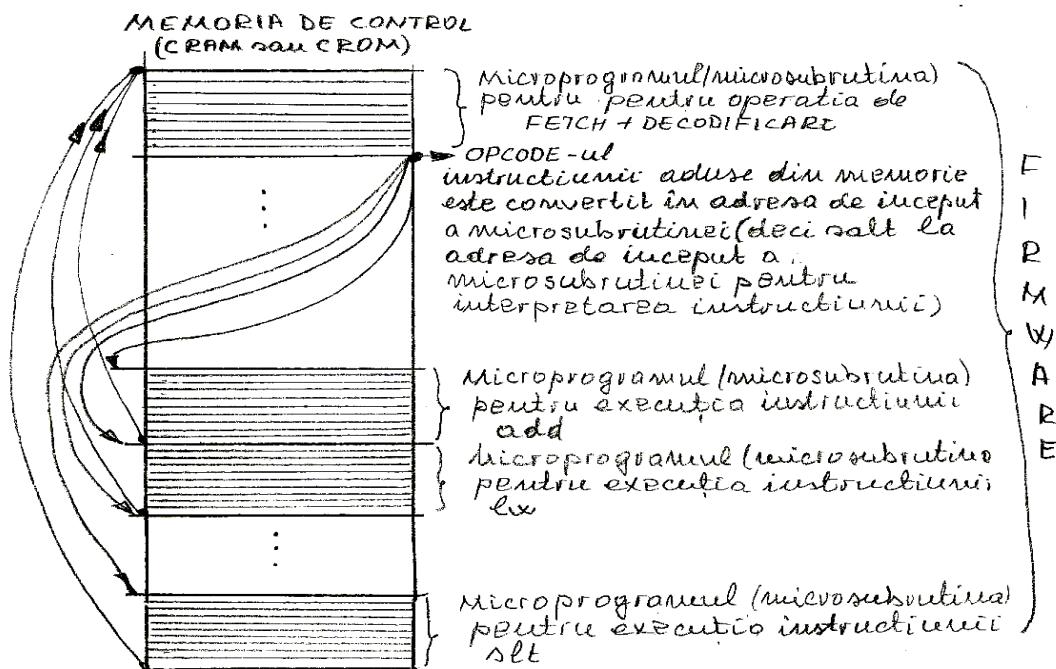
Ideea de microprogramare aparține prof. M.W. Wilkes, de la Cambridge Mathematical Laboratory, care în 1951 a implementat-o ca alternativă la complexitatea unei unități de control (UC) cablată, în tehnologia de realizare a unui procesor. De fapt, prin microprogramare prof Wilkes a construit pentru unitatea de control un procesor (de control) care generează semnalele de control, semnale ce se aplică în calea de date a procesorului, care execută instrucțiunile programului; apare, în acest fel, o recurență "procesor în procesor", procesor-unitatea de control, UC, - care comandă procesorul ce procesează programele. Similaritatea procesorului din interiorul procesorului cu procesorul care execută programul și pe care îl controlează, a dus la preluarea

termenilor de la procesor și adaptarea lor pentru procesorul din interior prin adăugarea prefixului micro, de exemplu: microoperație – MO; microinstrucțiune – μ I ; microcod; microprogram; microsubrutină; microprogram counter – μ PC; regisztr de microinstrucțiuni – μ IR etc.

Pentru explicarea tehnicii de microprogramare, în realizarea unității de control, se va utiliza reprezentarea din figura următoare. În această figură este prezentată diagrama de stare pentru două instrucțiuni, add și lw, în fiecare stare atât din ciclul Fetch + Decodificare cât și în stările din etapele (identificare operanzi, execuție, accesare memorie, înscriere rezultate) ciclului de Execuție sunt figurate semnalele de control { c_{ij} }, necesare a fi generate de unitate de control pentru a se asigura în calea de date intrepretarea instrucțiunii respective. Setul de semnale de control { c_{ij} } generate într-o anumită stare i este format dintr-un număr de biți, care pot fi strânși într-un cuvânt binar, cuvânt care formează o microinstrucțiune, deci pentru realizarea comenziilor într-o anumită stare în calea de date este necesar să se genereze această microinstrucțiune (în exemplul din figură s-a considerat pentru uniformizare și simplificare că microinstrucțiunile din fiecare stare sunt cuvinte cu lungimea de cinci biți $j=1,2,3,4,5$). De exemplu, pentru realizarea etapei de Fetch + Decodificare a oricărei instrucțiuni sunt necesare următoarele trei seturi succesive de semnale de control { c_{1j} },{ c_{2j} },{ c_{3j} }, adică de generarea succesivă a următoarelor trei cuvinte microinstrucțiune 10110, 01110, 01010, (valorile biților în aceste microinstrucțiuni sunt luate arbitrar), cuvinte care pot fi unite într-o microsubrutină, microsubrutină de interpretare a ciclului Fetch + Decodificare comună pentru toate instrucțiunile din programul rulat de procesor (ciclul de fetch este standard).



Traseul prim membrării de control pentru interpretarea instrucțiunilor



Pentru etapele din ciclul de execuție, ale instrucțiunii **lw**, este necesară să se genereze către unitatea de control următoarele seturi de semnale de control: $\{c_{7j}\}, \{c_{8j}\}, \{c_{9j}\}, \{c_{10j}\}$; adică succesiunea de cuvinte microinstructiune 10101, 11101, 00011, 10011 care pot fi strânse în microsubroutine de execuție a instrucțiunii **lw**.

Acste două microsubroutine, Fetch+Decodificare și Execuție, sunt stocate în memoria procesorului interior care constituie unitatea de control, memorie care este referată ca **Memorie de control**. Pentru interpretarea unei instrucțiuni a procesorului, procesorul de control trebuie să extragă microinstructiune după microinstructiune din memoria de control (la fel cum realizează procesorul care execută instrucțiunile programului, extrage instrucțiune după instrucțiune dar din memoria sistemului), să depună microinstructiunea într-un microregistru de microinstructiuni. Apoi, biții acestui microregistru se aplică în punctele din calea de date a procesorului, ca semnale de control, pentru a comanda realizarea **microoperațiilor** (selectare trasee, transferuri etc) corespunzătoare stării comandate de microinstructiunea respectivă. Deci pentru intrepretarea instrucțiunii programului unitatea de control extrage succesiv din memoria de control microinstructiunile microsubroutinei Fetch + Decodificare (care este comună pentru toate instrucțiunile), apoi se extrag succesiv microinstructiunile microsubroutinei de Execuție (pentru fiecare instrucțiune din ISA corespunde o microsubroutine de execuție), de exemplu, ale instrucțiunii **lw**, după care se trece la execuția instrucțiunii următoare a programului prin rularea

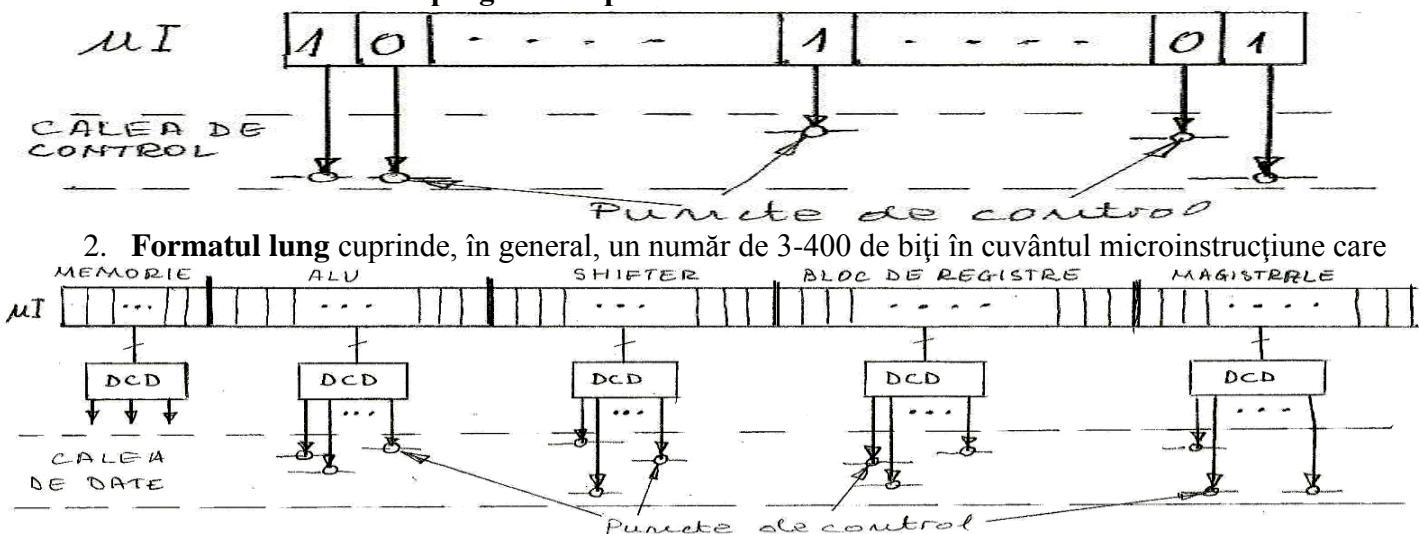
aceleiași microsubrutine Fetch + Decodificare și a microsubrutinei de Execuție corespunzătoare instrucțiunii respective; acest proces, de extragere și execuție de microinstructiuni din memoria de control, se repetă pentru fiecare instrucțiune din program rulat pe procesor.

Se pune întrebarea cum se înlănțuie (se trece) de la microsubrutina Fetch +Decodificare la microsubrutina specifică execuției instrucțiunii respective din programul de executat. Această trecere se realizează în modul următor: în momentul când s-au executat toate microinstructiunile microsubrutinei Fetch +Decodificare s-au realizat deja toate microoperațiile care au efectuat aducerea instrucțiunii programului din memoria sistemului în registrul de instrucțiuni al procesorului. Câmpul OPCODE al instrucțiunii programului, prezent acum în registrul de instrucțiuni al procesorului, este convertit în adresa din memoria de control a primei microinstructiuni din microsubrutina care va produce ciclul de execuție pentru instrucțiunea respectivă. Ultima microinstructiune din microsubrutina Fetch +Decodificare este o microinstructiune de salt în memoria de control, adresa de salt este tocmai adresa care s-a obținut din conversia OPCODE-ului instrucțiunii, deci în felul acesta microsubrutina de Fetch +Decodificare se continuă ("se leagă") în memoria de control cu microsubrutina de Execuție. Microsubrutina Execuție se termină obligatoriu cu o microinstructiune de salt necondiționat (jump) la adresa de început a microsubrutinei Fetch +Decodificare, deci se reia ciclul în memoria de control, prin această rulare ciclică în memoria de control, a microsubrutinei Fetch +Decodificare urmată de microsubrutina specifică de Execuție, se interpretează instrucțiune după instrucțiune din programul procesorului.

Microprogramul format din microsubrutina pentru ciclul Fetch + Decodificare împreună cu setul de microsubrute de execuție, specifice fiecărei instrucțiuni, înscris în memoria de control este referit prin termenul de **FIRMWARE**. Firmware-ul odată înscris în memoria de control stabilește instrucțiunile procesorului, ISA; eliminarea sau adăugarea unei microsubrute de execuție duce la eliminarea respectivă la adăugarea unei instrucțiuni în ISA. Însă utilizarea unei memorii de Control de tip RAM (CRAM) face posibilă modificarea ușoară a firmware-ului, deci a setului de instrucțiuni. Mai mult, prin schimbarea completă a firmware-ului se obține un nou procesor, un alt ISA, deci se poate "croi" un alt procesor numai prin elaborarea unui alt firmware. Această flexibilitate nu există la o unitate de control cablată, în schimb deși procesoarele cablate nu mai pot fi modificate au, în general, avantajul unor viteze de procesare mai ridicate decât cele obținute pe microprocesoarele microprogramate. Viteza de procesare mai redusă pe procesoarele cu unitate de control microprogramată rezultă din faptul că instrucțiunea din ISA este interpretată prin rularea unei succesiuni de microinstructiuni din memoria de control.

- Formatul microinstructiunilor.

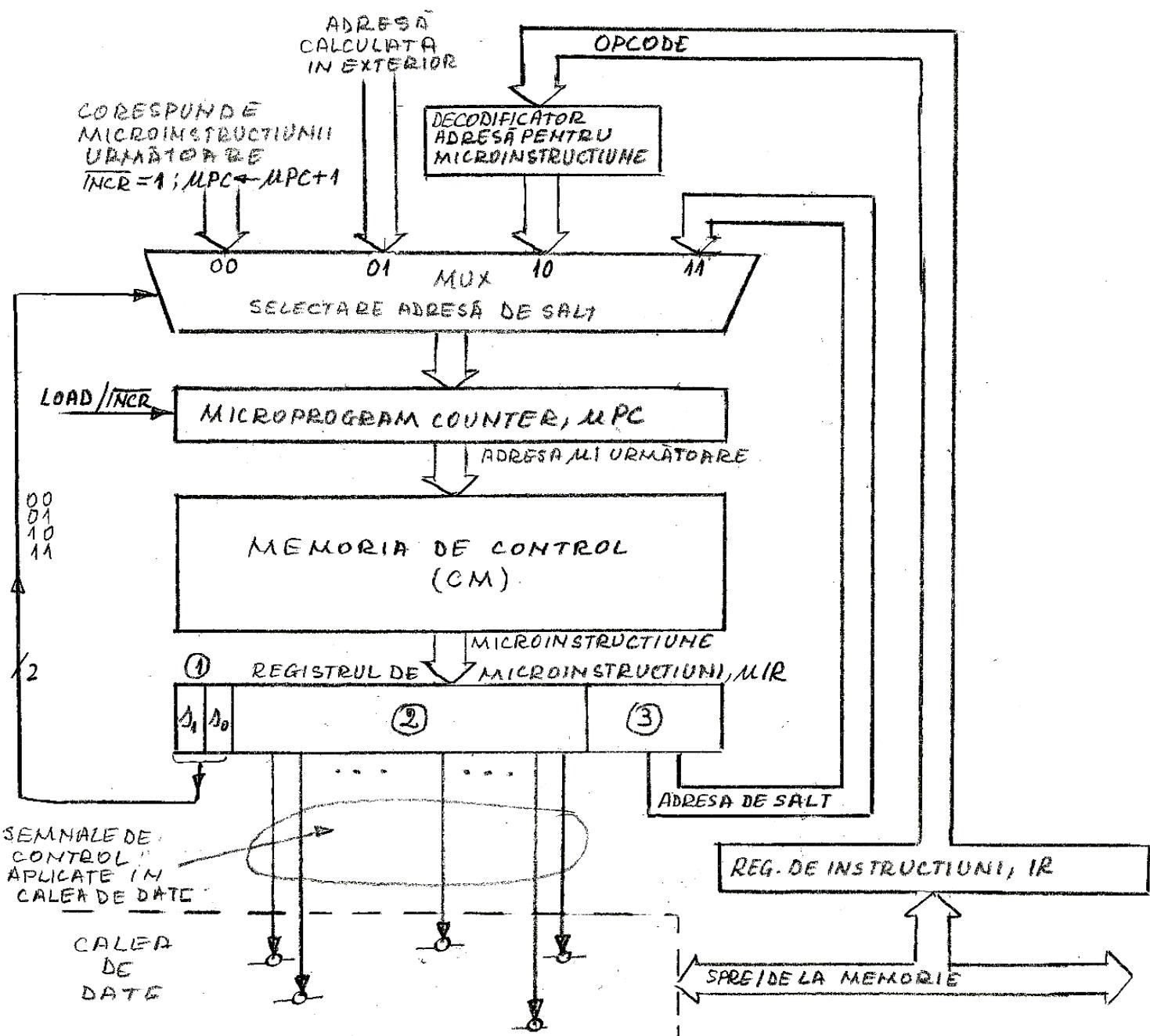
1. **Formatul scurt** cuprinde, în general, un număr de 20-30 biți în cuvântul microinstructiune, iar codificarea este liniară, adică pentru comanda unui punct din calea de date corespunde un bit în cuvântul microinstructiune. O microinstructiune, μI , de format scurt prin biții săi, ușual, realizează o microoperătie. Evident că o μI de format scurt rezultă microprograme lungi (un număr mare de microinstructiuni), mașinile cu μI de format scurt sunt referite cu **microprogramare pe verticală**.



este împărțită în subcâmpuri, pe fiecare câmp se realizează o codificare completă. Toate microoperațiile de executat pentru o anumită componentă a procesorului sunt grupate și comandate prin coduri produse pe același subcâmp al microinstructiunii, cu condiția ca microoperațiile grupate să nu fie cu execuție paralelă, adică să nu fie

executate simultan; dacă două microoperații sunt cu execuție simultană (paralelă) atunci se repartizează în subcâmpuri diferite. O μI de format lung după ce este extrasă din memoria de control și depusă în microregistru de microinstrucțiuni, fiecare câmp al său este decodificat separat, iar ieșirile active de la decodificatoare se aplică în calea de date ca semnale de control. Avantajele microprogramării cu μI de format lung sunt: viteză ridicată de interpretare (toate subcâmpurile din microinstrucțiune pot comanda în paralel), microprograme scurte (o singură μI poate interpreta operații complexe); microprogramarea cu instrucțiuni lungi este referită ca **microprogramare pe orizontală**.

- Organizarea de principiu pentru o unitate de control microprogramată este reprezentată în figura următoare. Firmware-ul corespunzător interpretării instrucțiunilor din ISA este stocat în memoria de control, CM. Conținutul registrului microprogram counter, μPC, se aplică la CM ca o adresă și din locația respectivă se extrage microinstrucțiunea, μI, și se depune în microregistru de microinstrucțiuni, μIR. Biții din μI sunt grupați în trei câmpuri:



- Câmpul 1 compus din doi biți s_{1s₂}, care specifică secvențialitatea în parcursarea microprogramului în modul următor:

– s_{1s₂} = 00, următoare μI va fi extrasă de la adresa următoare, prin incrementare adresei, INC_R = 1, din

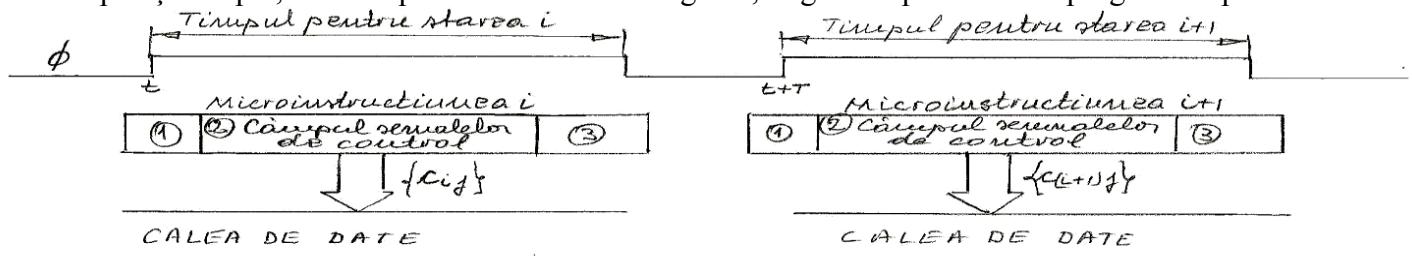
- $\mu\text{PC}, \mu\text{PC} \leftarrow \mu\text{PC} + 1;$
- $s_{1s2} = 01$, următoare μI va fi impusă din exteriorul unității de control;
 - $s_{1s2} = 10$, următoare μI se obține din OPCODE-ul instrucțiunii (la sfârșitul microsubrutinei Fetch instrucțiunea programului este deja adusă din memoria procesorului și depusă în registrul de instrucțiuni, IR. Subcâmpul codul operației, OPCODE, din IR este convertit în adresa de început din CM, unde se află microsubrutina pentru interpretarea ciclului Execuție al instrucțiunii, deci în CM se efectuează un salt necondiționat);
 - $s_{1s2} = 11$, următoare μI se află în CM la o adresă care este specificată (ca un imediat) în microinstrucțiunea din μIR deci în CM, se efectuează un salt necondiționat. (Un astfel de salt necondiționat se efectuează la sfârșitul fiecărei microsubroutine de execuție când se sare la prima microinstrucțiune din microsubrutina Fetch)

Selectarea între aceste patru variante pentru adresa μI următoare se realizează cu un grup de n MUX 4:1, comandat de cuvântul s_{1s2} .

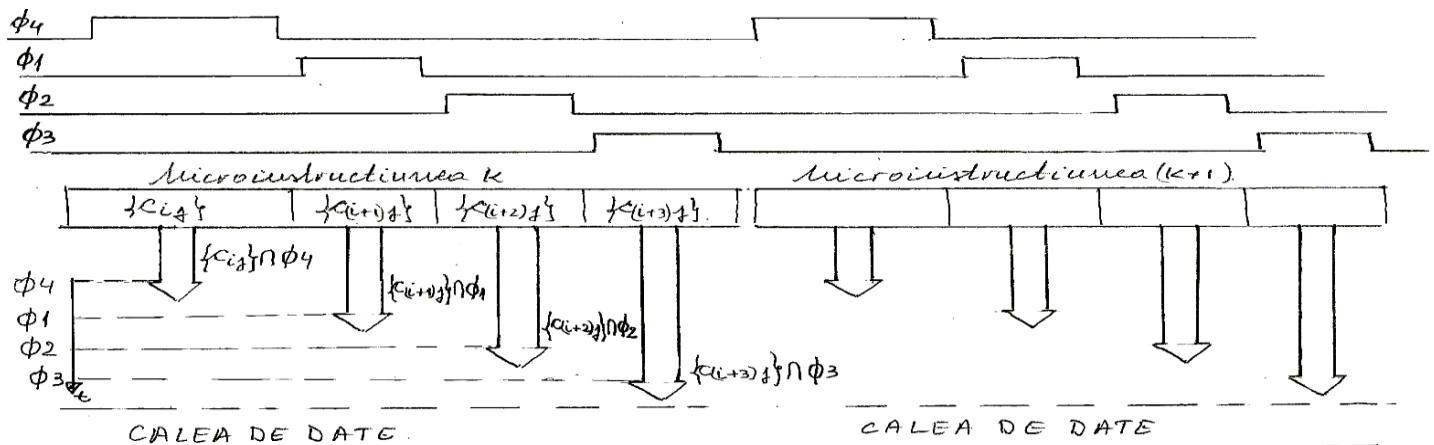
2. Câmpul 2 conține toate semnalele de control, $\{c_{ij}\}$, care se aplică în calea de date pentru comenziile efectuate de către μI respectivă.
3. Câmpul 3 conține o adresă de salt (imediat) în program, ($s_{1s2} = 11$).

- Secvențialitatea execuției microoperațiilor. Semnalele de control din microinstrucțiune comandă efectuarea anumitor microoperații în calea de date, corespunzătoare stării pentru care a fost concepută microinstrucțiunea respectivă. Generarea/activarea semnalelor de control pentru execuția microoperațiile din calea de date pot fi pe durata unui semnal de ceas (execuție monofazată) sau pe durata a mai multor semnale de ceas (execuție polifazată).

1. Comanda cu semnal de ceas monofazat. Semnale de control din corpul microinstrucțiunii, $\{c_{ij}\}$, sunt activate și aplicate în calea de date pe durata unui semnal de ceas, toate microoperațiile, pentru starea i sunt executate pe durata acelui semnal de ceas. Acest mod de executare a microoperațiilor se recomandă pentru microoperații simple, de exemplu transferuri între registre, în general pentru micropogramare pe verticală.



3. Comanda cu semnal de ceas polifazat. La micropogramarea pe orizontală, biții de control din



corpul microinstrucțiunii sunt grupați și codificați pe subcâmpuri (codificare completă și nu codificare liniară), în general un subcâmp corespunde pentru toate comenziile unui element din calea de date, iar comenziile corespunzătoare microoperațiilor din calea de date se obțin prin decodificarea codurilor din aceste subcâmpuri;

pot fi decodificate în paralel mai multe subcâmpuri ale microinstructiunii, deci se pot executa microoperații în paralel (vezi Exemplul 4.4). Pe fiecare fază a semnalului de ceas (polifazat) se generează semnale de control în calea de date, similar ca la comanda monofazată, care pot corespunde unei stări din diagrama de stări a unității de control. De exemplu, pe organizarea polifazată din figura anterioară succesiunea efectuării microoperațiilor pentru realizarea operației (instructiunii) $R_1 \leftarrow f(R_1, R_2)$ pot fi:

- pe faza $\Phi_1, \{c_{ij}\}$ se execută microoperațiile prin care se extrage din memorie instrucțiunea programului (fetch) și se generează adresa microinstructiunii următoare ;
- pe faza $\Phi_1, \{c_{(i+1)j}\}$ se execută microoperațiile prin care extrag conținuturile registrelor R_1 și R_2 și se aplică la unitatea funcțională;
- pe faza $\Phi_1, \{c_{(i+2)j}\}$ se execută microoperațiile prin care rezultatul obținut în unitatea funcțională se înscrie într-un registru temporar (latch);
- pe faza $\Phi_1, \{c_{(i+3)j}\}$ se execută microoperațiile prin care rezultatul din registru temporar este înscris în registrul R_1 .

- Micropogramarea modifică ” rigiditatea ” dintre ISA și cablajul căii de control (existent la un procesor cu UC cablat), introducând între hard și soft (limbaajul de asamblare) un nivel intremediar- Firmware, obținându-se astfel o flexibilitate. Substituind micropogramul din CM se obține o altă mașină sau o îmbunătățire a celei existente. **EMULAREA** unei mașini reprezintă simularea în hard (prin firmware) funcționarea unei mașini (adică înscrierea în CM a firmware-ului corespunzător).

- Nanopogramarea se obține prin introducerea a încă unui nivel de micropogramare între hard și CM, deci un nivel de firmware care comandă un alt nivel de firmware, de data aceasta recurența fiind un procesor (nanoprocesor) care controlează un alt procesor (cel de control, nivelul de firmware, micropogramarea) care la rândul său comandă un alt procesor (micropresorul care execută instrucțiunile programului), deci procesor – nanopresorul- în procesor –unitatea de control microprogramată - în procesor- micropresorul.

EXEMPLUL 4.3. Pentru calea de date din Exemplul 4.2, pentru care s-a elaborat o cale de control cablată, să se elaboraze o cale de control microprogramată.

Semnale de control rămân aceleiași $c_0 - c_{12}$ la care se mai adaugă c_{13} ($\mu\text{PC} \leftarrow \text{IR}(\text{OPCODE})$), prin care se comandă aplicarea la μPC a unei adrese de salt în memoria de control. Această adresă din memoria de control, care corespunde începutului microsubroutinei pentru interpretarea ciclului de execuție, este obținută din conversia OPCODE-ului înstrucțiunii din programul de executat.

Formatul microinstructiunii este un format scurt și are doar trei câmpuri:

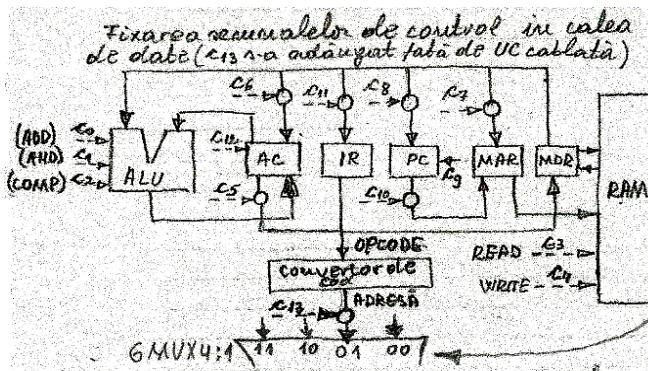
- câmpul pentru secvențialitate pe doi biți, $s_{12}00$ - adresa următoare, 01 - adresa obținută din OPCODE-ul instructiunii, 10 - salt condiționat, 11 - salt necondiționat);
- câmpul pentru adresa de salt (pe 6 biți, întreg firmware nu depășește 64 de microinstructiuni);
- câmpul pentru biții de control, în total 14 semnale de control; pentru controlul microoperațiilor din calea de date este o codificare liniară în μI , adică pentru fiecare punct de control corespunde un bit din μI .

Rezultă o microinstructiune cu lungimea de 22 biți. Setul de instrucțiuni cuprinde opt instrucțiuni:

LOAD	Adresă	; $\text{AC} \leftarrow M[\text{Adresă}]$
STORE	Adresă	; $M[\text{Adresă}] \leftarrow \text{AC}$
ADD	Adresă	; $\text{AC} \leftarrow \text{AC} + M[\text{Adresă}]$
AND	Adresă	; $\text{AC} \leftarrow \text{AC} \cap M[\text{Adresă}]$
JUMP	Adresă	; $\text{PC} \leftarrow \text{Adresă}$
JUMPZ	Adresă	; if $\text{AC} = 0$ then $\text{PC} \leftarrow \text{Adresă}$
COMP		; $\text{AC} \leftarrow \overline{\text{AC}}$
RSHIFT		; $\text{AC} \leftarrow \text{AC} \cdot 2^{-1}$

deci firmware-ul va conține opt microsubroutine de execuție, câte una pentru interpretarea ciclului de execuție al fiecarei instrucțiuni, plus microsubroutine de fetch care este comună pentru toate cele opt instrucțiuni.

Analizând pentru ciclul de fetch și pentru cele opt cicluri de execuție microoperațiile necesare din calea de date, pentru realizarea acestor cicluri, se deduc care sunt semnalele de control ce trebuie activate deci biții activi din fiecare microinstructiune (microinstructiunea este cu codificare liniară).



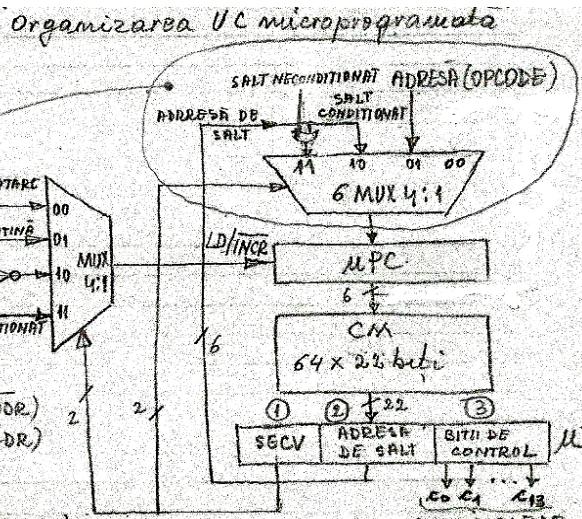
Semnalele de control

SEMNALUL MICROOPERATIA

C_0	$AC \leftarrow AC + MDR$
C_1	$AC \leftarrow AC \cap MDR$
C_2	$AC \leftarrow \bar{AC}$
C_3	$MDR \leftarrow M[MAR]$ (READ M)
C_4	$M[MAR] \leftarrow MDR$ (WRITE M)
C_5	$MDR \leftarrow AC$
C_6	$AC \leftarrow MDR$

SEMNALUL MICROOPERATIA

C_7	$MAR \leftarrow MDR(ADR)$
C_8	$PC \leftarrow MDR(ADR)$
C_9	$PC \leftarrow PC + 1$
C_{10}	$MAR \leftarrow PC$
C_{11}	$IR \leftarrow MDR(OPCODE)$
C_{12}	RIGHT-SHIFT AC
C_{13}	$MPC \leftarrow IR(OPCODE)$



Emulatorul (în limbaj de descriere) a rețelui de instrucțiuni în microcodul corespunzător

Microoperări din microsubrute

Microsubr. FETCH : $MAR \leftarrow PC$;
READ M ;
 $PC \leftarrow PC + 1$, $IR \leftarrow MDR(OPCODE)$;
go to MIR (adresa subrutinei);
se aplică la MIR)

Microsubr. LOAD : $MAR \leftarrow MDR(ADRESA)$;
READ M
 $AC \leftarrow MDR$, go to FETCH

Microsubr. STORE : $MAR \leftarrow MDR(ADRESA)$;
 $MDR \leftarrow AC$;
WRITE M, go to FETCH ;

Microsubr. ADD : $MAR \leftarrow MDR(ADRESA)$;
READ M ;
Formulă instrucțiuni: $AC \leftarrow AC + MDR$, go to FETCH;
Add ADRESĂ

Microsubr. AND : $MAR \leftarrow MDR(ADRESA)$;
READ M ;
 $AC \leftarrow AC \cap MDR$, go to FETCH ;

Microsubr. JUMP : $PC \leftarrow MDR(ADRESA)$, go to FETCH ;

→ JUMPAZ : If $AC = 0$ then go to FETCH ;
 $PC \leftarrow MDR(ADRESA)$, go to FETCH ;

COMP : $AC \leftarrow \bar{AC}$, go to FETCH ;

RSHIFT : RIGHT-SHIFT(AC), go to FETCH ;

M3	S0 S1	ADRESA DE SALT		C0, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13
		00	01	
1	00	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1 0 0 0 0
2	00	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0
3	00	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	0 1 0 0 0
4	01	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 1

1	00	ADRESA DE SALT		C0, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13
		00	01	
1	00	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2	00	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3	11	ADRESA FETCH	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1	00	ADRESA DE SALT		C0, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13
		00	01	
1	00	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2	00	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3	11	ADRESA FETCH	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1	00	ADRESA DE SALT		C0, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13
		00	01	
1	00	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2	00	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3	11	ADRESA FETCH	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1	00	ADRESA DE SALT		C0, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13
		00	01	
1	00	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2	00	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3	11	ADRESA FETCH	0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Micooperațiile necesare și succesiunea acestora comandate de microsubrutina Fetch sunt:

Fetch: $MAR \leftarrow PC$; comandat de activarea $c_{10} = 1$, $s_{1s2} = 00$

RED M ; comandat de activarea $c_3 = 1$, $s_{1s2} = 00$

$PC \leftarrow PC + 1$, $IR \leftarrow MDR(OPCODE)$; comandat de activarea $c_9 = 1$, $c_{11} = 1$ (aceste două micooperații se efectuează în paralel), $s_{1s2} = 00$

Go to μIR ; comandat de activarea $c_{13} = 1$, $s_{1s2} = 01$ (OPCODE transformat în adresa de început a microsubroutinei de execuție se înscrise în μIR).

Rezultă microsubrutina fetch compusă din patru microinstructiuni, iar după executarea acesteia, prin ultima microinstructiune, se efectuează salt la prima microinstructiune din una dintre microsubrutele de execuție. Pentru fiecare ciclu de execuție al celor opt instructiuni din ISA, similar ca și la subrutina Fetch, se deduc care sunt microoperațiile necesare și care este succesiunea lor, apoi biții cuvintelor microinstructiune apar evident prin identificarea semnalelor de control ce trebuie activate pentru microoperațiile respective.

Flexibilitatea oferită de micropogramarea unității de control se poate evidenția prin următoarea exemplificare. Să presupunem că în ISA s-a omis includerea unei instructiuni, pe care să o denumim CLEAR, a cărei funcție este de a reseta toți biții cuvântului din acumulator (în lipsa unui semnal de control de RESETARE ACUMULATOR).

Resetarea acumulatorului (înscrierea cu zero) poate fi rezultatul următoarei operați logice $A \cap \bar{A} = 0$.

Microoperațiile necesare a se realiza de către microsubrutina de execuție a instructiunii CLEAR, precum și biții (semnalele de control) din microinstructiunile respective sunt prezentate în figura următoare

MICROSUB. CLEAR: $MDR \leftarrow AC$

$AC \leftarrow \bar{A}$

$AC \leftarrow AC \cap MDR, go to FETCH$

b_1	b_2	ADR. DE SALT	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

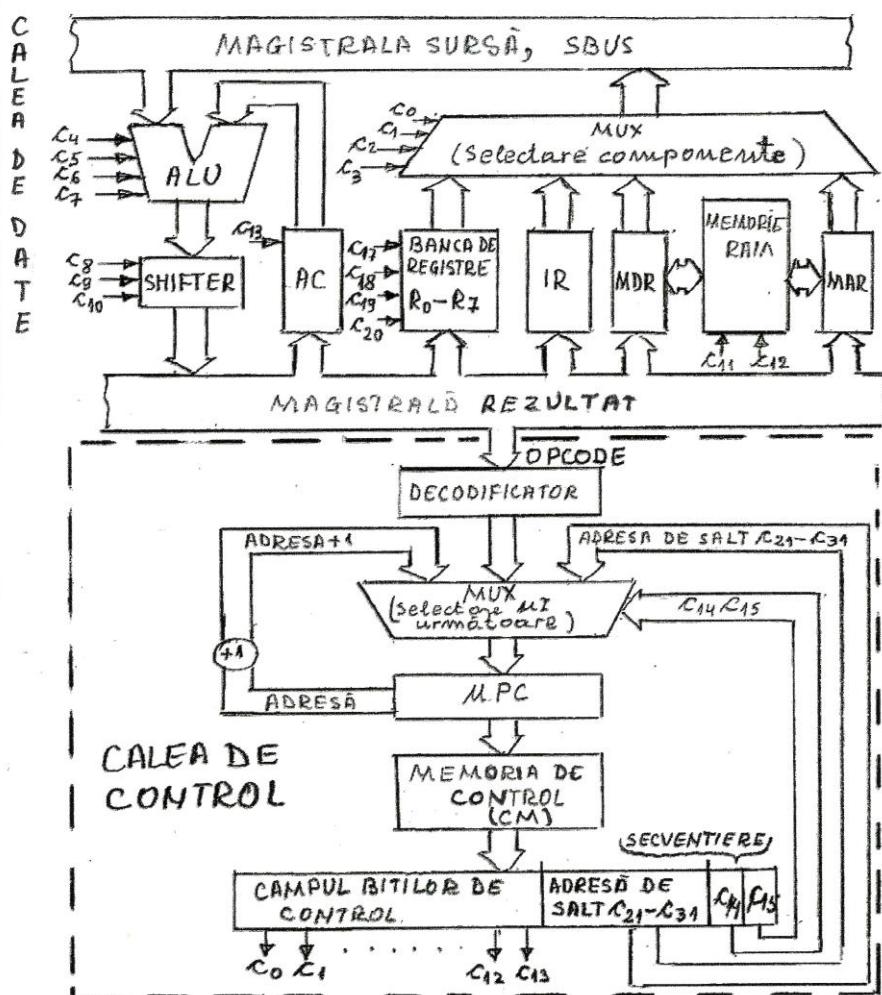
EXEMPLUL 4.4. Pentru calea de date reprezentată în figura următoare să se facă sinteza unei unități de control micropogramată.

Arhitectura acestui procesor este pe bază de acumulator (AC), ca și în Exemplul 4.3 diferență constând în faptul că aici există și o bancă de registre, deci apar aspecte de selectare a regiszrelor. Calea de date este organizată pe două magistrale: magistrașa sursă - SBus și magistrașa rezultat. În calea de date sunt următoarele componente:

- unitate aritmetică și logică, ALU, poate fi comandată pentru 16 operații aritmetice/ logice cu patru semnale de control, $c_0 - c_3$;
- registrul acumulator, AC, comandat pentru înscriere cu rezultatul din ALU cu semnalul $c_{13} = 1$;
- unitate de shiftare, SHIFTRE, cuvântul se poate deplasa stânga/dreapta logic sau aritmetic, rotație stânga/dreapta cu o poziție, este controlată de semnalele $c_8 - c_{10}$;
- blocul de 8 registre, R_i , $0 \leq i \leq 7$, registrul R_0 este utilizat ca program counter (PC), semnalele de control pentru înscrierea regiszrelor sunt $c_{17} - c_{20}$, corespunzând codurile $c_{17}c_{18}c_{19}c_{20}$ de la 0000 la 0111 (codurile până la 1111 sunt pentru celelalte regiszre din calea de date);
- registrul de instrucții, IR, în care se depune instrucția de program adusă din memorie, apoi OPCODE- instrucției este transferat (prin decodificare) în adresa microinstructiunii de început din microsubrutina pentru ciclul de execuție al instrucției. Înscrierea sa se realizează prin cuvântul de control $c_{17}c_{18}c_{19}c_{20} = 1000$;
- registrul buffer pentru adresarea memoriei, MAR (are rol numai electric nu și logic). Înscrierea se realizează prin cuvântul de control $c_{17}c_{18}c_{19}c_{20} = 1001$.
- registrul buffer de date de la/înspre memorie, MDR (are rol numai electric nu și logic). Înscrierea se realizează prin cuvântul de control $c_{17}c_{18}c_{19}c_{20} = 1010$;
- multiplexorul MUX prin care se selecteză elementele din calea de date pentru înscrierea pe magistrașa SBus, selectarea se efectuează prin semnalele de comandă c_0, c_1, c_2, c_3 ;
- decodificatorul transferului codului obținut din OPCODE spre unitatea de control, aplicarea cuvântului la intrarea sa se face prin cuvântul de control $c_{17}c_{18}c_{19}c_{20} = 1011$;
- semnalele de lucru cu memoria sunt: $c_{11}c_{12} = 10$ pentru citire, $MDR \leftarrow M[MAR]$ și $c_{11}c_{12} = 11$ pentru înscriere, $M[MAR] \leftarrow MDR$;

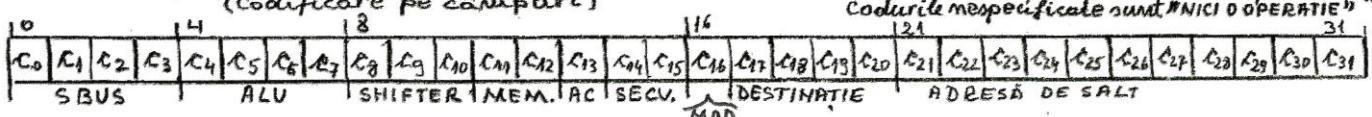
Unitatea de control micropogramată are o organizare similară cu organizarea de principiu prezentată anterior. Formatul microinstructiunii este de 32 biți cu codificare pe câmpuri (un câmp este asignat pentru codificarea comenziilor unui element din calea de date) și execuție polifazată (patru faze). Microsubrutina pentru ciclul Fetch și cele pentru ciclurile execuție ale instrucțiunilor rezultă din descrierea tuturor microoperațiilor pentru efectuarea

ciclului respectiv, apoi prin asignarea pentru aceste microoperații a unor valori de cod în câmpurile microinstrucțiunii.

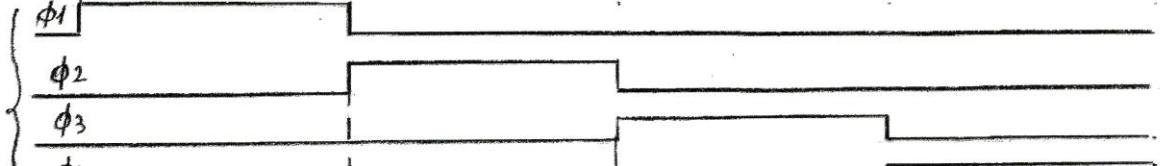


Pentru microoperatie se aleg urmatoarele campuri cu urmatoarele codificari Campul assignat unui element si colecturile (daca reccurent)		Microoperatie efectuata:
1	ALU \leftarrow AC	
2	ALU \leftarrow SBUS	
3	ALU \leftarrow AC + SBUS	
4	ALU \leftarrow AC - SBUS	
5	ALU \leftarrow SBUS - AC	
6	ALU \leftarrow AC + SBUS	
7	ALU \leftarrow AC - SBUS	
8	ALU \leftarrow SBUS	
9	ALU \leftarrow SBUS + 1	
10	ALU \leftarrow AC + 1	
11	ALU \leftarrow 0	
12	ALU \leftarrow 1	
MEMORIE C11, C12		MDR \leftarrow M[MAE] M[MAR] \leftarrow MDR
0	$OSi \leq 7$	SBUS \leftarrow R _i
1	SBUS	SBUS \leftarrow IR
2	C ₀ - C ₃	SBUS \leftarrow MAR
3		SBUS \leftarrow MDR
AC (C₁₃) C ₈ - C ₁₀		AC \leftarrow SHIFTER 0 SHIFTER \leftarrow (ALU) 1 SHIFTER \leftarrow deplasare stanga logica (ALU)
4		2 SHIFTER \leftarrow deplasare dreapta logica (ALU)
5		3 SHIFTER \leftarrow rotatie dr (ALU)
6		4 SHIFTER \leftarrow rotatie stg (ALU)
7		5 SHIFTER \leftarrow deplasare stanga aritmetica (ALU)
8		6 SHIFTER \leftarrow deplasare dreapta aritmetica (ALU)
SECVENTIERE C ₁₆ MOD=0		Urmatoarea MI MPC \leftarrow Decodificator(OPCODE) pt. if CARRY=0 then MPC \leftarrow ADRESA
9		11* if ZERO=1 then MPC \leftarrow ADRESA
10		12* MPC \leftarrow ADRESA
11		13* DCD \leftarrow SHIFTER
DESTINATIE C ₁₇ - C ₂₀		7 R _i \leftarrow SHIFTER 8 IR \leftarrow SHIFTER 9 MAR \leftarrow SHIFTER 10 MDR \leftarrow SHIFTER 11 DCD \leftarrow SHIFTER
*Conditii CARRY si ZERO se inverseaza pt MOD=1, (C ₁₆) Codurile nespecificate sunt "NICI O OPERATIE"		

FORMATUL MICROINSTRUCȚIUNII (32 biți)
(Codificare pe câmpuri)



Fazele pentru executia unei MI



Accele microoperații pot fi activate numai pe durata fazei pentru care au fost asignate, de exemplu comanda pentru ALU și obtinere primă conjectură dintr-o acastă comandă în phi2

SBUS \leftarrow MDR;
 SBUS \leftarrow MAR;
 SBUS \leftarrow IR;
 SBUS \leftarrow R_i; OSi ≤ 7
 (valioate prim $\phi_1 = 1$)

Operatii ALU;
 optional citire/înscriere memorie cu adaugare stari de WAIT
 (Validate prim $\phi_2 = 1$)

Shiftare;
 Calculu adresei urmatoare
 (Validate prim $\phi_3 = 1$)

MAR \leftarrow SHIFTER;
 MDR \leftarrow SHIFTER;
 IR \leftarrow SHIFTER;
 DCD \leftarrow SHIFTER;
 Optimal (AC \leftarrow SHIFTER);
 (Validate prim $\phi_4 = 1$)

Se va exemplifica pentru microsubrutina Fetch. Nu se va elabora instrucțiunile pentru ISA, cititorul poate să își definească propriul său set de instrucțiuni pentru acest procesor.

- Microsubrutina Fetch

– operațiile efectuate în ciclul Fetch:

$IR \leftarrow M[R_0]$; R_0 din banca de registre s-a asignat ca program counter, PC.

$R_0 \leftarrow R_0 + 1$;

– descompunerea operațiilor în microoperații sucesiune lor fiind validate cu fazele $\Phi_1, \Phi_2, \Phi_3, \Phi_4$,

Microinstruc. ui	Microoperații efectuate în fazele microinstrucțiunii			
	Φ_1	Φ_2	Φ_3	Φ_4
1	$SBUS \leftarrow R_0$	$ALU \leftarrow SBUS$	$SHIFTER \leftarrow ALU$ $UPC \leftarrow UPC + 1$	$MAR \leftarrow SHIFTER$
2	$SBUS \leftarrow R_0$	$ALU \leftarrow SBUS + 1$ $MDR \leftarrow M[MAR]$	$SHIFTER \leftarrow ALU$ $UPC \leftarrow UPC + 1$	$R_0 \leftarrow SHIFTER$
3	$SBUS \leftarrow MDR$	$ALU \leftarrow SBUS$	$SHIFTER \leftarrow ALU$ $UPC \leftarrow UPC + 1$	$IR \leftarrow SHIFTER$
4	$SBUS \leftarrow IR$	$ALU \leftarrow SBUS$	$SHIFTER \leftarrow ALU$	$DCD \leftarrow SHIFTER(OPCODE)$

Pentru aceste microoperații corespund în cadrul următoarelor valori ale coloanei (în zecimale)

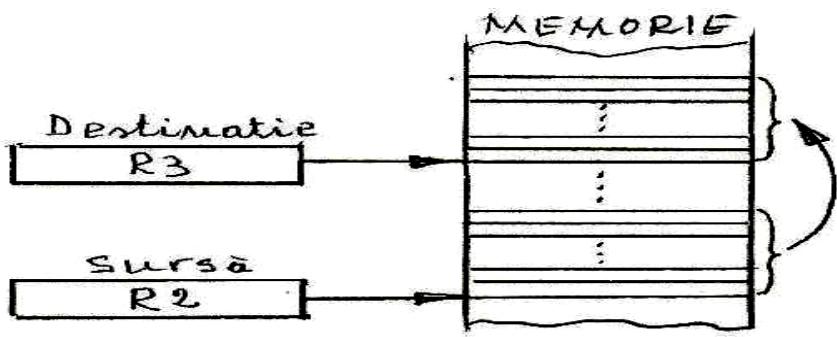
ui	cărți		SBUS	ALU	SHIFTER	MEMORIE	AC	SEQUENTA	MOD	DESTINATIE	ADRESA DE SALT
	1	2									
1	0	2	0	0	0	0	0	0	9	0	
2	0	9	0	2	0	0	0	0	0	0	
3	10	2	0	0	0	0	0	0	8	0	
4	8	2	0	0	0	0	1	0	11	0	

- Microsubrutină de execuție. Se consideră, ca exemplu, o instrucțiune de înalt nivel, **deplasează bloc**, care transferă un bloc de cuvinte dintr-o zonă de memorie (sursă) într-o altă zonă de memorie (destinație). Înainte de interpretarea instrucțiunii deplasează bloc, adresa de început din zona de memorie sursă, As, este încărcată în registrul (pointer) R2, adresa de început din zona de destinație, Ad, este încărcată în registrul (pointer) R3, iar dimensiunea n a blocului de transferat a fost încărcată în registrul (contor) R1. Transferul cuvânt după cuvânt se repetă atât timp cât este adevărată condiția $R_1 > 0$ (contorul nu a ajuns la zero), deci o buclă DO-WHILE. Considerând sintaxa instrucțiunii deplasează bloc, **moveb R₃,R₂,R₁**, programul se poate scrie astfel:

```

addi $R2, $zero, As ; $R2 ← As (adresa sursă)
addi $R3, $zero, Ad ; $R3 ← Ad (adresa destinație)
addi $R1, $zero, n ; $R1 ← n (contor)
moveb R3,R2,R1 ; M[R3] ← M[R2] de [R1] ori

```



Microoperațiile pentru microsubrutina de execuție a instrucțiunii moveb R₃, R₂, R₁, sunt:

1. while R₁ > 0 do begin
 2. R₁ ← R₁ - 1 ;
 3. MAR ← R₂ ;
 4. R₂ ← R₂ + 1 ; executie în paralel
 5. MDR ← M[MAR] ; executie în paralel
 6. MAR ← R₃ ;
 7. R₃ ← R₃ + 1 ; executie în paralel
 8. M[MAR] ← MDR ; executie în paralel
- end while

Microinstr. MI	Microoperatiile efectuate în fazele microinstructiunii			
	φ1	φ2	φ3	φ4
1	SBUS ← R ₁	ALU ← SBUS	if ZERO=1 then MPC ← end ADDRESS	
2	SBUS ← R ₁	ALU ← SBUS - 1	SHIFTER ← ALU, MPC ← MPC + 1	R ₁ ← SHIFTER
3	SBUS ← R ₂	ALU ← SBUS	SHIFTER ← ALU, MPC ← MPC + 1	MAR ← SHIFTER
4	SBUS ← R ₂	ALU ← SBUS + 1 MDR ← M[MAR]	SHIFTER ← ALU, MPC ← MPC + 1	R ₂ ← SHIFTER
5	SBUS ← R ₃	ALU ← SBUS	SHIFTER ← ALU, MPC ← MPC + 1	MAR ← SHIFTER
6	SBUS ← R ₃	ALU ← SBUS + 1 M[MAR] ← MDR	SHIFTER ← ALU, MPC ← begin ADDRESS	R ₃ ← SHIFTER

Pentru aceste microoperatii corespund in cimpurile microinstructiunii urmatoarele valori de cod (in decimal)

MI \ CAMPURI	SBUS	ALU	SHIFTER	MEMORIE	AC	SECU	MOD	DEST	ADRESA DE SALT
1	1	2	0	0	0	3	0	11	end ADDRESS = Adresa de inceput in microsub. FETCh
2	1	9	0	0	0	0	0	1	0
3	2	2	0	0	0	0	0	9	0
4	2	2	0	2	0	0	0	2	0
5	3	2	0	0	0	0	0	9	0
6	3	9	0	3	0	3	1	3	begin ADDRESS = ADRESA de inceput din acelă microsub.

CAP 5. TEHNICI ȘI STRUCTURI PENTRU CREȘTEREA PERFORMANȚELOR

5.1 PROCESAREA DE TIP PIPELINE

5.1.1. Organizarea de principiu pentru un pipeline

Pentru oricare circuit logic combinațional, CLC, se presupune că are la intrare un registru de intrare, care conține variabilele de intrare, $x_0, x_1, \dots, x_{k-2}, x_{k-1}$ și la ieșire un registru de ieșire în care se vor înscrie valorile calculate ale variabilelor de ieșire $y_0, y_1, \dots, y_{p-2}, y_{p-1}$, ca în figura a următoare. Considerând că timpul de calcul (de propagare) pe CLC este τ_{pCLC} , pe cele două registre (considerate identice) timpul de set-up este τ_{SU} , iar cel de propagare este τ_{pR} , rezultă că frecvența maximă de ceas, $f_{\min} = 1/T_{CLK(\min)}$, cu care se poate comanda aplicarea variabilele de intrare respectiv se pot înscrie rezultatele la ieșire, se calculează cu relația

$$T_{CLK(\min)} \geq \tau_{pR} + \tau_{pCLC} + \tau_{SU}$$

Când se inseriază n circuite logice secvențiale, ca în figura b, registrul de ieșire al circuitului CLCi devinind registrul de intrare al circuitului CLCi+1, se obține o structură de procesare de tip pipeline (conductă, l.engl.). Valorile variabilelor de ieșire, $y_0, y_1, \dots, y_{p-2}, y_{p-1}$, se obțin din variabilele de intrare, $x_0, x_1, \dots, x_{k-2}, x_{k-1}$, prin calcule succesive pe cele n circuite combinaționale componente din pipeline. Fiecare din cele n circuite componente pot fi identice dar programabile (Mux, PLA), deci fiecare dintre acestea se programă conform funcției logice, OPi, corespunzător procesării etapei i din pipeline.

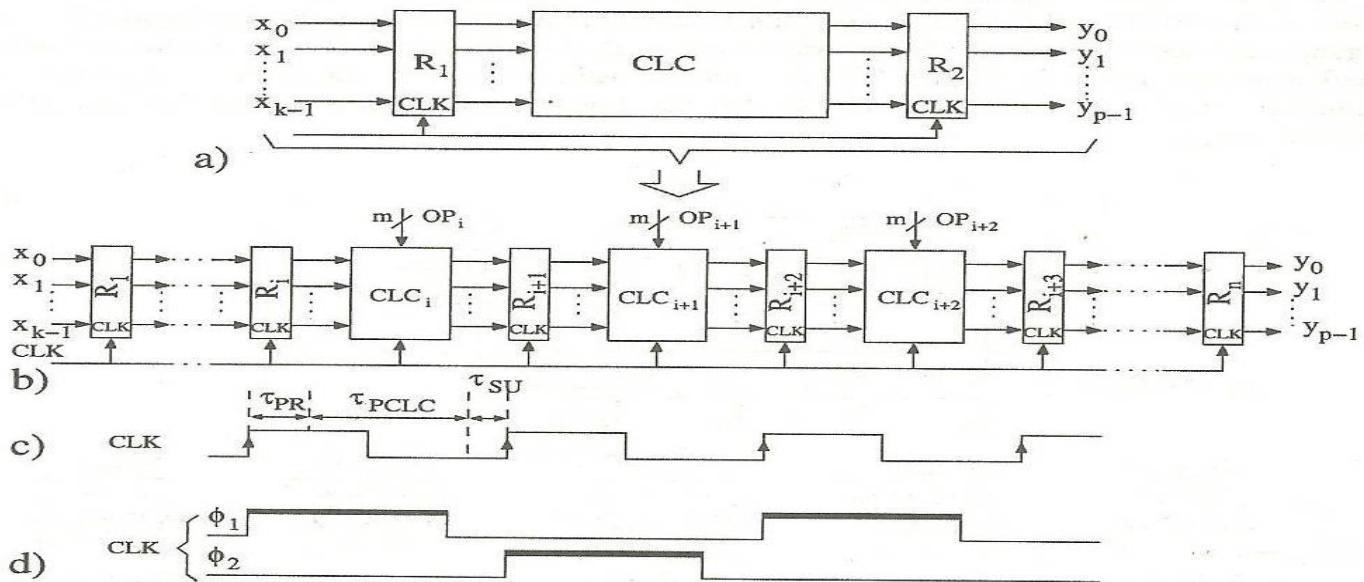


Figura 3.76 Organizarea de tip pipeline: a) structurarea pentru o procesare de tip non-pipe (clasică); b) structura unui pipe cu n etape de procesare; semnale de ceas pentru sincronizarea in-pipe (c) cu o singură fază (pe front) și (d) cu două faze Φ_1 și Φ_2 (pe palier)

Perioada minimă a ceasului cu care se comandă înscrierea regiszrelor pipe (dintre două CLC-uri vecine), inclusiv regiszrul de intrare și de ieșire, se calculează cu relația

$$\tau_{CLK(\min)} \geq \tau_{pR} + \max\{\tau_{pCLC1}, \tau_{pCLC2}, \dots, \tau_{pCLCn}\} + \tau_{SU} + \tau_{skew}$$

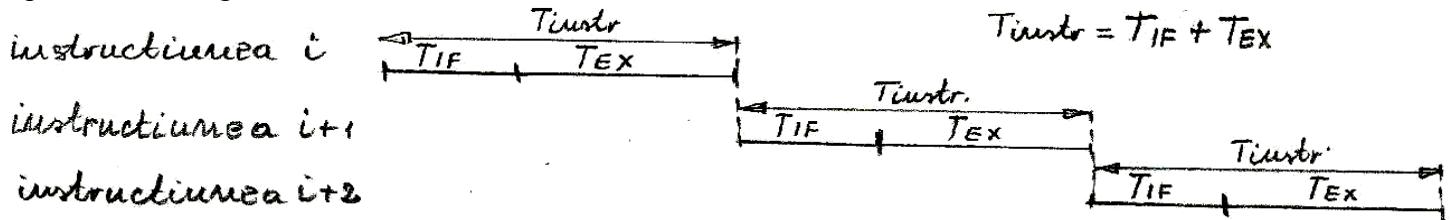
τ_{skew} - fiind timpul de defazaj al semnalului de ceas între primul și ultimul regiszru.

Perioada minimă a semnalului de ceas, este determinată de circuitul component din pipe care are timpul de propagare cel mai lung.

De la momentrul t_0 , al înscrerii unor valori pentru variabilele de intrare în registrul de intrare, R_1 , până la înscrerea valorilor calculate în registrul de ieșire, în momentul $t_1 + n \cdot T_{CLK}$, se aplică n tacte de ceas, rezultă că propagarea prin pipeline (**latența pipe-ului**) este $n \cdot T_{CLK}$. Dacă dacă acum se consideră că pe fiecare impuls de ceas se înscrise un set de valori pentru variabilele de intrare în registrul de intrare, R_1 , după o întârzire egală cu latența pipe ($n \cdot T_{CLK}$) se obține primul set de valori calculate pentru variabilele de ieșire, apoi, în continuare pe fiecare tact de ceas se obține câte un set de valori calculate pentru variabilele de ieșire; deci pe fiecare tact se aplică un set de valori pe intrare și se obține un set de valori calculate pe ieșire.

Procesarea de tip pipeline - Noțiuni fundamentale

- Procesarea secvențială a instrucțiunilor (clasică, non-pipe), instrucțiune-după-instrucțiune, este reprezentată în figura următoare



Pentru o procesare secvențială, cu timpul de execuție instrucțiune $T_{instr} = T_{IF} + T_{EX}$, pentru un program de N instrucțiuni, timpul total consumat de procesor este egal cu

$$CPU_{time} = N_{instr} \cdot CPI \cdot T_{CLK} = N_{instr} \cdot T_{instr}$$

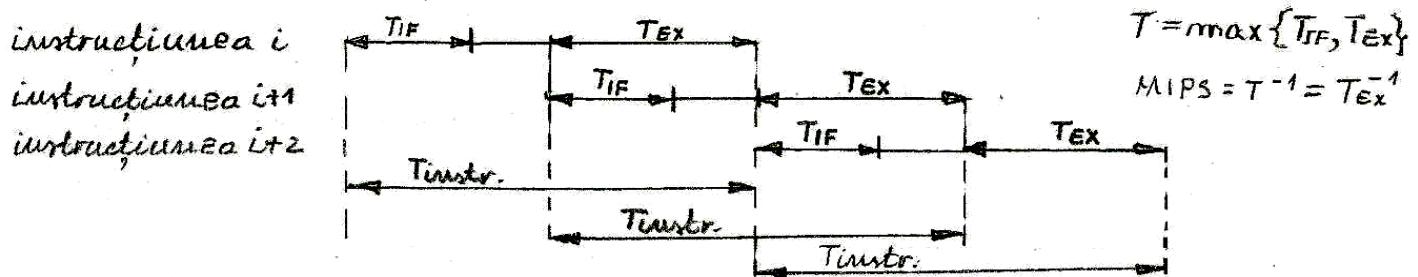
în care: $CPI = \text{cicluri}/(\text{tacte de ceas consumate})$ pe instrucțiune

T_{CLK} – perioada semnalului de ceas

$$T_{instr} = CPI \cdot T_{CLK}$$

$$MIPS = \frac{N_{instr} \cdot 10^{-6}}{CPU_{time}} = \frac{N_{instr} \cdot 10^{-6}}{N_{instr} \cdot CPI \cdot T_{CLK}} = \frac{10^{-6}}{CPI \cdot T_{CLK}} = \frac{10^{-6}}{\frac{T_{instr}}{T_{instr}} \left[\frac{s}{instr} \right]} = T_{instr}^{-1} \left[\frac{instr}{\mu s} \right]$$

- Procesarea de tip pipeline. Tehnica de procesare tip pipeline este o modalitate în executarea instrucțiunilor prin care fiecare instrucțiune este divizată în n părți (etape) distințe și executarea fiecărei etape se realizează în același timp (în paralel, cu alte etape de la alte instrucțiuni). Ce mai simplă modalitate de procesare pipeline a șirului de instrucțiuni dintr-un program se obține când fiecare instrucțiune este divizată în cele două părți, etapa FETCH și etapa Execuție, iar în rularea programului se suprapune realizarea etapei execuție de la instrucțiunea i cu efectuarea etapei fetch de la instrucțiunea următoare, $i+1$, ca în figura următoare ; ușual $T_{IF} < T_{EX}$.

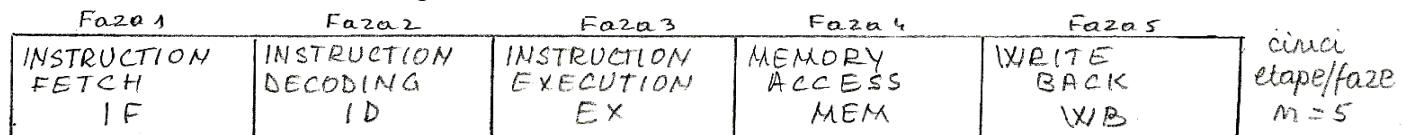


Deși timpul efectiv de execuție pentru o singură instrucțiune a crescut de la $T_{instr} = T_{IF} + T_{EX}$ la $T_{instr} = T_{EX} + T_{IF} = 2 \cdot T_{EX}$ timpul de execuție în pipe pentru o instrucțiune este mai mic, egal cu T_{EX} , iar $MIPS = T_{EX}^{-1}$.

Trecând de la divizarea instrucțiunii numai în cele două etape fech și execuție la o divizare în mai multe părți, de exemplu în n părți/etape/faze (T_{instr}/n pe o etapă) atunci teoretic viteza de procesare în pipe ar crește de

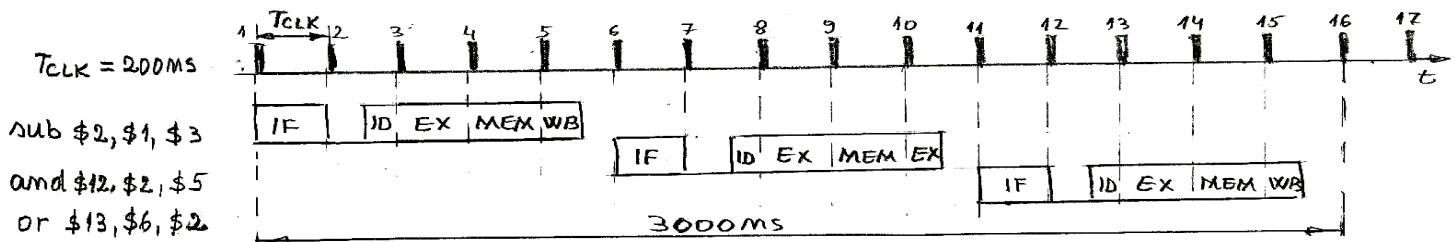
n ori conform relației anterioare $MIPS = (T_{instr} / n)^{-1}$. Uzual, pentru procesarea pipe clasică, instrucțiunea este divizată în cinci părți (etape/faze) cum este prezentată în figura următoare cu semnificațiile:

- IF – citirea din memorie a instrucțiunii și aducerea în procesor, **Instruction Fetch**;
- ID – accesarea și citirea operanzzilor, decodificarea OPCODE-ului, **Instruction Decoding**;
- EX – realizarea operației specificată în Opcode, **Execution**;
- MEM – citirea sau înscrierea datelor în memorie (dacă este cazul), **MEMory Access**;
- WB – încrierea rezultatului în registru, **Write Back**.

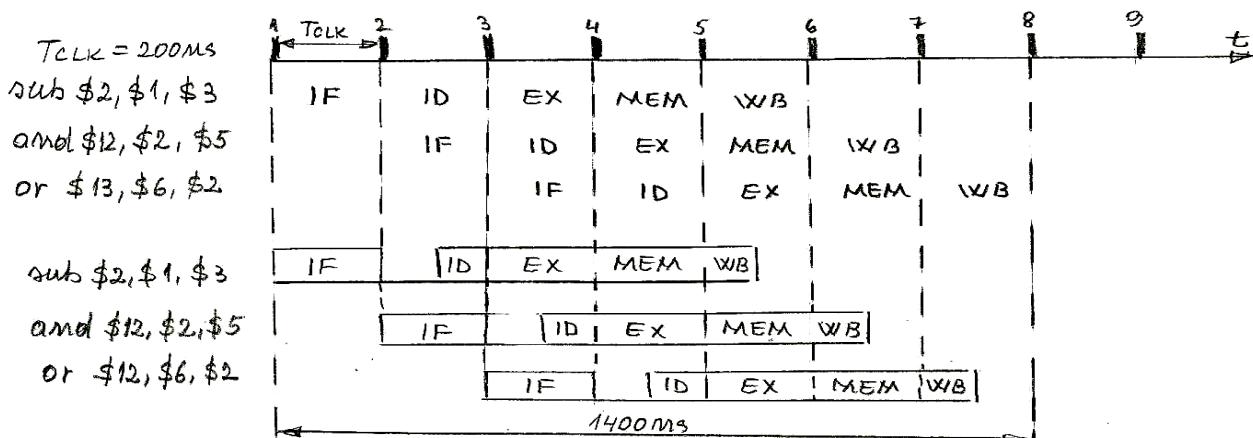


Divizarea instrucțiunii în faze care să ducă la tempi de procesare egali pentru fiecare fază nu este totdeauna posibil, totuși la procesarea în pipe se alocă un timp egal pentru toate fazele, acel timp este egal cu timpul necesar pentru procesarea fazei celei mai lungi. De exemplu, în divizarea anterioară, etapa ID și WB realizează citirea respectiv înscrierea unor registre, acest timp de citire sau de înscriere este mai scurt decât timpul necesar pentru celelalte faze (IF, EX, MEM). Citirea și înscrierea registrelor se poate realiza chiar într-o jumătate de perioadă de ceas, TCLK (în reprezentările care urmează aceste două faze sunt reprezentate doar pe jumătate din perioada de ceas, pe prima jumătate a perioadei de ceas se poate înscrie un registru, iar citirea se poate realiza pe a doua jumătate a perioadei de ceas). În figurile următoare este reprezentată, prin comparație, procesarea de tip clasic (secvențial) și procesare de tip pipeline pentru un segment de program de trei instrucțiuni, fiecare instrucțiune divizată în cince etape. Pentru procesarea de tip pipe s-a considerat că deja pipe-ul este plin și pe fiecare tact intră o instrucțiune și se generează rezultatul unei instrucțiuni procesate. Timpul necesar pentru procesarea secvențială este de 3000ns iar pentru procesarea pipe rezultă 1400ns.

PROCESARE SECVENTIALA



PROCESARE DE TIP PIPELINE



Creșterea (teoretică) de viteza la procesarea de tip pipe se poate determina în modul următor

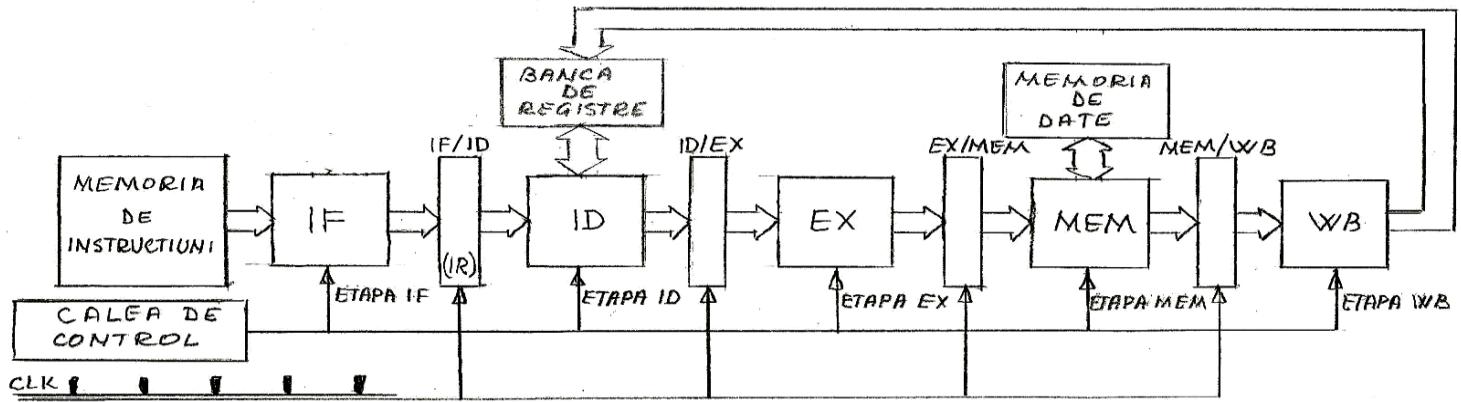
$$\text{Cresterea de viteza, } CV = \frac{\text{Timpul de procesare non - pipe}}{\text{Timpul de procesare pipe}} = \frac{N_{instr} \cdot n \cdot T_{CLK}}{n \cdot T_{CLK} + (N_{instr} - 1)T_{CLK}} = \frac{n \cdot N_{instr}}{(n + N_{instr} - 1)}$$

iar la limită când numărul de instrucțiuni, N_{instr} , ale programului procesat $\rightarrow \infty$ rezultă

$$CV = \left(\frac{n \cdot N_{\text{instr}}}{n + N_{\text{instr}} - 1} \right)_{N_{\text{instr}} \rightarrow \infty} = n$$

deci o creștere de viteza de n ori pentru un pipe cu n etape/faze (la o instrucțiune divizată în n părți). Prin execuția de tip pipeline nu se micșorează timpul de execuție al unei instrucțiuni, T_{instr} , dimpotrivă acesta crește puțin (cu regia pipe-ului) în raport cu timpul de execuție non-pipe al unei instrucțiuni, ceea ce crește este numărul de instrucțiuni (n) care se execută în unitatea de timp după ce pipe-ul este umplut (prin execuția în paralel de n etape diferite de la n instrucțiuni diferite).

- *Strucurarea de unei căi de date pipelinezată.* În figura următoare este reprezentată o cale de date, de tip pipeline, cu cinci etape (specificate anterior).



Între circuistica (circuite de tip CLC) fiecare etape din pipe sunt registre pipe notate cu abreviațiile etapelor vecine (IF/ID, ID/EX, EX/MEM, MEM/WB). La aplicarea unui impuls de ceas rezultat prin procesarea de la etapa anterioară se înscrie în registrul pipe și constituie semnalele de intrare în circuistica combinațională a etapei următoare până la aplicare următorului impuls de ceas.

Alegerea numărului n de etape de pipe în calea de date depinde de:

- raportul dintre timpul de acces la memorie supra timpului de acces la elementele din calea de date
- arhitectura setului de instrucțiuni
- frecvența semnalului de ceas.

Creșterea de viteza, teoretic, rezultă prin implementarea unei căi de date cu un număr n sporit de etape, acesta ar duce la o creștere de viteza de n ori în raport cu o cale de date cu procesare non-pipe; o cale de date cu un n ridicat, peste 10, caracterizează **procesoarele superpipelinizate**. De asemenea, creșterea de viteza este determinată pe lângă creșterea numărului de etape și prin micșorarea perioadei de ceas în care se efectuează o etapă, adică odată cu creștere a frecvenței de ceas. Aceste două modalități, mărirea numărului de etape și creșterea frecvenței de ceas, teoretic, ar duce la creșterea vitezei de procesare. Practic, prin aceste două modalități nu se obține o creștere liniară nelimitată de viteza. De la anumite valori în sus ale numărului de etape în pipe, n , se constată, dimpotrivă, o stagnare (explicabilă) a creșterii de viteza, mai mult chiar o scădere; astfel un număr mai mare de $n=10$ etape se întâlnește mai rar, iar frecvența de ceas se pare că pentru majoritatea implementărilor nu va mai depăși 3-3,5 GHz (din cauza puterii disipate).

5.1.3 Procesarea în pipeline la procesorul MIPS

Calea de date la MIPS este organizată sub forma unui pipeline cu cinci etape: IF, ID, EX, MEM, WB în care sunt procesate instrucțiunile de tipul: R, I și J, prezentate ca format în figura următoare. Fiecare etapă se desfășoară pe durata unei perioade de ceas, TCLK. Se vor descrie în continuare microoperațiile care se realizează în fiecare din cele cinci etape.

1. *Etapa IF (Instruction Fetch).* După aplicarea primului impuls de ceas, pe durata TCLK, conținutul registrului program counter (PC) se aplică la memoria de instrucțiuni, se extrage instrucțiunea care urmează să fie procesată, totodată pe sumatorul dedicat calculului adresei următoare, NPC (Next Program Counter) se realizează sumarea pentru determinarea adresei instrucțiunii următoare $NPC = PC + 4$ (Next Program Counter).

După intervalul TCLK, se aplică al doilea impus de ceas, pe al cărui front pozitiv în registrul pipe IF/ID (care poate fi considerat ca echivalentul registrului de instrucțiuni, IR, dintr-o structură non-pipe)

R-type	OPCODE	r_s	r_t	r_d	shamt	funct.	0
add	\$rd, \$rs, \$rt					$\begin{array}{l} \$rd \leftarrow \$rs + \$rt \\ ; \$rd \leftarrow \$rs \wedge \$rt \end{array}$	
and	\$rd, \$rs, \$rt					$\begin{array}{l} \$rd \leftarrow \$rs \wedge \$rt \\ ; if (\$rs < \$rt) atunci \$rd \leftarrow 1 \end{array}$	set less
slt	\$rd, \$rs, \$rt					$\begin{array}{l} if (\$rs < \$rt) atunci \$rd \leftarrow 1 \\ ; than \end{array}$	
I-type	OPCODE	r_s	r_t	[I ₁₅]	Imm/IMM (16 biti)	[I ₀]	0
addi	\$rt, \$rs, Imm					$\begin{array}{l} \$rt \leftarrow \$rs + \{ [I_{15}]^{16} \times \times [I_{15} \div I_0] \} \\ ; \$rt \leftarrow M[\$rs + \{ [I_{15}]^{16} \times \times [I_{15} \div I_0] \}] \end{array}$	adunare
lw	\$rt, Imm(\$rs)					$\begin{array}{l} \$rt \leftarrow M[\$rs + \{ [I_{15}]^{16} \times \times [I_{15} \div I_0] \}] \\ ; \$rt \leftarrow M[\$rs + \{ [I_{15}]^{16} \times \times [I_{15} \div I_0] \}] \end{array}$	incarcare
beq	\$rs, \$rt, Imm					$\begin{array}{l} if (\$rs == \$rt) atunci PC \leftarrow (PC+4) + (Imm \times 4) \\ ; if (\$rs == \$rt) atunci PC \leftarrow (PC+4) + (Imm \times 4) \end{array}$	salt
J-type	OPCODE	I ₂₅			address (26 biti)		I ₀
					$J \text{ address} ; PC \leftarrow (PC_{31} \div PC_{23}) \times \times \times [I_{25} \div I_0] \times 2^2$		

OPERĂRI LEZĂRĂTE ÎN FIECARE ETAPĂ DE PIPE PENTRU CELE TREI TIPI DE INSTRUCȚIUNI

1	2	3	4	5
ID/IF $\leftarrow M[PC]$	A $\leftarrow [\$r_s]$ B $\leftarrow [\$r_t]$ NPC $\leftarrow NPC$ Im = $[I_{15}]^{16} \times \times \times [I_{15} \div I_0]$ ID/EX $\leftarrow [I_{20} \div I_{16}]$ ID/EX $\leftarrow [I_{15} \div I_0]$	ALU rez $\leftarrow A \text{ op } B$ rd $\leftarrow [I_{15} \div I_{17}]$ ALU rez $\leftarrow A \text{ op } Im$ rt $\leftarrow [I_{20} \div I_{16}]$ ALU rez $\leftarrow A \text{ op } Im$ rt $\leftarrow [I_{20} \div I_{16}]$ Comd $\leftarrow A \text{ op zero}$ EX/MEM $\leftarrow NPC + Im$		rd $\leftarrow ALU \text{ rezultat}$ rt $\leftarrow ALU \text{ rezultat}$ MEM/WB $\leftarrow M[ALU \text{ rez}]$ M[ALU rez] $\leftarrow B \text{ (sw)}$ if Comd = 1 atunci PC $\leftarrow NPC + Im$ if Comd #1 atunci PC $\leftarrow NPC$
PC $\leftarrow PC + 4$				
ID/IF $\leftarrow NPC = PC + 4$				
CLK	1	2	3	4

Selectarea adresei pentru instrucțiunea următoare

Adresa de salt

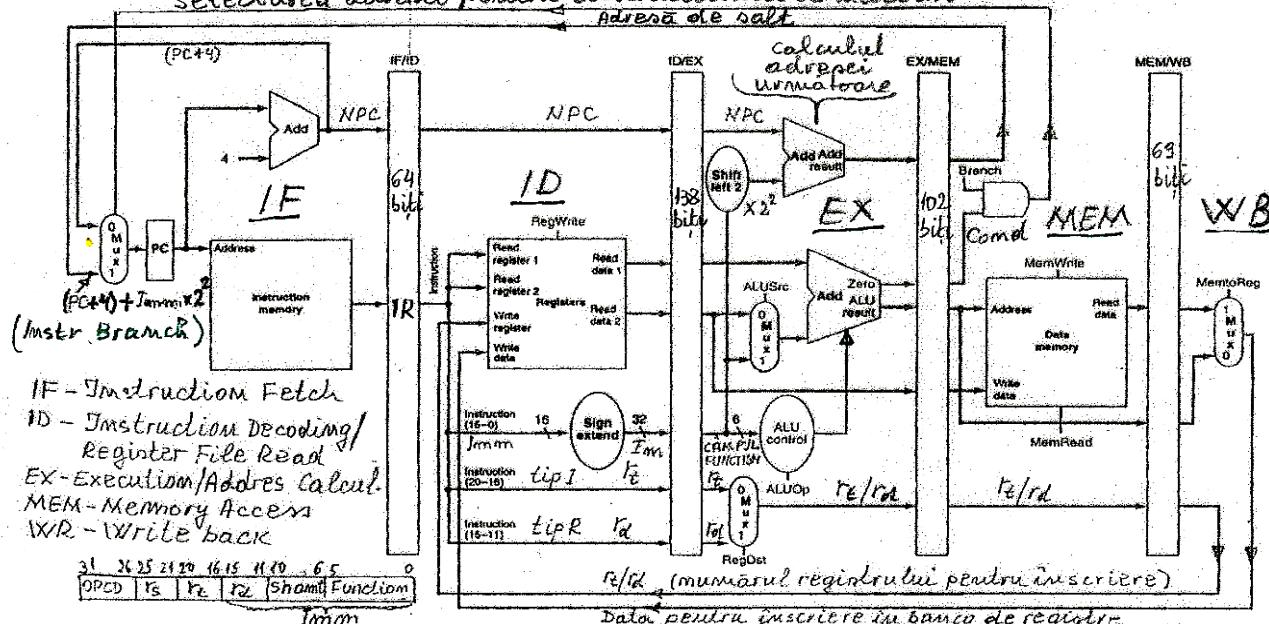


FIGURE 6.22 The pipelined datapath of Figure 6.17 with the control signals identified. This datapath borrows the control logic for PC source, register destination number, and ALU control from Chapter 5. Note that we now need the 6-bit funct field (function code) of the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register. Recall that these 6 bits are also the 6 least significant bits of the immediate field in the instruction, so the ID/EX pipeline register can supply them from the immediate field since sign extension leaves these bits unchanged.

se înscriu următoarele două cuvinte de 32 biți:

IF/ID \leftarrow Instrucțiunea citită din memorie;
IF/ID \leftarrow NPC = PC+4;
PC \leftarrow PC+4;

2. Etapa decodificare și citire operanzi, ID (Instruction Decoding). Câmpul OPCODE (26-31) este aplicat la unitatea de control pentru decodificare, simultan cu citirea operanzilor. Citirea operanzilor se poate realiza în paralel cu decodificarea codului instrucțiuni deoarece instrucțiunea este cu câmpuri fixe, deci se pot citi din banca de registre valorile celor doi operanzi sursă; adresa (registratorul) operandului A este specificată în câmpul 21-25; iar cea a lui B este specificată în câmpul 16-20. Totodată se extrage din corpul instrucțiunii (chiar dacă nu se va folosi, adică nu este o instrucțiune cu Immediat) câmpul 0-15 care ar putea fi un Immediat și acestui cuvânt de 16 biți i se extinde bitul de semn, obținându-se un cuvânt de 32 biți. Deoarece în acest moment nu se cunoaște, pentru că OPCODE nu a fost încă decodificat, dacă instrucțiunea este de tip R, care are numărul registratorului destinație, rd, specificat în câmpul 11-15 sau dacă instrucțiune este de tip I, care are numărul registratorului destinație, rt, specificat în câmpul 16-20, se vor citi ambele câmpuri din corpul instrucțiunii și se vor trimite înainte ambele câmpuri cu numerelor corespunzătoare registratorilor destinație rd și rt.

Timpul de acces pentru citirea unui registrator este chiar mai mic decât $1/2T_{CLK}$, de aceea se poate considera că citirea registratorilor a căror adresă a fost specificată în cuvântul instrucțiune se efectuează în a doua jumătate din perioada semnalului de ceas.

Prin decodificarea câmpului OPCODE, pe unitatea de control (neconfigurată în acest desen) se generează următoarele 9 semnale de control, necesare comenziilor din următoarele etape:

Pentru etapa EX (patru semnale):

1. RegDst – care aplicat multiplexorului din etapa EX selectează numărul unuia din cele două registre destinație rd sau rt (în acel moment se cunoaște tipul instrucțiunii deoarece OPCODE a fost deja decodificat);
2. ALUOp1, 3. ALUOp2 – acești doi biți specifică blocului ALU control, din etapa Execuție (împreună cu câmpul Function – 6biți, pozițiile 0-5 din corpul instrucțiunii) care este operația care se va efectua pe ALU pentru instrucțiune;
4. ALUSrs – cu acest semnal în etapa de execuție se selectează pe un MUX 2:1 sursa celui de al doilea operand (B) aplicat la ALU, care poate fi conținutul registratorului sursă rt (instrucțiuni de tip R) sau este un Immediat (instrucțiunile de tip I sau J) extins la 32 biți.

Pentru etapa MEM (trei semnale):

5. Branch – acest semnal se generează când instrucțiunea este de salt condiționat, care în etapa MEM, în conjuncție cu semnalul Zero, va comanda încărcarea în program counter nu a adresei următoare, PC+4, ci a adresei tintă, PC \leftarrow adresa de salt;
6. MemRead; 7. MemWrite – sunt semnalele care comandă citirea sau înscrierea memoriei pentru instrucțiunea lw respectiv instrucțiune sw (aceste instrucțiuni au un al doilea acces la memorie).

Pentru etapa MEM (trei semnale):

8. RegWrite – cu acest semnal în etapa de WB se înscrive operandul rezultat în registratorul destinație (rd sau rt) din banca de registre (care este de fapt pentru registrator este semnalul de Load);
9. MemtoReg – selecteză prin Mux 2:1 din etapa WB care este sursa pentru operandul rezultat: fie data citită din memorie (pentru instrucțiunea lw) fie data produsă de ALU, pentru o instrucțiune de tip R.

La aplicarea celui de al treilea impuls de ceas în registratorul ID/EX se înscrive un cuvânt de 147 biți compus din:

ID/EX \leftarrow RegDest (1bit)
ID/EX \leftarrow ALUOp1, ALUOp2 (2biți)
ID/EX \leftarrow ALUSrs (1bit)
ID/EX \leftarrow Branch (1bit)
ID/EX \leftarrow MemRead, MemWrite (2biți)
ID/EX \leftarrow RegWrite (1bit)
ID/EX \leftarrow MemtoReg (1bit)
ID/EX \leftarrow rs (32biți)
ID/EX \leftarrow rt (32 biți)

ID/EX \leftarrow Imm = $[I_{15}]^6 \text{xxx} [I_{15} - I_0]$, imediatul extins cu bitul de semn (32 biți)

ID/EX \leftarrow câmpul (16-20) al registrului rt (5biți)

ID/EX \leftarrow câmpul (11-15) al registrului rd (5biți)

ID/EX \leftarrow NPC, adresa instrucțiunii următoare (32 biți)

3. *Etapa de Execuție.* În această etapă, deoarece acum se cunoaște tipul de instrucțiune, pe MUX2:1 se selectează între rd (dacă a fost instrucțiune de tip R) sau rt (dacă a fost instrucțiune de tip I) cu semnalul RegDest, care este registrul de destinație corect pentru instrucțiune.

Selectarea operației efectuată de instrucțiune pe ALU este realizată de semnalele de control, generate de circuitul combinațional ALU control la intrarea căruia se aplică semnalele: câmpul Funcțion din instrucțiune (6biți) și semnalele ALUOp1, ALUOp2 (2biți) produse de unitatea de control prin decodificarea OPCODE.

Operandul A aplicat la ALU este conținutul registrului rs. Operandul B se obține prin selectarea pe MUX 2:1, comandat de semnalul ALUSrs, între conținutul registrului rt sau Immediatul extins la 32 de biți. Pentru instrucțiunea sw conținutul registrului rt se va aplica pentru înscrisere la memorie în etapa următoare.

Pentru instrucțiunile de branch pe circuitul sumator (diferit de ALU) se calculează adresa de salt prin sumarea între NPC (= PC+4) și Immediatul extins la 32 de biți apoi shiftat cu două poziții la stânga (multiplicat cu 4). Totodată, pe ALU se generază și semnalul Zero (= 1 dacă cei doi operanzi aplicati la ALU sunt egali sau =0 dacă cei doi operanzi nu sunt egali) care este condiția de validare a saltului în etapa MEM (pentru beq sau bne).

La aplicarea celui de al patrulea impuls de ceas în registrul EX/MEM se înscrise un cuvânt de 107 biți

EX/MEM \leftarrow Branch (1bit)

EX/MEM \leftarrow MemRead, MemWrite (2biți)

EX/MEM \leftarrow RegWrite (1bit)

EX/MEM \leftarrow MemtoReg (1bit)

EX/MEM \leftarrow rt (32 biți)

EX/MEM \leftarrow NPC + $[I_{15}]^6 \text{xxx} [I_{15} - I_0] \cdot 2^2$ (32 biți) \$rs + $[I_{15}]^6 \text{xxx} [I_{15} - I_0] \cdot 2^2$

EX/MEM \leftarrow ALU result, AopB, (32 biți)

EX/MEM \leftarrow rd/rt (5 biți)

EX/MEM \leftarrow Zero

4. *Etapa de acces la memoria și salt, MEM.* În această etapă rezultatul de la ALU poate fi utilizat ca o adresă de access la memoria de date (\$rs + $[I_{15}]^6 \text{xxx} [I_{15} - I_0] \cdot 2^2$) pentru instrucțiunile lw sau sw, sau este transmis fără nici o modificare spre etapa WB, ca rezultat pentru instrucțiuni aritmetice și logice. Pentru instrucțiunea sw conținutul registrului rt se aplică la memoria de date pentru a fi înscris la adresa accesată (ALU result), iar pentru instrucțiuna lw se citește memoria de date de la adresa accesată (calculată pe ALU). Semnalele de control pentru înscriserea sau citirea memoriei de date sunt MemWrite respectiv MemRead.

Dacă instrucțiunea este de salt condiționat (Branch = 1) și condiția de salt este realizată, Zero =1, atunci se generază semnalul PCSrc = 1 care comandă MUX 2:1, figurat în etapa IF, prin care se selectează pentru adresa instrucțiunii următoare adresa de salt calculată (PC \leftarrow NPC + $[I_{15}]^6 \text{xxx} [I_{15} - I_0] \cdot 2^2$, iar pentru PCSrc = 0 se selectează adresa instrucțiunii următoare (PC \leftarrow NPC = PC +4).

La aplicarea celui de al cincilea impuls de ceas în registrul MEM/WB se înscrise un cuvânt de 71 biți

MEM/WB \leftarrow RegWrite (1bit)

MEM/WB \leftarrow MemtoReg (1bit)

MEM/WB \leftarrow M[ALUresult], pentru instrucțiunea lw (32 biți)

MEM/WB \leftarrow ALUresult, pentru instrucțiune de tip R (32biți)

MEM/WB \leftarrow rs/rt (5 biți)

5. *Etapa de înscrisere a rezultatului, WB (Write Back).* În această etapă se selectează rezultatul, prin semnalul de control MemtoReg pe MUX2:1, între o dată extrasă din memoria de date (pentru instrucțiunea lw) sau o dată produsă în etapa de execuție, ALU result (pentru instrucțiuni de tip aritmetice sau logice), data selectată se înscrise în banca de registre la registrul de adresă rd/rt, sub acțiunea semnalului de control RegWrite.

Registru rt \leftarrow M[ALUresult] sau Registru rd \leftarrow ALUresult

Timpul de acces pentru înscriserea/citirea unui registru este chiar mai mic decât 1/2TCLK, de aceea se poate considera că înscriserea se efectuează în etapa a cincea, WB, pe prima jumătate din perioada semnalului de ceas. Deci dacă înscriserea se efectuează în prima jumătate a perioadei de ceas (corespunzător etapei WB), atunci pe

aceeași perioadă de ceas (corespunzător etapei ID) în a doua jumătate se poate realiza și citirea registrului respectiv.

În figura următoare în structura de pipeline, diferit față de figura anterioară, este reprezentată și unitatea de control cu semnalele generate și punctele în care aceste semnale de control sunt aplicate; apoi sunt reprezentate numai generarea semnalelor de control și explicațiile aferente fiecărui semnal de control.

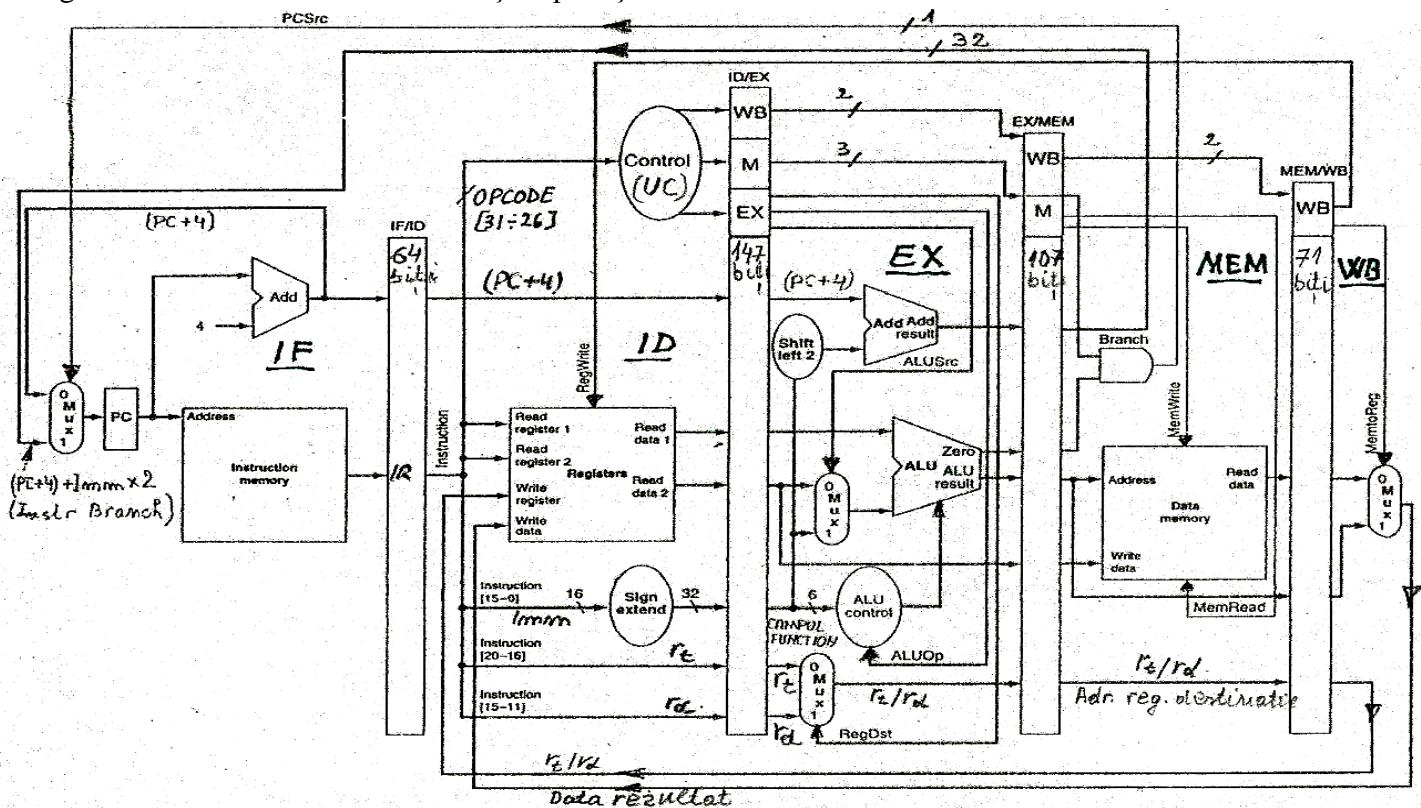


FIGURE 6.27 The pipelined datapath of Figure 6.22, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

Generarea semnalelor de control în unitatea de control și specificarea acțiunii fiecărui semnal de control

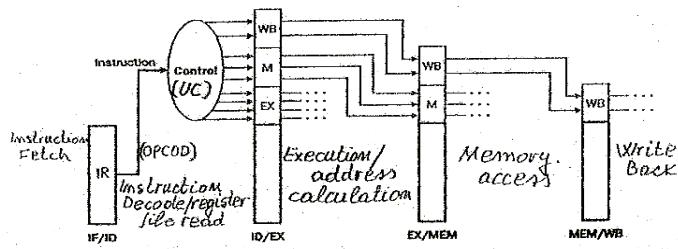


FIGURE 6.29 The control lines for the first three stages. Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage.

ALU control generăza 3 bici pentru comanda ALU pe baza a 6 bici din instrucțiune (cărțeați FUNCTION) și 2 bici ALUOp generati de CONTROL(UC) pe baza cămpului OPCODE din instrucțiune.

Semnalele de control generate de UC, în etapa ID, pe baza cămpurilor OPCODE (+FUNCTION) din instrucțiune, furnesc, împreună cu celelalte cămpuri din instrucțiune (r_1, r_d, Imm_1, A, B) în următoarele etape (EX, MEM, WB). În fiecare etapă aceste semnale se va "consuma" semnalele de control respective.

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Mem to Reg
Format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	x	0	0	1	0	0	1	0	x
beq	x	0	1	0	1	0	0	0	x

FIGURE 6.28 A copy of Figure 5.14 from page 388. This figure shows how the ALU control bits are set depending on the ALUOp control bits and the different function codes for the R-type instruction.

Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	010
SW	00	store word	XXXXXX	add	010
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less than	101010	set on less than	111

Valoarea semnalelor de control pentru tipurile de instrucțiuni

FIGURE 6.27 A copy of Figure 5.15 from page 389. The function of each of seven control signals is defined. The ALU control lines (ALUOp) are defined in the second column of Figure 6.26. When a 1-bit control to a two-way multiplexer is asserted, the multiplexer selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexer selects the 0 input. Note that PCSrc is controlled by an AND gate in Figure 6.25. If the Branch signal and the ALU Zero signal are both set, then PCSrc is 1; otherwise, it is 0. Control sets the Branch signal only during a beq instruction; otherwise, PCSrc is set to 0.

5.1.4. Hazardul în pipeline

Cresterea de viteza (CV) de n ori la o procesare intr-un pipeline cu n etape, in raport cu o procesare de tip non-pipeline, se poate atinge doar cand toate instrucțiunile sunt independente (procesarea uneia nu depinde de celelalte, deci se pot procesa in paralel) se ajunge la performanta de o instrucțiune pe tact, CPI = 1 (se presupune că în pipe fiecare etapă necesită doar un tact).

- Toate tehniciile de procesare pe mașini pipelinate sau pe mașini superscalare se bazează pe identificarea sau crearea de instrucțiuni independente, cu procesare paralelă - **ILP** (Instruction Level Parallel), în timp sau în spațiu (paralelism temporal sau spațial).
- Imposibilitatea de procesare în pipe instrucțiune după instrucțiune, pe fiecare tact, se reflectă prin apariția situațiilor de hazard, adică a unor etape goale în pipe (**stall**), ceea ce înseamnă că se consumă un tact sau mai multe dar nu se efectuează procesarea instrucțiunii în acea etapă/etape. Aceste situații de hazard sunt puternic dependente de organizarea pipe-ului, un program poate fi afectat de apariția de hazard pe o organizare de pipe și să nu fie afectat de apariție de hazard pe o altă organizare de pipe.
- Obținerea unui flux continuu – fără hazard în pipe – este o problemă fundamentată care se pune pentru creșterea performanței de viteza, atât la mașinile pipelinizate cât și la mașinile superscalare. Creșterea de viteza în prezența hazardului în pipe este

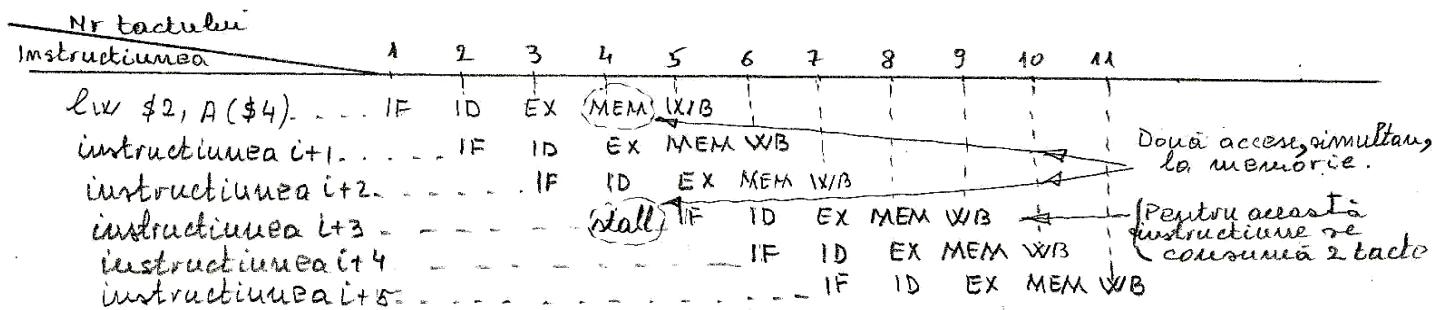
$$CV = \frac{n}{1 + \text{numărul de etape goale în pipe}} < n$$

Situatiile de hazard, după natura care le generează, pot fi de trei tipuri:

- Hazardul structural
- Hazardul de date
- Hazardul de control

5.1.4.1. Hazardul structural

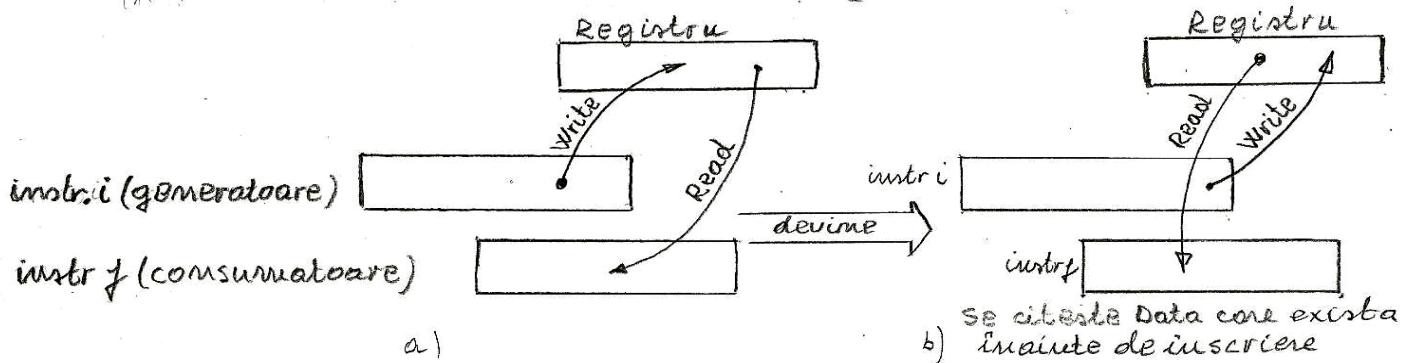
Hazardul structural referă acea situație în pipe prin care se introduc etape goale în fluxul de procesare din cauza lipsei de resurse. Uneori, se acceptă situația de hazard structural (etape goale/stall) dacă înlăturarea sa, prin adăugare de resurse suplimentare, ar costa prea mult în raport cu creșterea de performanță (vezi legea lui Amdahl). În figura următoare se exemplifică o situație de hazard prin forțarea introducerii unui stall în pipe datorită faptului că pe tactul 4 sunt concurente două accesări la o memorie care este de tip singur port pe ieșire.



În figura anterioară, dacă arhitectura este de tip von Neumann (și nu Harvard) sau memoria nu este dublu port pe ieșire atunci pe tactul 4 există competiție la memorie între instrucțiunea lw pentru a citi date și instrucțiunea i+3 pentru etapa IF. Rezolvarea acestei situații de hazard se rezolvă prin oprirea etapei IF pentru instrucțiunea i+3, prin introducerea unei etape goale în pipe, adică între instrucțiunea i+2 și i+3 se introduce o etapă goală care se propagă prin pipe de la intrare spre ieșire (la fel ca o instrucțiune). Introducerea etapei goale în pipe pe tactul 4 determină, efectiv, ca pentru procesarea instrucțiunii i+3 să se consume 2 tacte și nu un tact ca pentru celelalte instrucțiuni. Aplicând relația anterioară pentru calculul creșterii de viteză, CV, rezultă o valoare <n. Evident, o altă rezolvare fără stall în pipe este prin introducerea a unei memorii de date separat (arhitectură Harvard) sau utilizarea unei memorii comune pentru instrucțiuni și date dar dublu port pe ieșire, deci introducerea de noi resurse hardware (care costă).

5.1.4.2. Hazardul de date

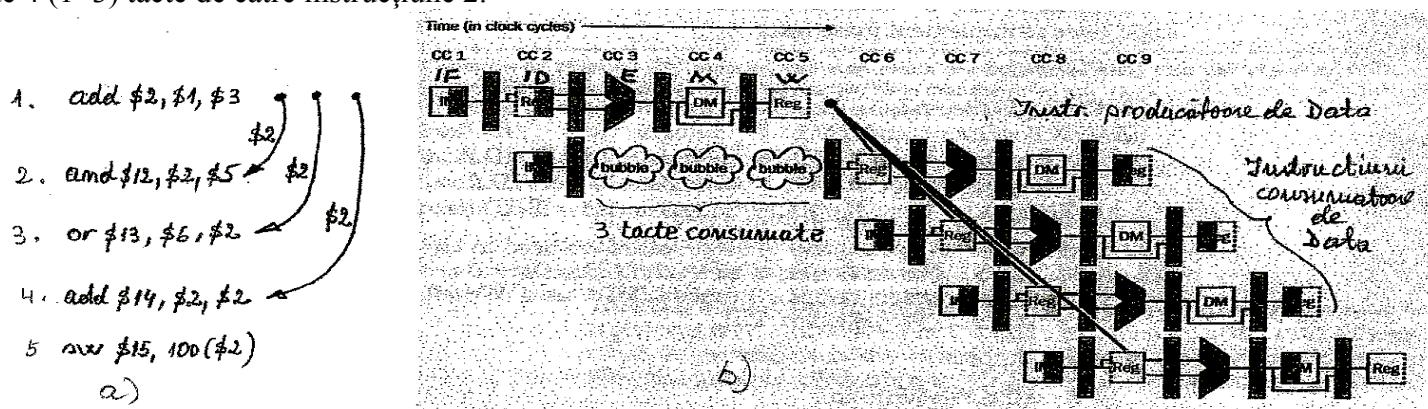
A. Hazardul de tip RAW (Read After Write). Această situație de hazard apare în pipe când o instrucțiune care produce DATA (instrucțiune generatoare) încă nu a generat data până la momentul în care acea dată este necesară pentru o instrucțiune următoare (instrucțiune consumatoare), ceea ce impune introducerea de unu sau mai multe stall-uri în fluxul de instrucțiuni din pipe; această situație de hazard RAW este reprezentată în figura următoare (cele două instrucțiuni i și j se consideră ca fiind successive în program).



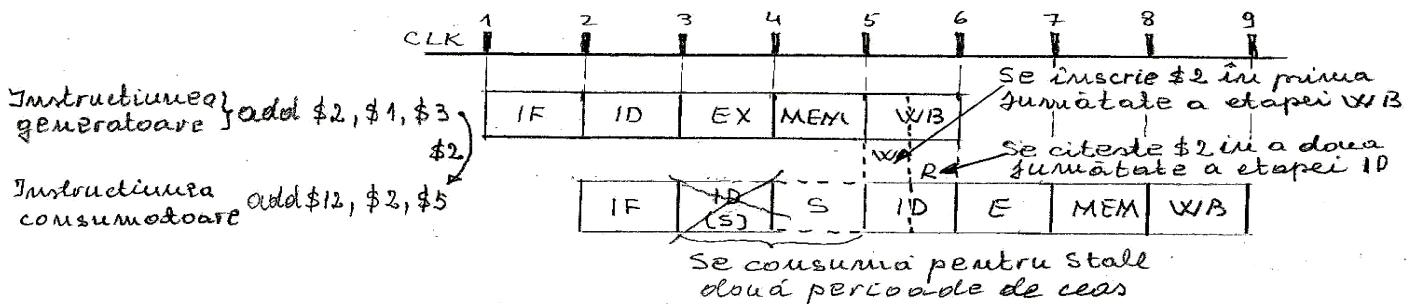
Instrucțiunea consumatoare (j) trebuie să citească data înscrisă de instrucțiunea generatoare (i), Figura a), dar dacă instrucțiunea generatoare încă nu a înscris data până la momentul când cea consumatoare trebuie să citească data respectivă apare situația, figura b), adică cea consumatoare va citi data care există seja în registru sau locație de memorie înainte de înscrierea de către instrucțiunea generatoare. În astfel de situații, de hazard RAW, trebuie opriță (prin tehnici hard sau soft) citirea datei de către instrucțiunea j până când instrucțiunea i va înscrive data respectivă produsă. Dependența de date între cele două instrucțiuni se datorează algoritmului programului care se

rulează și această dependență fundamentală nu poate fi eliminată doar, eventual, dacă se modifică algoritmul, pe când hazardul în pipe depinde de organizarea căii de date.

În figura următoare a) pentru un segment de program sunt reprezentate prin săgeți dependențele de date între prima instrucțiune (generatoarea), add \$2, \$1, \$3, care are ca destinație registrul \$2 și următoarele trei instrucțiuni (consumatoare) care au unul sau ambele registre sursă tot registrul \$2. Această situație de hazard RAW este rezolvată în figura b) prin blocarea pipe-ului, oprirea introducerii de noi instrucțiuni (noi etape IF) în pipe până când prima instrucțiune înscrie data (etapa WB) în registrul destinație \$2; dar aceasta rezolvare de hazard prin blocare a impus introducerea de trei etape goale în pipe (stall), ceea ce este echivalent cu un consum de 4 (1+3) tacte de către instrucțiune 2.



Segmentul de program cu dependență de date prezentat anterior, pentru rezolvarea situației de hazard RAW, la procesorul MIPS necesită nu trei stall-uri de introdus în pipe ci numai două datorită unei particularități a organizării pipe-ului. Această particularitate constă în faptul că perioada semnalului de ceas, TCLK, este destul de lungă (fixată de durata etapelor de access la memorie IF și MEM) încât în etapa de WB înscrierea registrului destinație, \$2, se poate realiza pe durată primei jumătăți a perioadei de ceas, iar pentru citirea unui registru sursă din banca de registre, în care s-a înscris data, în etapa de ID este suficientă doar a doua jumătate din perioada semnalului de ceas (deci numărul de stall-uri se reduce de la 3 la 2).



Exploatând această particularitate, se poate ca pe durata unei singure perioade a semnalului de ceas, în prima jumătate de perioadă se înscrie registrul destinație de către instrucțiunea generatoare în prima jumătate a etapei WB, iar instrucțiunea consumatoare citește registrul sursă (registrar care a fost destinație la instrucțiunea generatoare) doar în a doua jumătate a semnalului de ceas din etapa ID. În figura anterioară se execută etapa IF pentru instrucțiunea consumatoare pe tactul 2, se trece apoi în etapa ID pe tactul 3 (când ar trebui să se citească registrul \$2, dar la aplicarea tactului 4 rezultatele etapei a doua ale instrucțiunii consumatoare nu sunt înschise/transmise în registrul pipe ID/EX, ci rămân blocate în această etapă, instrucțiunea nu mai avansează în pipe și nici nu se execută etapa EX, în pipe apare o etapă goală (un stall) pe tactul 4. Pe durata perioadei tactului 5 în prima jumătate instrucțiunea generatoare înscrie data în registrul destinație \$2, iar în a doua jumătate a perioadei tactului de ceas instrucțiunea consumatoare citește registrul sursă \$2, deci efectiv etapa ID (de citire/aducere a operandilor) pentru instrucțiunea consumatoare este pe durata tactului 5, iar pe tactul 6 instrucțiunea este trecută în etapa EX, în consecință pentru instrucțiunea consumatoare s-au consumat 3 tacte (1+2stall). Pentru instrucțiunea consumatoare etapa ID este efectuată doar pe durata tactului 5 și nu pe tactul 3 (în figură etapa pe acest tact este ștersă), pe tactele 3 și 4 instrucțiunea consumatoare nu a efectuat nici o operație, ceea ce în pipe este echivalent cu 2 stall-uri.

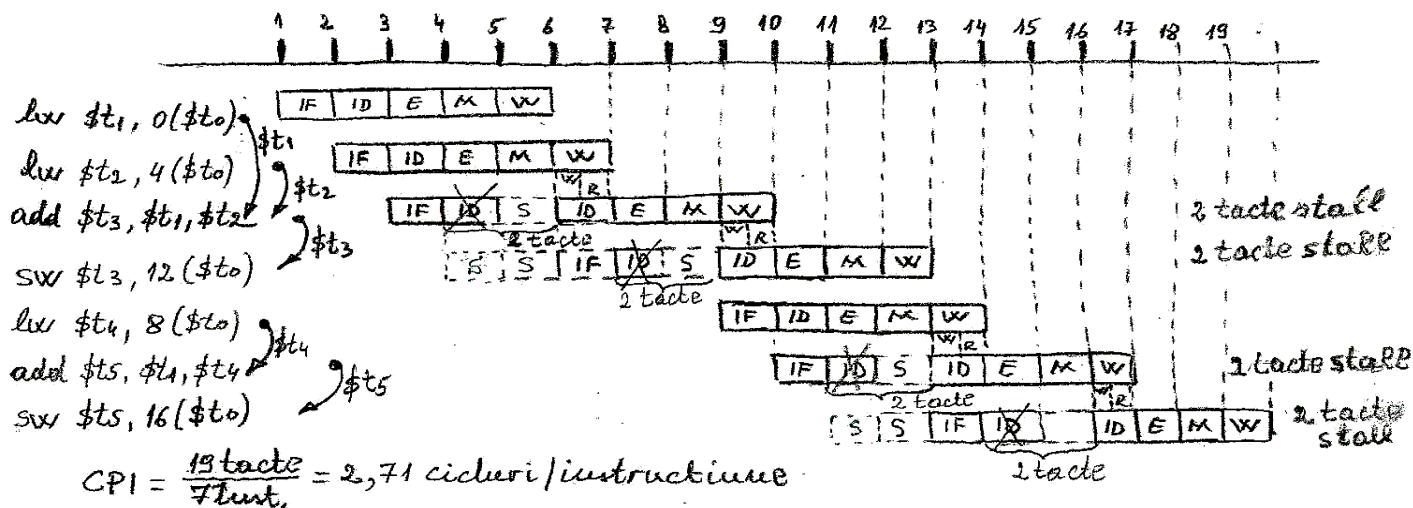
Rezolvarea situației de hazard RAW în pipe se poate realiza fie prin hard fie prin soft, ceea ce se va exemplifica pentru procesorul MIPS.

1. Rezolvarea situației de hazard RAW prin blocarea pipe-ului. Acest mod de rezolvare în hard se reduce la blocarea pipe-ului pentru instrucțiunea consumatoare (neavansare) în etapa ID, care trebuie să citească date din registrul destinație al instrucțiunii generatoare, iar pentru instrucțiunea următoare (după cea consumatoare) nu se realizează etapa IF, instrucțiunea generatoare de DATA își continuă etapele în pipe, dar prin aceasta în pipe între instrucțiunea generatoare și cea consumatoare apar etape goale. Acest mod de procesare cu blocarea pipe-ului este exemplificat în figura următoare în care se procesează un program care realizează operațiile:

$$A = B + D$$

$$C = B + F$$

pentru care compilatorul a alocat registrele: $A \rightarrow \$t_3$; $B \rightarrow \$t_1$; $C \rightarrow \$t_5$; $D \rightarrow \$t_2$; $F \rightarrow \$t_4$



2. Rezolvarea situației de hazard RAW în sof prin rearanjarea instrucțiunilor de către compilator (**Compiler Scheduling**). Prin această modalitate soft, compilatorul inserează între instrucțiunea generatoare de DATA și cea consumatoare de această DATA atâtea instrucțiuni câte etape de stall ar fi fost necesare prin metoda de rezolvare prin blocarea de pipe. Aceste instrucțiuni de inserat sunt aduse din intregul program, dar prin această inserare de instrucțiuni nu trebuie să se modifice semantica programului, dacă nu se găsesc suficiente instrucțiuni pentru inserat atunci pentru completare se introduc instrucțiuni NOP. Pentru același segment de program utilizat în figura anterioară, în figura următoare se exemplifică tehnica de compiler scheduling

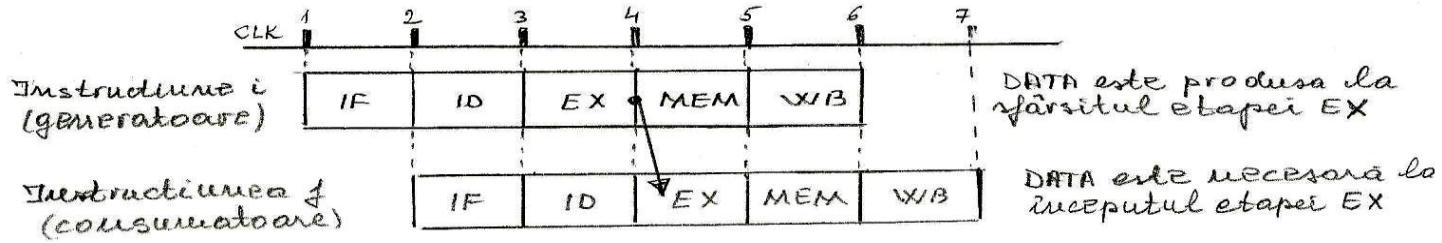
I ₁	lw \$t ₁ , 0(\$t ₀) ;
I ₂	lw \$t ₂ , 4(\$t ₀) ;
I ₃	lw \$t ₄ , 8(\$t ₀) ; Instr I ₃ este adusă înainte (din poziția 5)
I ₄	NOP ; se așteaptă etapa W de la I ₂ pentru înscriserea lui \$t ₂
I ₅	add \$t ₃ , \$t ₁ , \$t ₂ ; se așteaptă etapa W de la I ₅ pentru înscriserea lui \$t ₃
I ₆	NOP ; se așteaptă etapa W de la I ₅ pentru înscriserea lui \$t ₃
I ₇	NOP ; se așteaptă etapa W de la I ₅ pentru înscriserea lui \$t ₃
I ₈	sw \$t ₃ , 12(\$t ₀) ;
I ₉	add \$t ₅ , \$t ₁ , \$t ₄ ;
I ₁₀	NOP ; se așteaptă etapa W de la I ₉ pentru înscriserea lui \$t ₅
I ₁₁	NOP ; se așteaptă etapa W de la I ₉ pentru înscriserea lui \$t ₅
I ₁₂	sw \$t ₅ , 16(\$t ₀) ;

$$Nr de tacă = 16 tacă (5 tacă prima instrucțiune plus 11 intero I₂-I₁₂)$$

$$CPI = \frac{16 \text{ tacă}}{7 \text{ instr.}} = 2,28 \text{ cicluri/instrucțiune}$$

Ideal, ar fi fost ca să se găsească la compilare, în program, încă cinci instrucțiuni pentru a fi inserate în locul instrucțiunilo NOP, cum s-a găsit instrucțiune lw \$t4, 8(\$t0) pentru poziția trei, I₃.

3. Eliminarea hazardului RAW (eliminare stall) prin **forwarding**. Tehnica de eliminare a etapelor de stall introduse pentru eliminarea hazardului RAW se bazează pe observația că DATA produsă de instrucțiunea generatoare este deja produsă în momentul cînd instrucțiunea următoare (consumatoare) necesită DATA pentru procesare, cum este prezentat în figura următoare.

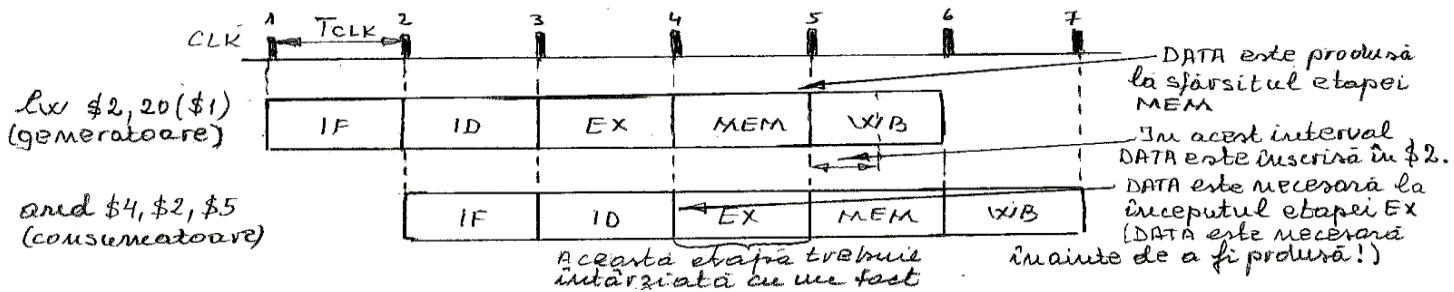


- Dacă instrucțiunea generatoare este de tip R (nu și lw) atunci data este produsă în etapa EX, iar la aplicarea tactului 4 această dată este înscrisă în registrul pipe EX/MEM deci este disponibilă la începutul perioadei T4. Instrucțiunea următoare (consumatoare) necesită data la începutul etapei EX, adică la începutul perioadei de tact 4, deci se poate trimite înainte (forwarding) data înscrisă în registrul pipe EX/MEM la una din intrările ALU încât etapa EX a instrucțiuni consumatoare să se poată realiza. Dar cum se poate determina hazardul de date RAW între instrucțiuni? aceasta se realizează prin utilizarea a şase comparatoare în modul următor:

- se compară numărul registrului destinație al instrucțiunii din etapa EX cu numerele registrelor sursă ale instrucțiunii care este în etapa ID (două comparatoare);
- se compară numărul registrului destinație al instrucțiunii din etapa MEM cu numerele registrelor sursă ale instrucțiunii care este în etapa ID (două comparatoare);
- se compară numărul registrului destinație al instrucțiunii din etapa WB cu numerele registrelor sursă ale instrucțiunii care este în etapa ID (două comparatoare).

Când există o identitate între numerele registrelor comparate se generează semnalul care indică o situație de hazard și se trimit date deja calculată de instrucțiunea generatoare la începutul etapei EX a instrucțiunii consumatoare. De fapt, sunt necesare doar 4 comparatoare deoarece compararea între etapa WB cu etapa ID (punctul 3 anterior) nu mai este necesară, pentru că data calculată se înscrise în registru destinație pe prima jumătate a perioadei de ceas, iar în a doua jumătate se poate citi registrul sursă de către instrucțiunea consumatoare

- Dacă instrucțiunea generatoare este load, lw, atunci data este generată prin citirea memoriei la sfârșitul etapei a patra, MEM, iar la aplicarea tactului al cincilea data este înscrisă în registrul pipe MEM/WB, dar această dată nu mai poate fi utilizată de instrucțiunea următoare (consumatoare) deoarece durata pentru etapa EX a acesteia, tactul patru, s-a încheiat.



Rezultă că mecanismul de forwarding nu mai poate elimina complet introducerea de stall în pipe, data înscrisă în registrul pipe MEM/WB, obținută prin citirea memoriei de date, se poate totuși trimite prin forwarding la una din intrările ALU necesară procesării din instrucțiunea următoare (consumatoare) numai dacă etapa EX a acestei instrucțiuni a fost întârziată cu un tact, instrucțiunea consumatoare a fost blocată în etapa ID, adică s-a introdus un stall în pipe. Înseamnă că perechea instrucțiune lw urmată imediat de o instrucțiune consumatoare a datei extrasă din memorie necesită totdeauna introducerea între cele două instrucțiuni a unei etape goale sau o altă instrucțiune adusă din program.

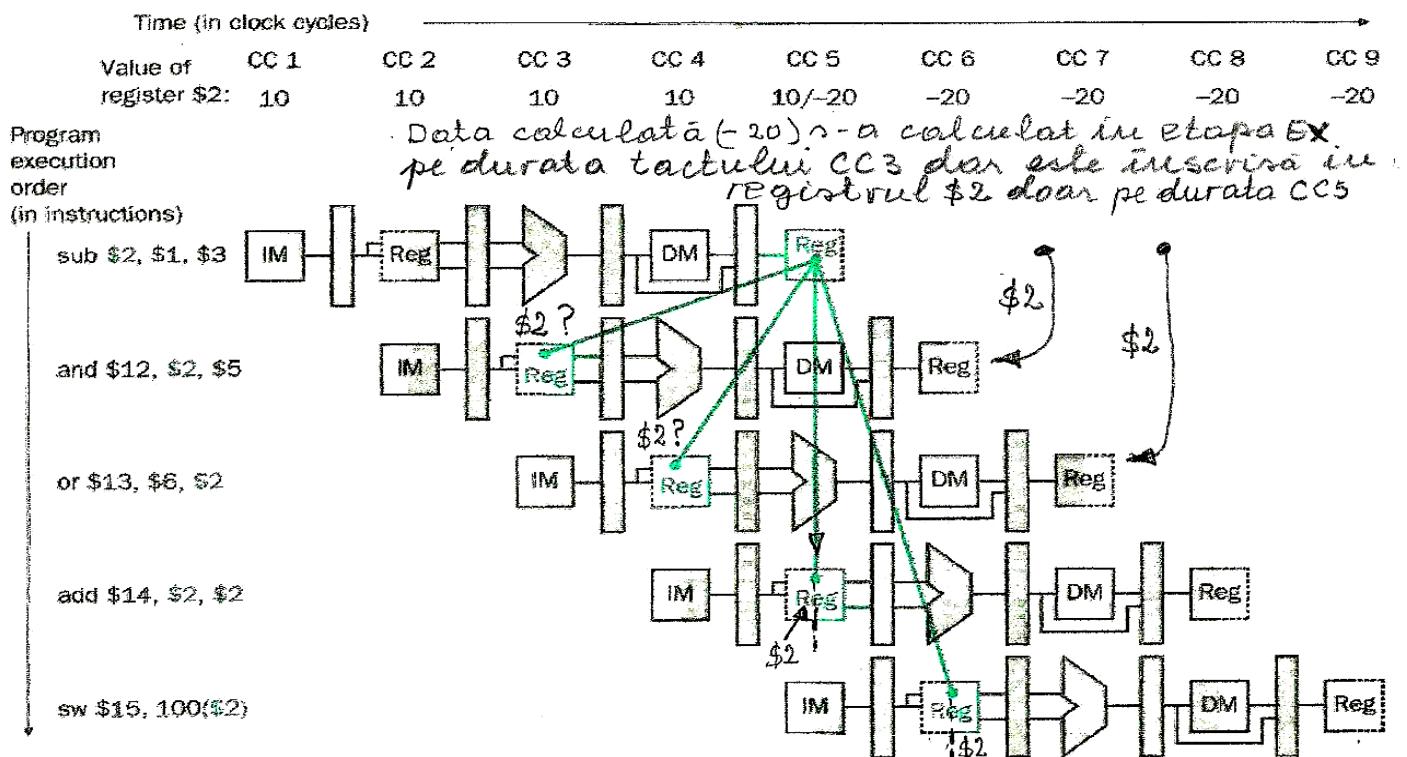


FIGURE 6.36 Pipelined dependencies in a five-instruction sequence using simplified datapaths to show the dependencies. All the dependent actions are shown in color, and “CC *i*” at the top of the figure means clock cycle *i*. The first instruction writes into \$2, and all the following instructions read \$2. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5. (Our register file will read the value written in that clock cycle.) The colored lines from the top datapath to the lower ones show the dependencies. Those that must go backwards in time are *pipeline data hazards*.

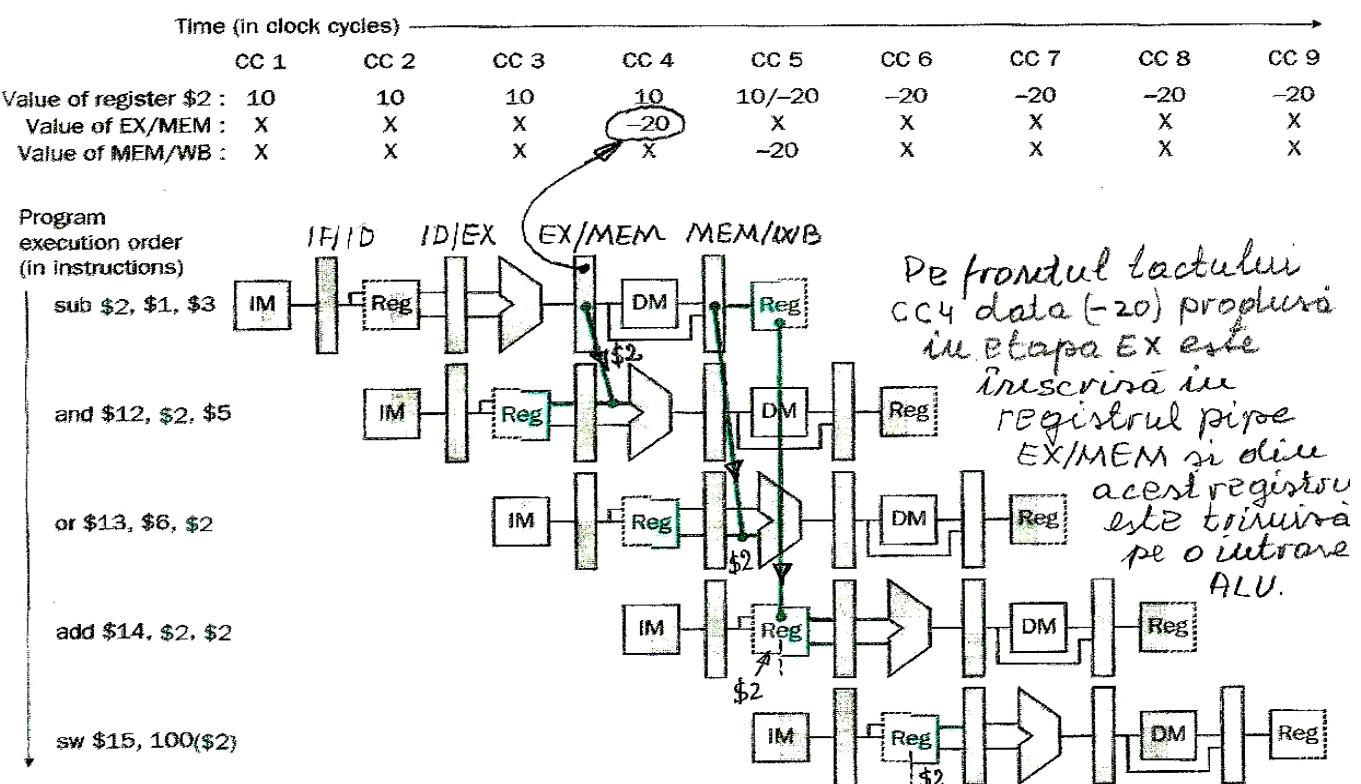


FIGURE 6.37 The dependencies between the pipeline registers move forward in time, so it is possible to supply the inputs to the ALU needed by the and instruction and or instruction by forwarding the results found in the pipeline registers. The values in the pipeline registers show that the desired value is available before it is written into the register file. We assume that the register file forwards values that are read and written during the same clock cycle, so the add does not stall, but the values come from the register file instead of a pipeline register. Register file “forwarding”—that is, the read gets the value of the write in that clock cycle—is why clock cycle 5 shows register \$2 having the value 10 at the beginning and -20 at the end of the clock cycle.

În figura următoare este prezentat modul cum se implementează metoda de eliminare a hazardului de tip RAW când instrucțiunea generatoare este o instrucțiune load, în acest caz eliminarea hazardului se reduce doar la o singură etapă goală în pipe.

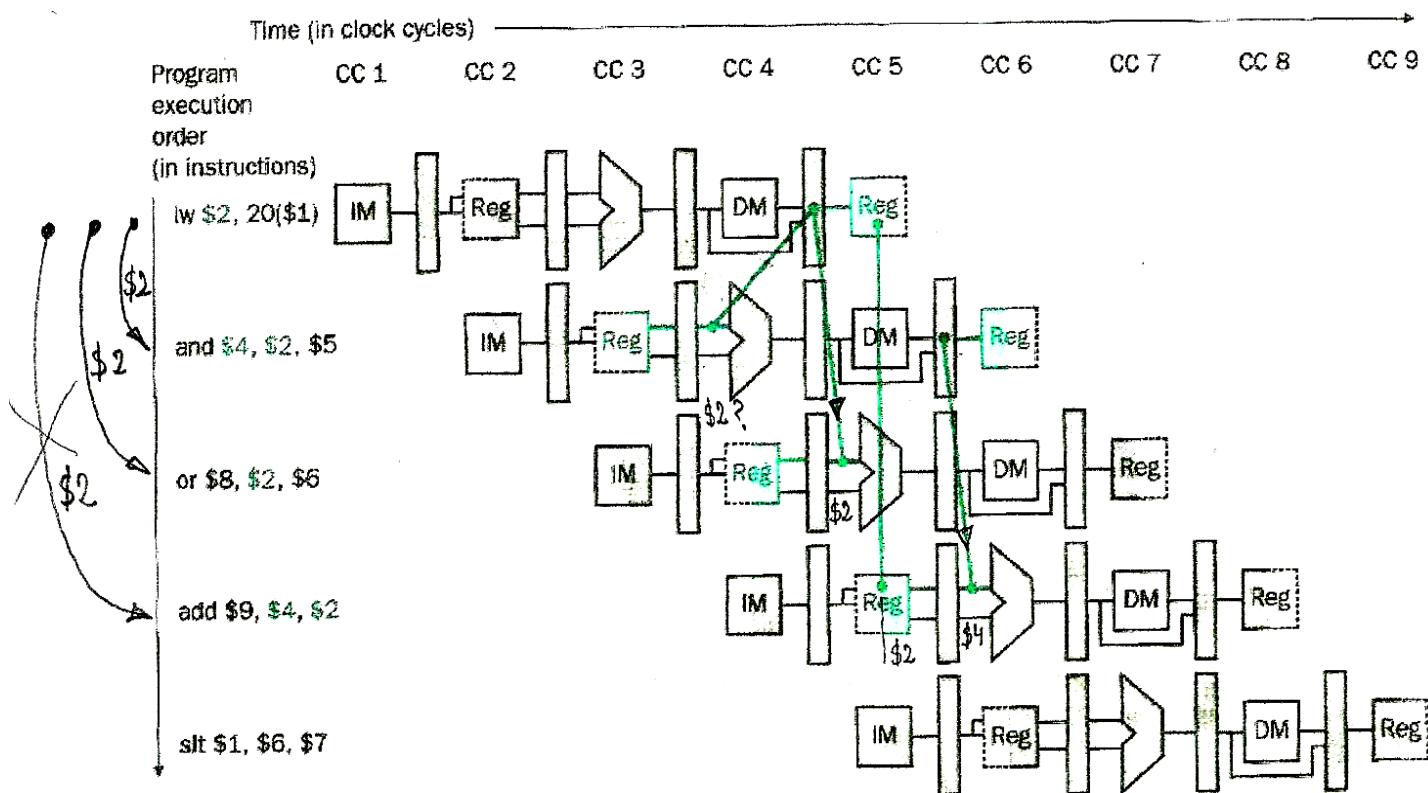


FIGURE 6.44 A pipelined sequence of instructions. Since the dependence between the load and the following instruction (and) goes backwards in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.

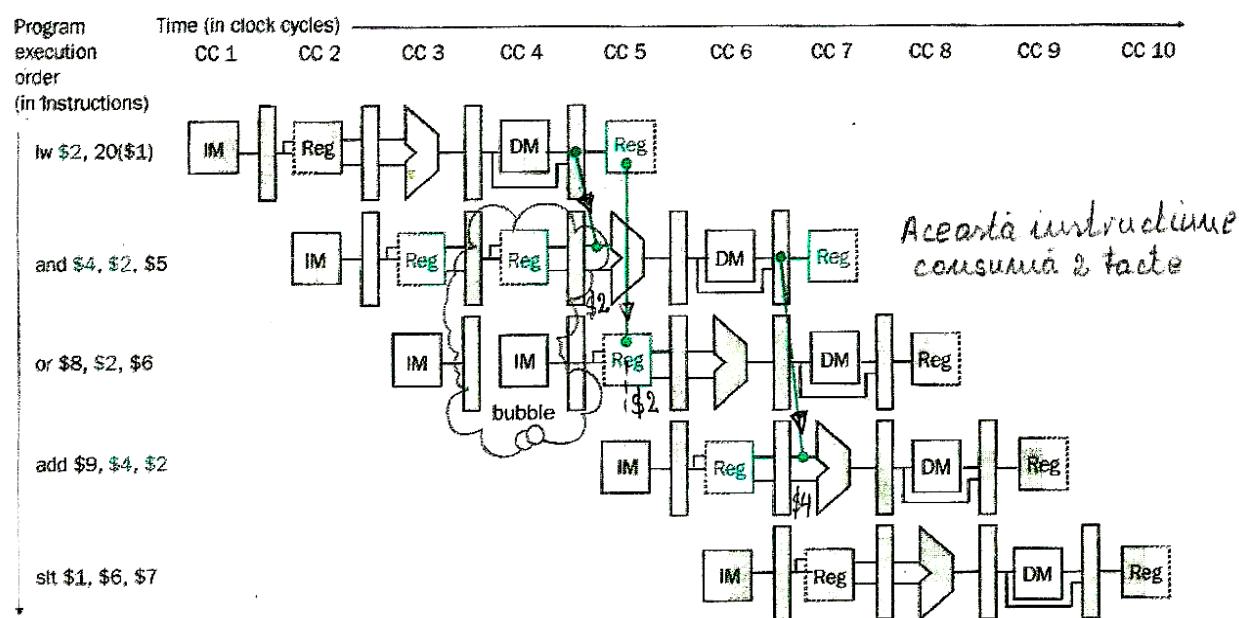
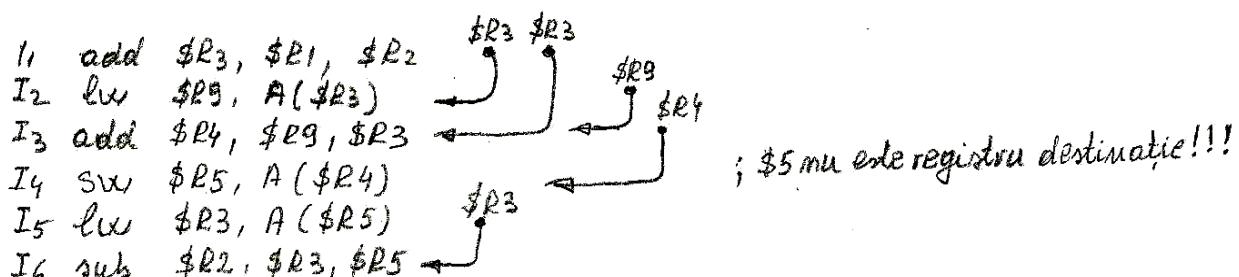


FIGURE 6.45 The way stalls are really inserted into the pipeline. Since the dependencies go forward in time, there are no data hazards.

EXEMPLUL 5.1. Pentru segmentul de program dat:

1. să indice prin săgeți între instrucțiuni apariția de hazard la procesarea în pipe;
2. să se rezolve situațiile de hazard prin introducere de stall-uri prin blocarea pipe-ului, cât este CPI?
3. să se rezolve situațiile de hazard prin metoda forwarding, cât este CPI?

1. – Identificarea situațiilor de hazard în pipe.



2. – Rezolvare prin blocarea pipe-ului (introducere ale stall)

TACTE CONSUMATE	Nr tact →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	STALL INTRODUSE
5+0	I ₁ add \$R3, \$R1, \$R2	IF	ID	E	M	W														
1+2	I ₂ lwx \$R9, A(\$R3)	IF	ID	S	S	E	M	W												2 stall
1+2	I ₃ add \$R4, \$R9, \$R3	IF	S	S	ID	S	S	E	M	W										2 stall
1+2	I ₄ sw \$R5, A(\$R4)	IF	S	S	ID	S	S	E	M	W										2 stall
1+0	I ₅ lwx \$R3, A(\$R5)																			
1+2	I ₆ sub \$R2, \$R3, \$R5																			2 stall.
18 tacte																				TOTAL 18 tacte CPI = $\frac{18 \text{ tacte}}{6 \text{ inst}} = 3 \text{ tacte/instr}$

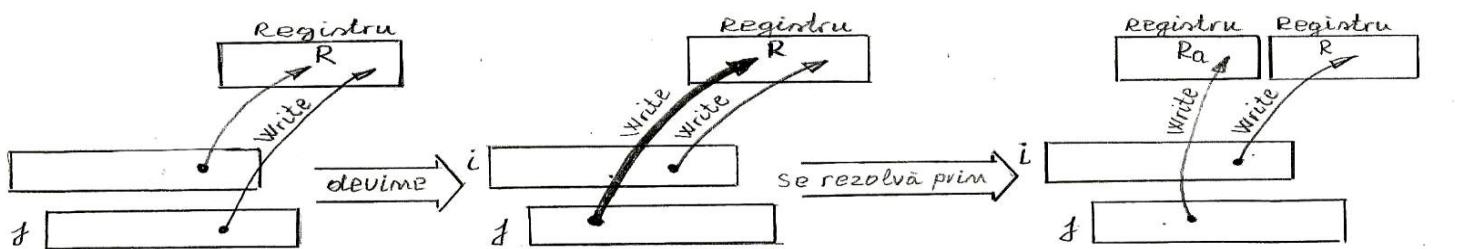
3. – Rezolvare prin mecanismul de forward.

TACTE CONSUMATE	Nr. tact →	1	2	3	4	5	6	7	8	9	10	11	12	STALL INTRODUSE					
5+0	I ₁ add \$R3, \$R1, \$R2	IF	ID	E	M	W													
1+0	I ₂ lwx \$R9, A(\$R3)	IF	ID	E	M	W	\$R3												
1+1	I ₃ add \$R4, \$R9, \$R3	IF	ID	S	E	M	W	\$R9											1 stall
1+0	I ₄ sw \$R5, A(\$R4)	IF	S	ID	E	M	W	\$R4											
1+0	I ₅ lwx \$R3, A(\$R5)	IF	ID	E	M	W													
1+1	I ₆ sub \$R2, \$R3, \$R5	IF	ID	S	E	M	W	\$R3											1 stall
12 tacte																			TOTAL 12 tacte CPI = $\frac{12 \text{ tacte}}{6 \text{ inst}} = 2 \text{ tacte/instr}$

5.1.4.3. Hazardul de nume.

Dependența de date impusă de algoritmul problemei poate determina în pipe situația de hazard RAW, de asemenea lipsa de resurse în pipe poate determina situația de hazard structural, ambele aceste tipuri de hazard deteriorează corectitudinea fluxului de date în program precum și succesiunea de execuție. Situația de hazard de nume apare în pipe când succesiunea valorilor înscrise în registre nu mai reflectă succesiunea corectă impusă de logica programului. Efectiv, situația de hazard se reflectă printr-o concurență a instrucțiunilor în utilizarea registrelor pentru înscriere, în consecință în pipe este necesar să se introducă etape goale/stall. Există două situații de hazard de nume referite ca: 1. hazard de tip WAW (Write-After-Write); 2. Hazardul de tip WAR (Write-After-Read).

1. Hazardul WAW (dependență de ieșire). Acest tip de hazard apare când instrucțiunile i și j (consecutive) au la destinație același regisztr sau locație de memorie, dar instrucțiunea j înscrise înaintea instrucțiunii i. Soluția pentru această situație de hazard constă în duplicarea de registre cum este reprezentat în figura următoare

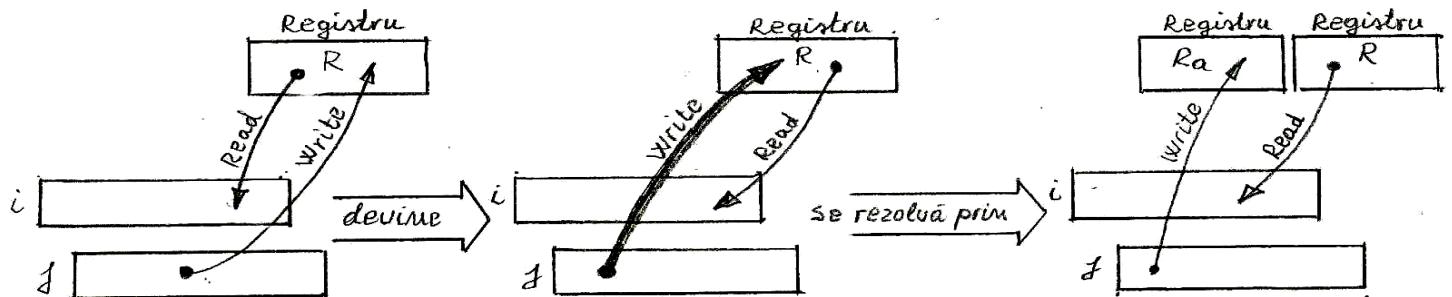


EXEMPLU:

$lwx \$R_1, 0(\$R_2)$ IF ID EX MEM1 MEM2 (W/B)
 add \\$R_1, \\$R_2, \\$R_3 IF ID EX (X/B)

; pentru faptul ridicată accesul la memoria
 ; de date necesită în pipe două etape
 ; MEM1, MEM2. Instrucțiunea pentru ALU nu
 ; necesită acces la memoria (doar 4
 ; etape) deci înscrise \$R_1 înainte ca
 ; instrucțiunea lwx să înscrive \$R_1

2. Hazardul de tip WAR (antidependență). Acest tip de hazard apare cînd ordinea normală din program prin care instrucțiunea i citește data din regisztrul R, iar apoi instrucțiunea j înscrise o altă data în același regisztr este inversată, adică întâi instrucțiunea j înscrise data în regisztrul R și apoi instrucțiunea i citește data din acel regisztr. Soluția pentru această situație de hazard, la fel ca și la hazardul WAW, constă în duplicarea de registre cum este reprezentat în figura următoare.



Denumirea de antidependență este corelată cu situația de dependență normală de date la care prima instrucțiune produce date și a doua instrucțiune utilizează acea valoare, pe când la hazardul WAR este tocmai invers, adică a doua instrucțiune distrugă (prin înscriere) data pe care prima instrucțiune ar trebui să o utilizeze.

Hazardul de nume nu poate apărea la organizare de pipe în care toate instrucțiunile ISA au același format și procesarea instrucțiunilor în pipe necesită aceeași durată ciclu instrucțiune, T_{instr} , și se face strict în ordinea în care acestea sunt în succesiunea din program. În schimb, hazardul de nume poate apărea când în ISA există mai multe tipuri de formate de instrucțiuni sau chiar când există un singur tip de format dar instrucțiunile sunt trimise spre procesare sau sunt terminate de procesat în pipe în afara ordinii în raport cu succesiunea instrucțiunilor din traseul parcurs în program (vezi mașinile superscalare).

Hazardul de nume în pipe se rezolvă prin duplicare de registre, mecanism care este referit ca **redenumirea regisrelor (register renaming)**, ceea ce se va exemplifica în continuare pe următorul segment de program

- I1: $R1 \leftarrow R2 / R3$; această instrucțiune necesită un T_{instr} mult mai lung decât instrucțiunile I2 și I3.
 I2: $R4 \leftarrow R1 + R5$; RAW cu I1
 I3: $R5 \leftarrow R6 + R7$; WAR cu I2
 I4: $R1 \leftarrow R8 + R9$; WAW cu I1

Între I1 și I2 există hazard RAW, I2 trebuie să aștepte până I1 înscrie registrul destinație R1. Pentru că I2 trebuie să aștepte după I1 (care are T_{instr} lung) atunci și I3 trebuie să aștepte pe I2 deoarece între I2 și I3 apare un hazard WAR, dacă I3 este executată înainte de I2 atunci sursa acesteia R5 ar fi înscrisă de destinația instrucțiunii I3 care este tot R5. Dar nici I4 nu poate fi inițiată pentru procesare până când nu este terminată I1, deoarece între I4 și I1 există hazard WAW prin registrul R1. Aceste dependențe pot fi rezolvate prin metoda de redenumire a registrelor după cum se va explica în continuare

Prin denumirea registrelor:

- un registru destinație Ri dintr-o instrucțiune, notat fără index alfabetic, este considerat ca un registru logic conform instrucțiunii din program;
- prin redenumire, oricare nouă valoare creată de o instrucțiune având ca destinație registrul logic Ri va fi de fiecare dată înscrisă într-un alt registru fizic notat succesiv și cu index alfabetic Ria, Rib, Ric, \dots printr-un proces de redenumire dinamic (aceste registre fizice obținute prin redenumire, sunt alocate dinamic de procesor dintre registrele libere la acel moment, unele procesoare au un set de registre libere special pentru operația de redenumire);
- oricare referire într-o instrucțiune la registrul logic Ri , ca registru sursă, va fi dirijată către cea mai recentă (recent în sensul secvenței instrucțiunilor din program) înscriere specificată către registrul logic Ri , dar efectuată într-un registru fizic Ria, Rib, Ric, \dots

Redenumind registrele în segmentul de program anterior se obține următorul program (se consideră că toate registrele nu au fost redenumite înainte în acest segment de program, deci nu au nici un index):

- I1: $R1a \leftarrow R2 / R3$; se redenumește registrul logic R1, devine registrul fizic R1a
 I2: $R4a \leftarrow R1a + R5$; se redenumește registrul logic R4, devine registrul fizic R4a
 I3: $R5a \leftarrow R6 + R7$; se redenumește registrul logic R5, devine registrul fizic R5a
 I4: $R1b \leftarrow R8 + R9$; se redenumește registrul logic R1 (a doua oară), devine registrul fizic R1b

Se consideră, ca exemplificare, că redenumirea începe doar cu acest segment de program, adică cu instrucțiunea I1, iar registrele fizice libere sunt în plus, peste cele din banca de registre și sunt: R32, R33, R34, R35, Prin redenumire segmentul de program devine

- I1: $R32 \leftarrow R2 / R3$
 I2: $R33 \leftarrow R32 + R5$
 I3: $R34 \leftarrow R6 + R7$
 I4: $R35 \leftarrow R8 + R9$

Destinația pentru I1 este registrul logic R1, prin redenumire, valoarea se înscrie în registrul fizic $R1a = R32$. Pentru I2 sursa este $R1a = R32$, iar destinația la registrul logic R4, înscrierea este direcționată spre registrul fizic $R4a = R33$.

Înstrucțiunea I3 are ca destinație pe R5, dar redenumit înscrierea este direcționată spre $R5a = R34$

Înstrucțiunea I4 are ca destinație registrul logic R1 , dar prin redenumire (a doua oară) înscrierea este direcționată spre $R1b = R35$.

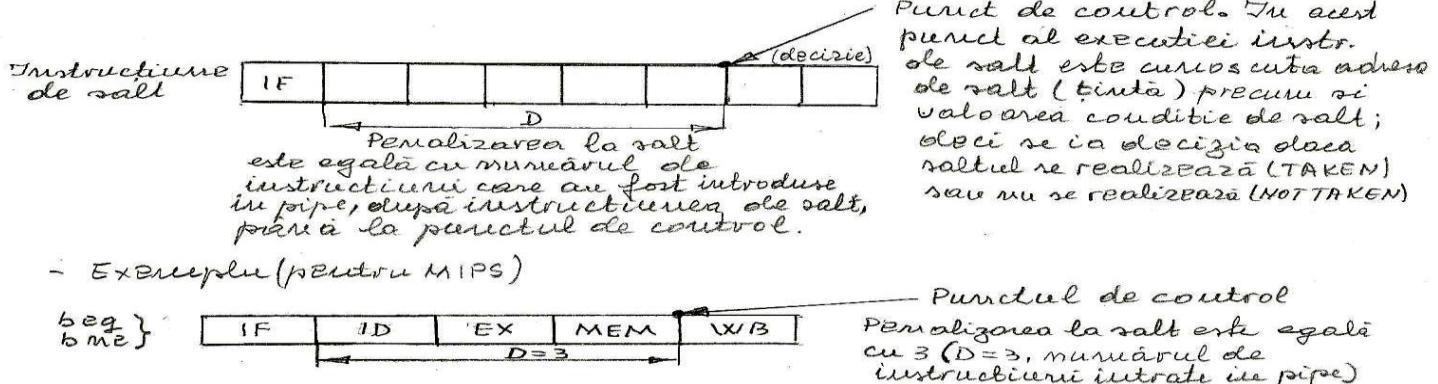
Prin redenumirea registrului destinație R5 din I3 redirecționat în registrul fizic R5a dispare antidependanța (WAR) între I2 și I3, acum I3 poate fi lansată în execuție imediat (se oarece înscrie în R34) fără redenumire I3 ar fi trebuit să aștepte până I1 ar fi fost procesată complet iar I2 ar fi fost lansată în procesare. De asemenea, prin redenumire dispare dependența de ieșire (WAW) între I1 și I4 (I1 înscrie în R32 iar I4 înscrie în R35) deci I4 poate fi lansată în procesare imediat după I3.

5.1.4.4. Hazardul de control

Situatiile de hazard denumite de control sunt generate de instrucțiunile care provoacă salt în program.

Programul este parcurs adresă după adresă, în ordinea de creștere a numerelor naturale, doar în cadrul unui bloc liniar. Un **bloc liniar (bloc de bază)** este o succesiune de instrucțiuni cuprins între două instrucțiuni de salt, deci o secvențialitate neîntreruptă, în blocul liniar se intră prin prima instrucțiune care este o instrucțiune țintă de salt și se ieșe din blocul liniar printr-o instrucțiune de salt. *Hazardul de control contribuie cu cea mai mare pondere în micșorarea performanței de viteză a procesorului*; statistic, între 4-7 instrucțiuni de program una este de salt.

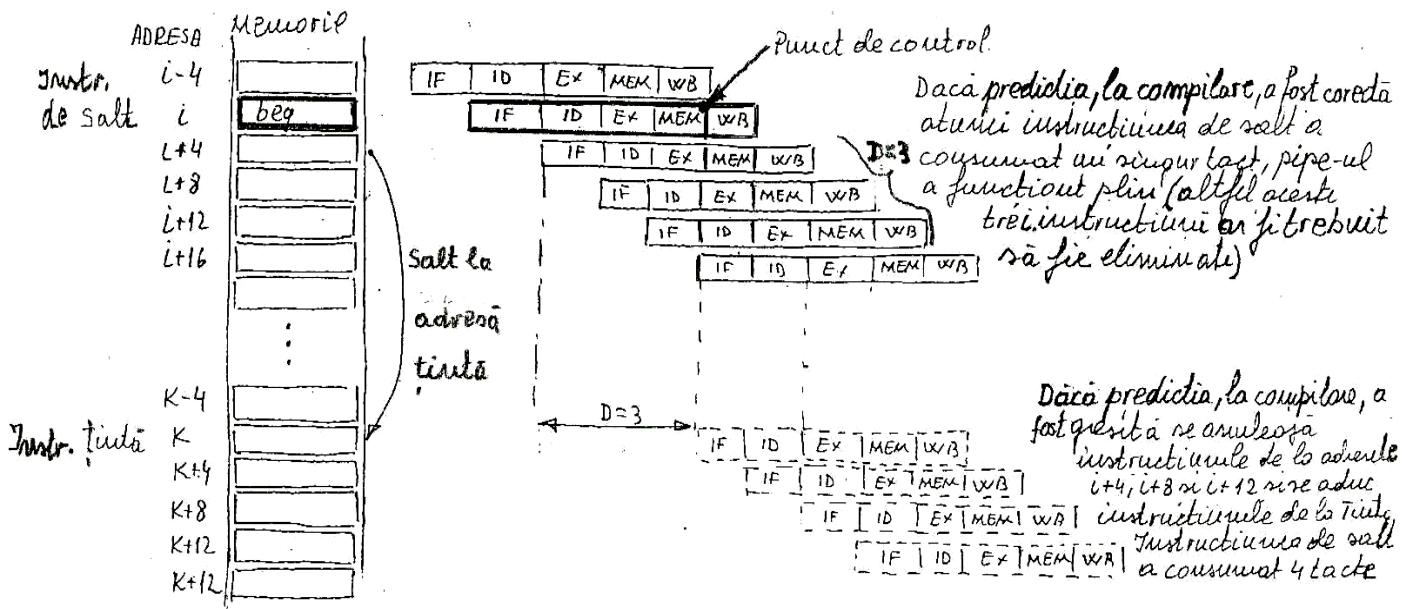
Micșorarea preformanței de viteză este determinată de penalizarea ("prețul") la salt datorită faptului că pentru instrucțiunea de salt în etapa de fetch încă nu se cunoaște dacă saltul se realizează (TAKEN) sau nu se realizează (NOT TAKEN) până în punctul de control/decizie al instrucțiunii de salt, în consecință toate instrucțiunile următoare care au fost introduse în pipe, după instrucțiunea de salt respectivă, până la momentul deciziei din punctul de control sunt anulate sau nu, după caz. Numărul de instrucțiuni, D, intrate în pipe după instrucțiunea de salt dar anulate, dacă condiția de salt este adevarată, este măsura penalizării la salt, după cum este prezentat în figura următoare.



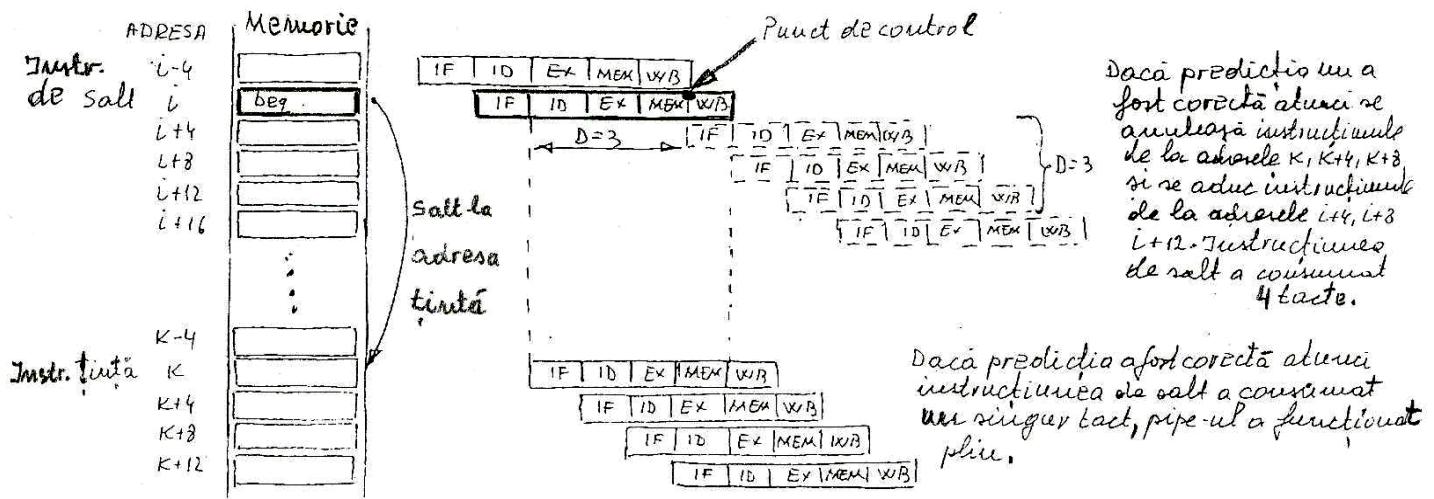
Pentru rezolvarea hazardului de control este necesară o informație pe baza căreia se decide, cu o anumită probabilitate, dacă saltul se execută sau nu, iar această decizie poate fi luată static (în etapa de compilare) sau dinamic (pe durata procesării instrucțiunii de salt) sau combinată.

A. Rezolvarea statică a situației de hazard de control. Rezolvarea statică se reduce la decizia care ia pentru instrucțiunea de salt în etapa de compilare a programului și care poate fi: predicția că saltul se execută sau predicția că saltul nu se execută.

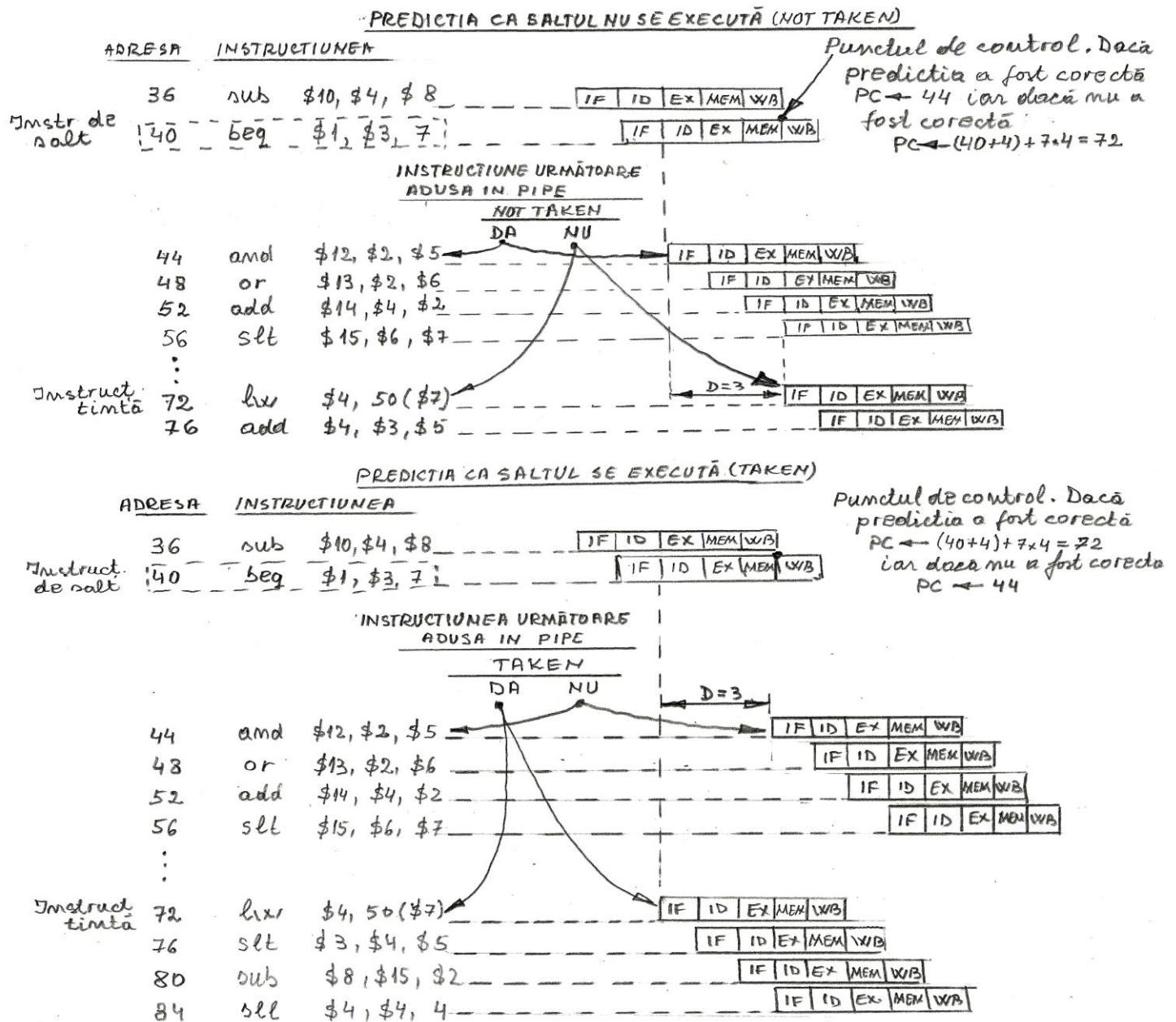
- Predicția (statică) saltul nu se execută



- Predicția (statică) saltul se execută



În exemplul următor se analizează situația de salt pentru MIPS când predicția statică este că saltul nu se execută (Not Taken) și când predicția este că saltul se execută (taken).



În aceste analize de predicție la salt, pentru simplificare, s-a considerat că se cunoaște adresa de salt, iar în punctul de control se obține doar validarea condiției de salt. Dacă s-ar fi considerat că și adresa de salt se calculează (nu se cunoaște anterior) atunci nu s-ar fi putut aduce instrucțiunea de la adresa țintă calculată (k) și introdusă imediat după instrucțiunea de salt. În realitatea instrucțiunea de salt calculează atât adresa de salt cât și efectuarea validării condiției de salt, deci informația completă pentru ambele este disponibilă doar în punctul de control. Oricum, această analiză pentru ambele componente ale instrucțiunii de salt condiționat (adresă țintă, condiție de salt) trebuie făcută pentru fiecare procesor în parte în funcție de particularitățile sale.

Dar care este informația necesară compilatorului pe care se bazează luarea decizei/predicției de taken sau not taken? Mai trebuie specificat că o predicție asupra unei instrucțiuni nu este fixă, odată pentru totdeauna. Informația necesară pe care se bazează predicția este de natură statistică, un exemplu de o astfel de statistică este prezentată în tabelul următor[4].

Tipul de Aplicație	% de instr. de: branch,jump, call, return	% branch (din care % Taken)	% jump (din care % direct)	% Call	% Return
SPEC95int	20,4	14,9 (46)	1,1 (77)	2,2	2,1
Desktop	18,7	13 (39)	1,1 (92)	2,4	2,1

În general, saltul înapoi este realizat cu o probabilitate foarte mare, prin faptul că acesta efectuează saltul de la sfârșit de buclă, pe când saltul condiționat înainte este realizat cu o probabilitate mai mică.

O recomandare arhitecturală care micșorează penalizarea la salt (D) constă în realizarea instrucțiunilor de salt având punctul de control într-o etapă cât mai la începutul intrării în pipe, în această modalitate numărul de instrucțiuni care urmează în pipe după instrucțiunea de salt este mai mic, deci în caz că aceste instrucțiuni trebuie eliminate implicit penalizarea este mai redusă. La MIPS aplicând această recomandare, punctul de control de la sfârșitul etapei a patra (înscrierea pe al cincilea impus de tact în registrul pipe MEM/WB) este mutat în etapa a doua ID, deci penalizarea la salt se reduce de la D = 3 la D = 1. Pentru aceasta, după cum rezultă din figura următoare, în etapa a doua se introduc un comparator (pentru detectarea condiției = sau ≠) și un sumator (pentru calculul adresei instrucțiunii țintă, $PC \leftarrow NPC + Imm \times 4$)

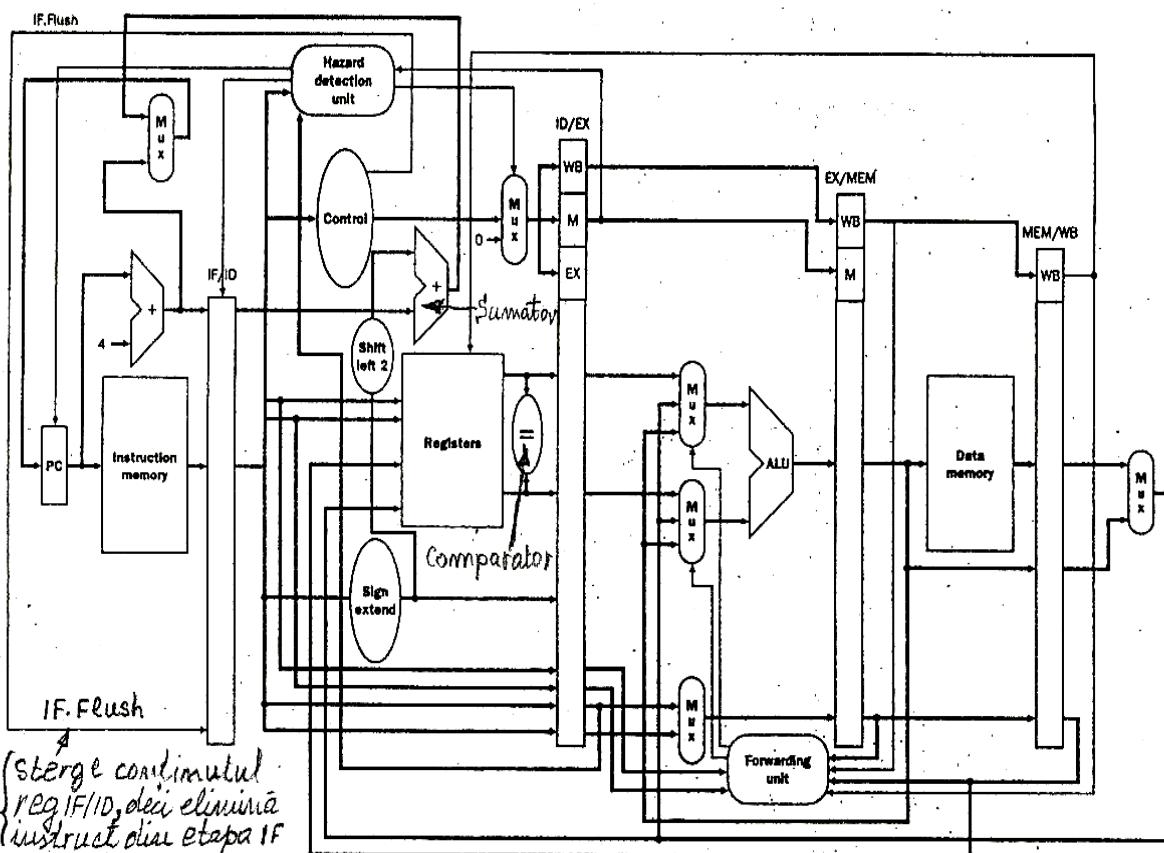
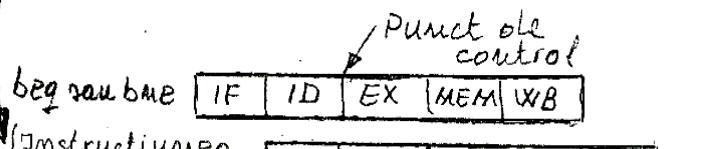


FIGURE 6.51 Datapath for branch, including hardware to flush the instruction that follows the branch. This optimization moves the branch decision from the fourth pipeline stage to the second; only one instruction that follows the branch will be in the pipe at that time. The control lines IF.Flush turns the fetched instruction into a nop by zeroing the IF/ID pipeline register. Although the flush line is shown coming from the control unit in this figure, in reality it comes from hardware that determines if a branch is taken, labeled with an equal sign to the right of the registers in the ID stage.



(D=1) →

Instructiona tura: IF | ID | EX | MEM | WB

contine doar doar ilustrația lui de salt bne \$1, \$2, ACORO, beg \$1, \$2, ACOL0.

Pentru acestea condiția de salt = sare ≠ se poate calcula cuor în etapa

a doua prin compararea conținutului celor două registre selectate (cu porti NXOR-identitate).

Calculul adresei de salt, la fel se calculează cuor

priu introducerea unei reușite în etapa ID care adună $(PC+4) + [Imm15]^{16} \# Imm15 \div 0$

Acste calcule se fac pe cale fizică instrucțiune, doar atunci este o

instrucțiune de salt (decodificarea este realizată la sfârșitul durată etapei

ID) și atunci saltul se execută atunci adresa calculată pe cale fizică adresa tura

se introduce în PC și prin IF.Flush se sterge continutul din reg PC(IF/ID)

(se sterge instrucțiunea din etapa IF) Doar nu este instrucție de salt sau atunci

saltul nu se execută aceste calcule nu sunt utile, instrucțiunea de la

adresă din memorie în etapa IF nu este eliminată.

Deplasarea punctului de control al saltului din etapa a patra, MEM, (D=3) în etapa a doua, ID, (D=1) se poate realiza usor pentru MIPS deoarece setul de instrucțiuni

deplasarea punctului de control al saltului din etapa a patra, MEM,

(D=3) în etapa a doua, ID, (D=1) se

poate realiza usor pentru MIPS

deoarece setul de instrucțiuni

deplasarea punctului de control al saltului din etapa a patra, MEM,

(D=3) în etapa a doua, ID, (D=1) se

poate realiza usor pentru MIPS

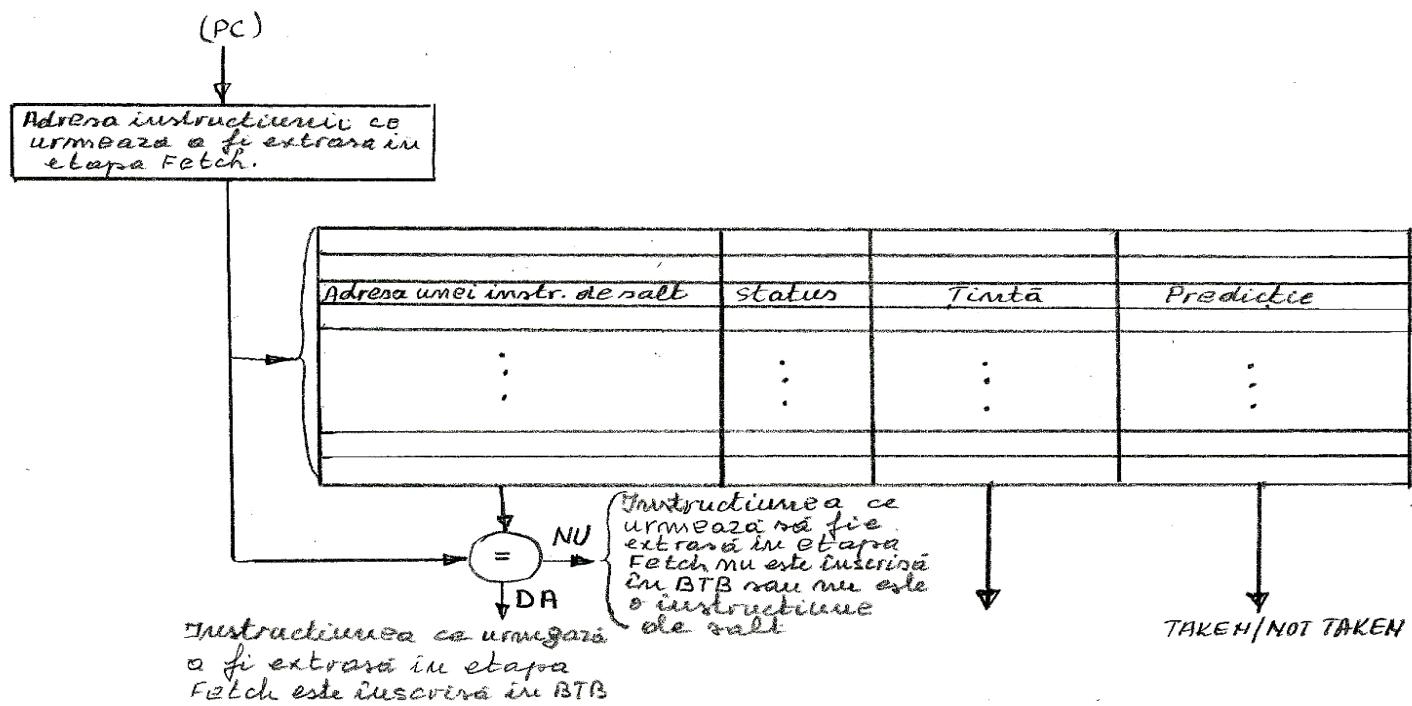
deoarece setul de instrucțiuni

deplasarea punctului de control al saltului din etapa a patra, MEM,

(D=3) în etapa a doua, ID, (D=1) se

poate realiza usor pentru MIPS

B. Rezolvarea dinamică. Prin metodele dinamice de rezolvare a hazardului de control decizia dacă instrucțiunea de salt condiționat se execută sau nu (taken/not taken) este luată chiar în timpul rulării instrucțiunii respective, adică o **predicție dinamică**. Luarea deciziei se bazează pe informația obținută din analiza efectuării și a neefectuării salturilor anterioare ale instrucțiunii respective de salt (adică "istoria locală" a salturilor) și uneori și a analizei efectuării și a neefectuării salturilor ale altor instrucțiuni de salt anterioare din programul care se execută (adică "istoria globală a salturilor"). Structura de principiu pentru implementarea predicției dinamice este prezentată în figura următoare și este referită prin Bufferul Țintei de Salt, BTB (Branch Target Buffer).

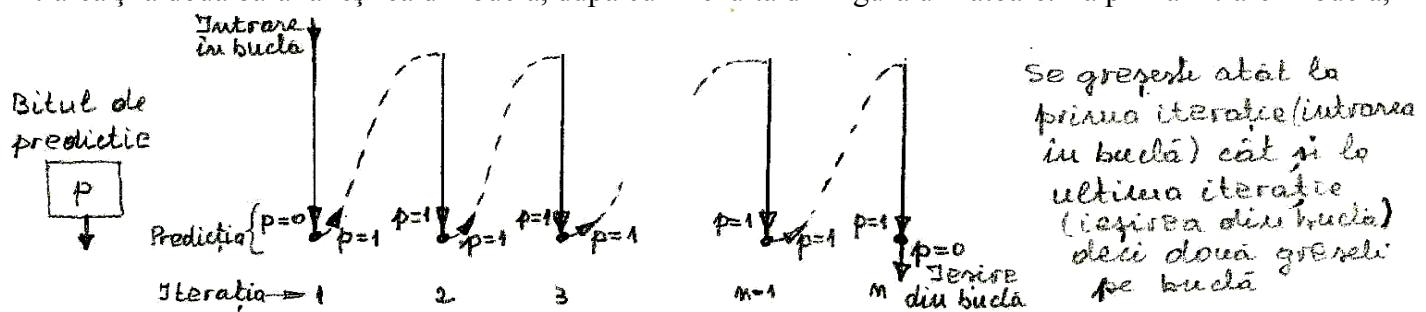


Câmpurile bufferului țintei de salt sunt:

- Adresa instrucțiunii de salt. În acest câmp sunt înscrise adresele tuturor instrucțiunilor de salt din pagina curentă a programului. Când în etapa Fetch este accesată o instrucțiune în memoria I-cache, adresa instrucțiunii respective se aplică în paralel și la BTB, care este o memorie cu accesare asociativă, deci adresa respectivă din PC se compară simultan cu toate adresele instrucțiunilor de salt înscrise ; dacă adresa respectivă există în acest câmp din BTB înseamnă că pentru instrucțiunea care este în curs de aducere din memoria I-cache în etapa Fetch se știe deja că este o instrucțiune de salt pentru care va trebui să se determine adresa de salt (care se extrage din câmpul Țintă) și să se decidă (prin informația din câmpul Predicție) dacă saltul se execută sau nu. Dacă ieșirea comparatorului este NU, iar după decodificarea instrucțiunii aduse rezultă că este totuși o instrucțiune de salt, după execuția ei, sistemul de operare va înscrie (actualiza) în BTB și adresa acestei instrucțiuni de salt (care nu fuseseră înscrisă în BTB);
- câmpul status conține informație referitoare la managementul BTB;
- câmpul predicție conține predicția dacă instrucțiunea adusă în etapa Fetch se execută sau nu (taken/not taken). După fiecare execuție a instrucțiunii de salt în acest câmp se reînscrive cu, după cum instrucțiunea a efectuat sau nu saltul la adresa țintă și aceasta va constitui predicția de taken/not taken pentru următoare accesare a acestei instrucțiuni. Informația taken/not taken poate fi conținută într-un cuvânt de unu, doi sau mai mulți biți.
- câmpul țintă conține informație despre ținta instrucțiunii de salt, această informație despre țintă poate fi stocată sub forma:
 1. instrucțiunea de la adresa țintă sau o succesiune de instrucțiuni începând cu instrucțiunea de la adresa țintă;
 2. adresa instrucțiunii țintă pentru saltul instrucțiunii de salt ;
 3. câteva instrucțiuni (de început) de la țintă plus adresele cătorva instrucțiuni următoare.

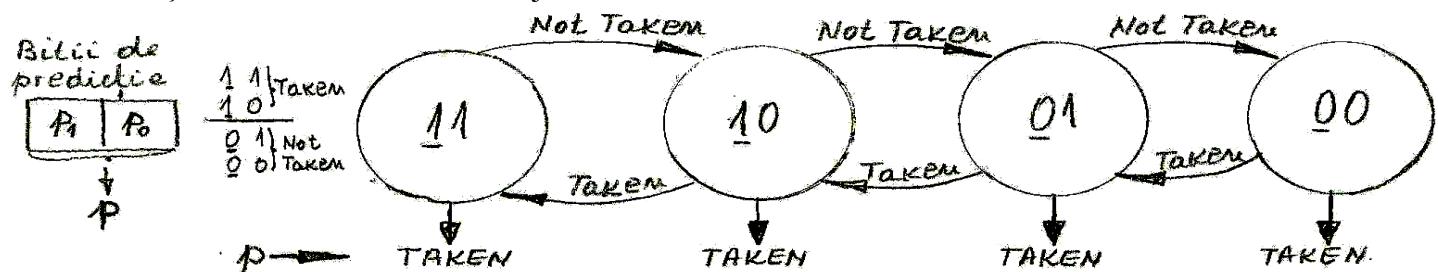
Predictorul pe un bit indică efectuarea saltului prin valoarea unui singur bit, dacă valoarea bitului de predicție, p, din câmpul predicție este înscris în $p=1$ indică pentru instrucțiunea care se extrage în etapa fetch că saltul se va efectua, iar pentru $p=0$ că saltul nu se va efectua; valoarea bitului de predicție a fost înscrisă tot de această instrucțiunea de salt, ultima dată când a fost executată. Dacă la execuția prezentă se găsește bitul de predicție $p=1$ și saltul se efectuează, instrucțiunea de salt, la terminare, reînscrise bitul de predicție tot în valoarea 1, iar dacă saltul nu se efectuează înscrie $p=0$; invers, dacă la execuția prezentă se găsește bitul de predicție $p=0$ și saltul se efectuează, instrucțiunea de salt la terminare înscrie bitul de predicție $p=1$, iar dacă saltul nu se efectuează reînscrise $p=0$. Rezultă că instrucțiunea de salt nu modifică valoarea bitului de predicție dacă se comportă ca și data trecută și modifică valoarea bitului de predicție dacă se comportă diferit față de data trecută. Regula de predicție se bazează pe supozitia că: dacă saltul s-a efectuat ultima dată când s-a executat instrucțiunea de salt atunci se va efectua și la execuția prezentă, iar dacă saltul nu s-a efectuat data trecută nu se va efectua nici de data aceasta.

Acest tip de predictor pe un bit, în câmpul predicție, este simplu de implementat, dar are dezavantajul că pentru buclele din program, care sunt foarte frecvente, predictorul indică greșit de două ori valoarea predicției, prima data la intrarea și a doua oară la ieșirea din buclă, după cum rezultă din figura următoare. La prima intrare în buclă,



la sfârșitul primei iterării evident că valoarea predictorului (pentru instrucțiunea de salt de la sfârșitul buclei) are valoarea zero indicând că saltul nu se face dar în realitate saltul se efectuează, deci instrucțiunea de salt înscrie $p=1$; după parcurgerea ultimei iterării se găsește $p=1$, dar saltul nu se mai face deoarece se ieșe din buclă, deci instrucțiunea de salt înscrie $p=0$; rezultă în total două greșeli în predicția saltului în execuția buclei din program. De fiecare dată când se vine și se ieșe din buclă se repetă cele două greșeli.

Dezavantajul, prin apariția celor două greșeli de predicție la fiecare început și sfârșit de executare a buclelor de program, poate fi eliminat dacă se realizează un **predictor pe doi biți**. De data aceasta câmpul predictor din BTB, pentru cazul general, este structurat ca un numărător reversibil cu saturăție de n biți. Un numărător cu saturăție numără în sens direct de la 000....00 (gol) până la umplerea totală (saturăție) 111....11, iar în continuare stările următoare rămân tot la 111....11 fără a trece la 000....00, la numărarea în sens invers numără de la 111....11 până la golirea totală (saturăție) 000....00, iar în continuare stările următoare rămân tot la 000....000 fără a trece la 111....11. Pentru un numărător cu saturăție modulo 2ⁿ totdeauna instrucțiunea de salt efectuează saltul (Taken) când bitul cel mai semnificativ al numărătorului cu saturăție (predictor) are valoarea 1xx....xx (se situează în jumătatea superioară a intervalului de numărare) și nu efectuează saltul (not Taken) când bitul cel mai semnificativ are valoarea 0xx..xx (se situează în jumătatea inferioară a intervalului de numărare). Trecerea dintre Taken și not Taken se realizează la jumătatea intervalului de numărare.



Pentru un predictor cu doi biți, cu diagrama de stări reprezentată în figura anterioară, numărătorul cu saturăție va indica faptul că saltul se efectuează când are valoarea în jumătatea superioară a intervalului de numărare, adică pentru valorile 11 și 10, iar când are valoarea în jumătatea inferioară a intervalului de numărare, adică pentru valorile 10 și 00, indică faptul că saltul nu se efectuează. Rezultă că numai două greșeli consecutive în efectuarea saltului pot determina modificarea valoii predicției, apărând astfel un interval de histerezis în funcționarea

predictorului. La predictoarele pe doi biți întru-un BTB cu 4096 intrări exactitatea predicției poate atinge o probabilitate de peste 80%.

Pentru determinarea direcția de salt (Taken/not Taken) de la structurile de salt mai complexe, de exemplu cele de la primitiva IF-THEN-ELSE, s-a observat că este afectată și de direcțiile de salt ale altor instrucțiuni de salt executate anterior instrucțiunii de salt curentă, altfel spus pe lângă "istoria locală" a instrucțiunii de salt curente contribuie și "istoria globală" a celorlalte instrucțiuni de salt; Pe baza acestei observații s-au realizat predictoare corelativе care "cuplează" istoria locală a instrucțiunii curente cu influența istoriei globală a instrucțiunilor de salt anterioare.

Predictorul corelativ sau predictor pe două niveluri își bazează predicția pe propria istorie a efectuării salturilor la instrucțiunea curentă (primul nivel, istoria locală) și pe istoria efectuării salturilor de la ultimile k instrucțiuni de salt (al doilea nivel, istoria globală) parcuse în program înainte de a se ajunge la instrucțiunea de salt curentă. Există multe variante de predictoare corelativе [4], [7], cele mai sofisticate predictoare corelativе ridicând exactitatea în determinarea predicției, de la 80 % pentru predictoarelor necorelativе pe doi biți, până la o exactitate de predicție de peste 95% pentru programele cu numere intregi și peste 97% pentru aplicațiile științifice (virgulă flotantă).

5.2 MICROPROCESOARE CU EXECUȚII MULTIPLE

Aceste organizări de μP inițiază într-un tact de ceas execuția mai multor instrucțiuni, deci realizează un CPI<1; (Cycle Per Instruction), dar pentru a fi mai intuitiv în cuantificare de performanță se utilizează inversul acestei metriki, adică numărul de instrucțiuni pe tact IPC =1/CPI (Instruction Per Cycle). În acest capitol se vor prezenta trei tipuri de organizare de μP cu execuții multiple:

1. Microprocesorul superscalar;
2. Microprocesorul VLIW/EPIC
3. Microprocesorul vectorial

5.2.1. Microprocesoare superscalare

Timpul consumat de procesor pentru execuția unui număr de instrucțiuni, N_{instr} , este exprimat de relația

$$CPUtime = N_{instr} \cdot CPI \cdot T_{CLK}$$

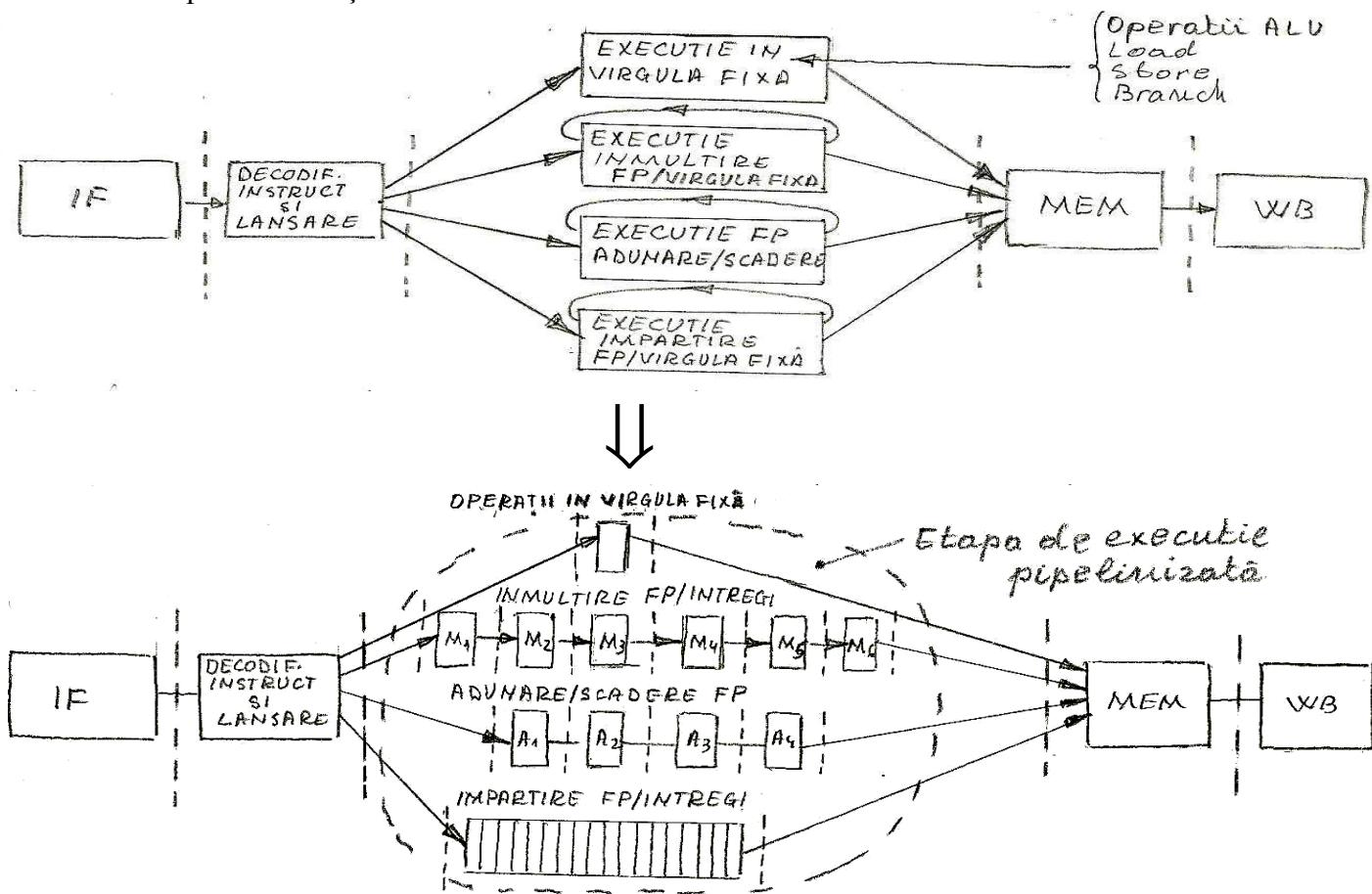
iar pentru micșorarea CPUtime se poate proceda, fie prin micșorarea CPI, fie prin micșorarea T_{CLK} ori ambele componente/factori simultan.

Pentru micșorarea perioadei de ceas, T_{CLK}, sau creșterea frecvenței, f_{CLK}, este necesar să se divida instrucțiunea în cât mai multe etape, încât procesarea din fiecare etapă să poată fi efectuată într-o perioadă de ceas mai redusă, această cale de implementare a dus la procesoare cu foarte multe etape în pipeline referite prin **procesoare superpipelinizate**. Dar, tendința de creștere de frecvență este restricționată de:

- creșterea de putere disipată în tehnologia CMOS, $P_d = C \cdot V^2 \cdot f$
- timpul de propagare prin registrele pipe.

Pentru micșorarea numărului de tacte pe instrucțiune, CPI<1, adică mărirea numărului de **instrucțiuni executate pe cycle/tact, IPC (Instruction Per Cycle)** la valori mai mari de 1, IPC=1/CPI >1, este necesar ca la un anumit moment, în același etapă de procesare, să fie procesate nu una ci mai multe instrucțiuni; procedând în acest mod se face trecerea de la procesorul scalar pipelinizat la **procesorul superscalar**. Dacă pentru o structură clasică de pipe cu cinci etape existența a mai multor instrucțiuni procesate în același etapă de procesare, de exemplu în IF, ID, WB, se poate realiza relativ ușor nu la fel se poate realiza și în etapa de procesare EX. Pentru procesarea mai multor instrucțiuni simultan în etapă EX sunt necesare mai multe unități funcționale, după cum rezultă din figura următoare. Unele unități funcționale, cum sunt cele pentru: operațiile ALU, load, store, branch, pot realiza operația

respectivă într-un singur tact, dar altele cum sunt: înmulțire, împărțire sau operațiile în virgulă flotantă necesită mai multe tacte pentru execuție.



Pentru a obține execuția tuturor instrucțiunilor și din etapa EX într-un singur tact pe instrucțiune trebuie ca, pentru unele operații, unitățile funcționale respective să fie și acestea, la rândul, pipelinizate și, evident, aceste pipe-uri de execuție să funcționeze full. Pentru ca fiecare unitate funcțională pipelinizată să execute o instrucțiune pe tact etapa de execuție EX a proceorului trebuie alimentată simultan cu mai multe instrucțiuni, deci și în etapa fetch trebuie aduse din I-cache mai multe instrucțiuni. Un procesor superscalar la care se aduc din I-cache în etapa fetch m instrucțiuni este referit ca **m-way superscalar**. Transformările descrise anterior, de la un procesor scalar cu o pipelinizare clasică, pe cinci etape, la un m-way superscalar, impune modificarea structurii și în restul etapelor din pipe, nu numai în etapa de fetch. Teoretic, la un m-way superscalar la care s-a crescut de k ori frecvența de ceas față de unul scalar cu aceeași ISA se obține, teoretic, o creștere de viteza de $k \times m$ ori. Se poate tinde spre această creștere teoretică de viteza, mk , doar la valori mici de m și k , pentru că la valori mari ale acestora apar multe aspecte limitative datorită: funcționării, proiectării și tehnologiei.

Structurarea etapelor din pipe este reprezentată în figura următoare, pipe-ul fiind partajat în front-end și back-end. Partea de front-end, care ar corespunde etapelor de IF și ID din pipelinizarea clasică, acum este compusă din etapele: *fetch, decode, rename și dispatch*, iar partea de back-end care ar corespunde etapelor de EX, MEM și WB din pipelinizarea clasică, acum este compusă din etapele: *issue, execute și commit*.

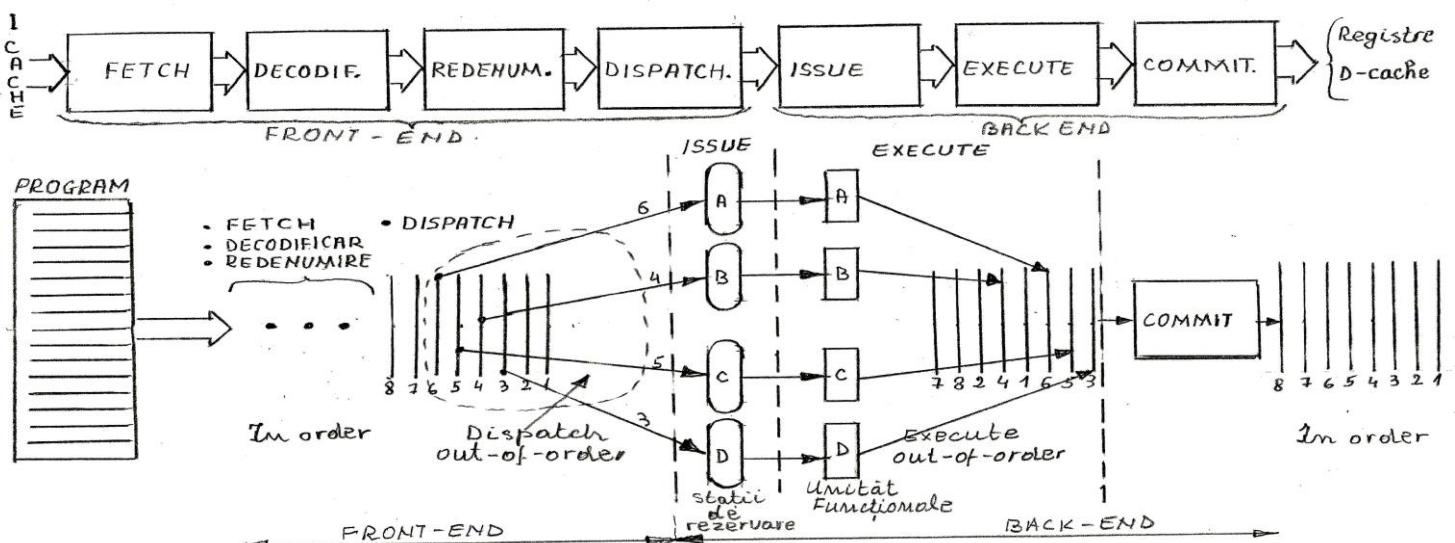
- Instruction fetch.* În această etapă, similar ca la pipe-ul clasic, sunt extrase instrucțiunile din I-cache și depuse într-un buffer de instrucțiuni, IB. Pentru un m-way superscalar sunt extrase și aduse simultan până la m instrucțiuni pe tact, deși unele estimări pentru valorile lui m sunt de 8 sau chiar 16 instrucțiuni extrase și aduse pe tact, pentru implementările curente nu s-a depășit 6 instrucțiuni și nu există tendință de a crește acest număr. În această etapă este de asemenea realizată predicția de salt pe baza unor variante de branch target buffer, BTB, deci este o etapă de *fetch și predicție la salt*.
- Decoding.* Instrucțiunile din IB sunt trecute în etapa de decodificare unde pot exista până la m decodificatoare.
- Renaming registre.* Redenumirea regisrelor este necesară pentru eliminarea situațiile de hazard de tip WAW și WAR, cum s-a arătat în secțiunea 5.1.4.3 prin exemplul cu următorul segment de program.

- I1: $R1 \leftarrow R2 / R3$; această instrucțiune necesită un T_{instr} mai lung decât instrucțiunile I2 și I3, deci ; consumă mai multe tacte.
I2: $R4 \leftarrow R1 + R5$; RAW cu I1
I3: $R5 \leftarrow R6 + R7$; WAR cu I2
I4: $R1 \leftarrow R8 + R9$; WAW cu I1

Utilizând registrele libere R32, R33, R34, R35, și prin redenumire, segmentul de program devine

- I1: $R32 \leftarrow R2 / R3$; această instrucțiune necesită un T_{instr} mai lung decât instrucțiunile I2 și I3
I2: $R33 \leftarrow R32 + R5$; RAW cu I1
I3: $R34 \leftarrow R6 + R7$
I4: $R35 \leftarrow R8 + R9$

pentru care acum au dispărut hazardurile de nume WAW și WAR (dar nu și hazardul fundamental RAW între I1 și I2). Este foarte probabil ca rezultatele instrucțiunilor I3 și I4 să se obțină înaintea rezultatelor de la instrucțiunile I1 și I2, ceea ce face ca valorile din registrele R34 și R35 să fie înscrise în registrele destinație logice, R1 și R5, înainte ca valorile din registrele R32 și R33 să fie înscrise în registrele destinație logice R1 și respectiv R4, deci valoarea calculată în I1 se va înscris în R1 peste valoarea calculată în I4. Valorile în registrele destinație logice trebuie să fie înscrise în ordinea în care sunt în program, adică trebuie să păstreze semnificația programului.



Înscrierea rezultatelor în registrele logice, după execuția instrucțiunilor, conform ordinii din program se realizează cu ajutorul **bufferului de reordonare**, **ROB (ReOrder Buffer)** care este o structură FIFO (First In First Out), adică o structură de regisztr de deplasare. După redenumire, fiecare instrucțiune, în ordinea în care se află în program, este introdusă în coada (tail, ultima poziție) din ROB împreună cu tuplul corespunzător. Tuplul unei instrucțiuni din program înscris, într-o intrare din ROB, cuprinde următoarele parametrii: *flag*, *valoare*, *numele registrului destinație*, *tipul de instrucțiune*, cu următoarele semnificații:

flag – un bit a cărei valoarea 1 indică faptul că execuția instrucțiunii respective s-a terminat, acest flag se înscrise cu valoarea 0 când o instrucțiune din etapa de redenumire este înscrisă/introdusă în coada (tail) din ROB;

valoare – valoarea calculată de instrucțiune;

numele registrului destinație – registrul Rd, în care se va înscris valoarea calculată de instrucțiunea respectivă; *tipul de instrucțiune* – OP (aritmetică, load, store, branch etc).

În urma redenumirii unei instrucțiuni tuplul corespunzător este introdus în intrarea din coada din ROB, cu valoare 0 pentru flag și fără valoarea calculată (NVC):

$$ROB(tail) = (0, NVC, Rd, OP)$$

$$Tail \leftarrow next(tail)$$

După execuția instrucțiunii în pipe, pentru a se putea realiza cuplarea între instrucțiunea redenumită și intrarea aceleiași instrucțiuni în ROB, în etapa de redenumire, se atașează același index/tag atât instrucțiunii redenumite care intră în pipe pentru execuție cât și întrările instrucțiunii respective în ROB. Când execuția instrucțiunii în pipe este completă, în intrarea corespunzătoare din ROB este înscrisă valoarea calculată a rezultatului (NVC) precum și fanionul flag = 1.

4. *Dispatch*. În această etapă instrucțiunile sunt dispecerizate/trimise spre partea de back-end din pipe. Deoarece în etapa de redenumire au fost eliminate doar situațiile de hazard de nume (WAW și WAR) nu și situațiile de hazard structural și de date, RAW, unele instrucțiunile încă nu pot fi trimise în back-end până când nu sunt eliminate și aceste situații de hazard, care ar genera etape de stall. Pentru a nu introduce etape de stall în pipe, dispecerizarea instrucțiunilor spre back-end se face nu în ordinea din program ci **în afara ordinii (out-of-order)**; dispecerizând în afara ordinii, instrucțiunile care pot fi executate, nu trebuie să se aștepte dispecerizarea până cînd pentru instrucțiunile din față lor se rezolvă hazardul RAW sau structural. În segmentul de program anterior, deoarece I2 este în dependență RAW cu I1 și nu poate fi executată până când I1 înscrie R1, deci I2 trebuie să aștepte, atunci se trimit în back-end următoarele două instrucțiuni, I3 și I4. O instrucțiune poate să fie menținută în această etapă și în cazul în care în back-end încă nu există o unitate funcțională liberă pentru operația repectivă. Dispecerizarea se face spre etapa de issue din back-end.

5. *Issue*. În instrucțiunile dispecerizate în etapa anterioară sunt în această etapă (issue) bufferate sub forma unei cozi de așteptare situată înaintea unităților de execuție. Această coadă de așteptare poate fi implementată într-un singur buffer centralizat pentru toate unitățile funcționale (**instruction window**) sau pot fi mai multe cozi de așteptare în buffere separate situate în fața fiecărei unități funcționale, aceste buffere fiind denumite **stații de rezervare** (reservation station). Instrucțiunea prin dispecerizare este copiată în stația de rezervare corespunzătoare operației care trebuie efectuată. Instrucțiunea este menținută în stația de rezervare până în momentul când operanții săi sunt disponibili și unitatea funcțională repectivă este liberă pentru procesare ei. Operanții instrucțiunii care sunt deja disponibili în registre (din banca de registre) sunt imediat copiați în stația de rezervare, iar pentru operanții care nu sunt încă produși de către o altă unitate funcțională se așteaptă până când acești operanții devin disponibili. Când un operand este produs de o unitate funcțională acel operand, chiar înainte de a fi înscris în registrul său logic din banca de registre, dacă este un operand așteptat de alte instrucțiuni, este trimis la toate stațiile de rezervare unde este așteptat (forwarding) ca operand sursă. Când instrucțiunea din stația de așteptare are toți operanții disponibili și unitatea funcțională este liberă este **lansată** (issue) spre execuție, dacă există concurență între mai multe instrucțiuni pentru lansare (au deja operanții sursă disponibili) spre unitatea funcțională se va lansa instrucțiunea care este cea mai veche în stația de rezervare. În general, la mașinile actuale numărul de instrucțiuni lansate pe tact nu este mai mare de săse.

6. *Execuție*. În această etapă, după lansarea unei instrucțiuni din stația de rezervare spre unitatea funcțională respectivă, se realizează execuția instrucțiunii. Deoarece dispecerizarea se face în afara ordinii din program, la fel și lansarea din stațiile de lucru spre unitățile funcționale, iar timpul de execuție nu este egal pentru toate instrucțiunile, toate acestea determină că și completarea/terminarea execuției instrucțiunilor să fie în afara ordinii.

7 *Commitment*. În această etapă instrucțiunile care au fost terminate în afara ordinii sunt readuse/reordonate pentru ca rezultatatele obținute să fie înscrise în registrele logice de destinație/D-cache în succesiunea în care au fost instrucțiunile în program. Reordonarea fiecărei instrucțiuni executate se bazează pe ordinea pe care tuplul instrucțiunii respective îl are în ROB. În etapa de redenumire a operanților atât instrucțiunea cât și tuplul introdus în RO au primit același index/tag, deci pe baza acestuia, instrucțiunea după ce a fost executată și tuplul pot fi cuplate în etapa commit; reamintim că succesiunea/ordinea, de deplasare a tuplurilor, din ROB este identică/copie cu succesiunea instrucțiunilor din program. Când instrucțiunea, *Instri*, a fost completată/executată rezultatul/valoarea este înscris în câmpul NVC (valoarea=NVC) din tuplul din ROB, la fel este înscris și fanionul flag =1, indicând că rezultatul poate fi înscris la destinație. Dar înscriere rezultatului /valoarii în registrul destinație, *Rdi*, poate fi realizată numai când tuplul instrucțiunii, *Instri*, a ajuns în poziția de la ieșirea din structura FIFO (head), ceea ce formal se exprimă:

```
If ((ROB(head) = Instri) and flag(ROB(head)))
    then begin Rdi = rezultat ; head ← next (head) end
```

else repetă același test în următorul ciclu

Procesoarele actuale cele mai performante sunt acestea de tip superscalar la care aranjarea instrucțiunilor (scheduling) pentru procesare este out-of-order și care se realizează în timpul procesării (**dynamic pipeline scheduling**), pentru exemplificare în figura următoare este prezentat procesorul AMD Opteron X4 (Barcelona). Aceste organizări de procesor implică o foarte complexă parte de control, din acestă cauză dezvoltare lor în continuare este pusă sub semnul întrebării deoarece există concurență prin apariția celor de tip multicore (multiprocessors on a chip, CMP), cu partea de control pentru un core relativ mai simplă.

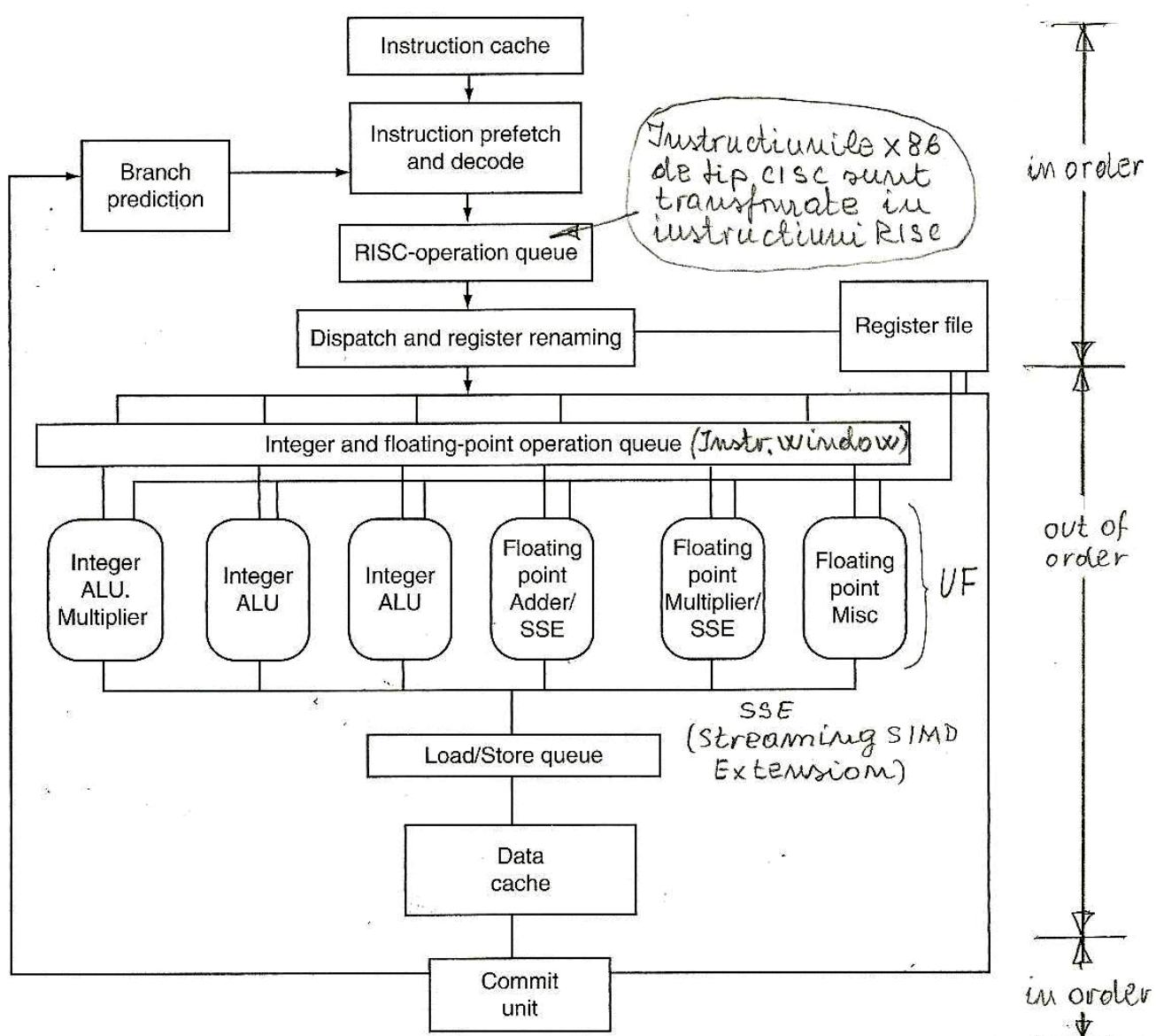
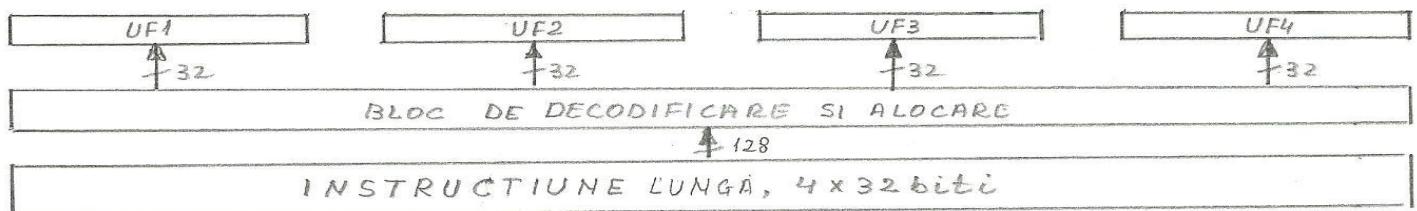


FIGURE 4.74 The microarchitecture of AMD Opteron X4. The extensive queues allow up to 106 RISC operations to be outstanding, including 24 integer operations, 36 floating point/SSE operations, and 44 loads and stores. The load and store units are actually separated into two parts, with the first part handling address calculation in the Integer ALU units and the second part responsible for the actual memory reference. There is an extensive bypass network among the functional units; since the pipeline is dynamic rather than static, bypassing is done by tagging results and tracking source operands, so as to allow a match when a result is produced for an instruction in one of the queues that needs the result.

5.2.2. Micropresesoare de tip VLIW (Very Long Instruction Word)

Ideea procesorului cu instrucțiune lungă are ca origine micropogramarea pe orizontală unde o microinstructiune, prin lungimea sa de peste 400 biți și codificare pe câmpuri pentru diferite componente din cale de date, poate comanda simultan diferite componente.

- Prin procesul de compilare/asamblare instrucțiunile din program care pot fi executate în paralel sunt selectate și jonctionate într-o instrucțiune lungă (un singur cuvânt). Aceste instrucțiuni lungi, realizate de compilator, sunt stocate ca cuvinte distincte în memoria de programe. Plasarea instrucțiunii într-o anumită poziție în cuvântul instrucțiune lungă fixeză deja unitatea funcțională pe care acea instrucțiune va fi executată. O structurare de principiu, pentru un µP de tip WLIW cu patru unități funcționale, este prezentat în figura următoare.

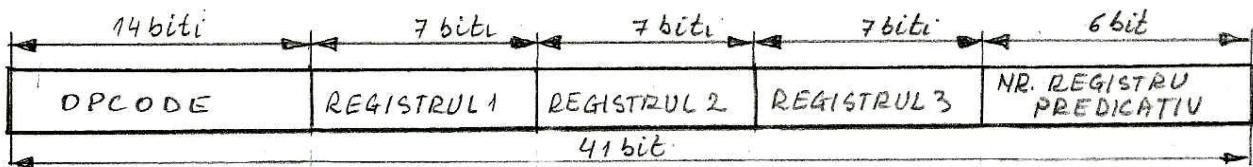


- Dacă în cuvântul lung este inclus și un câmp prin care se prezintă (procesorului) explicit și informație despre procesarea paralelă a instrucțiunilor din cuvântul lung denumirea este de arhitectură **EPIC** (Explicit Parallel Instruction Computer).
- Performanțele acestor procesoare este esențial determinată de „inteligenta” compilatoarelor. Prin angajarea masivă a compilatorului în realizarea unui înalt grad de paralelism în execuție, în raport cu procesoarele superscalare unde obținerea unui înalt grad de paralelism se realizează (în hard) prin rearanjarea dinamică a instrucțiunilor în pipe, se pot obține procesoare care sunt mult mai simple (rearanjarea statică a instrucțiunilor fiind mai simplu de realizat decât rearanjarea dinamică). Cauzele care reduc obținerea unui grad ridicat de paralelism (în execuție) la nivel de instrucțiune (ILP) sunt:
 - hazardul de date (RAW) și de nume (WAW, WAR);
 - hazardul de control;
 - latența memoriei.

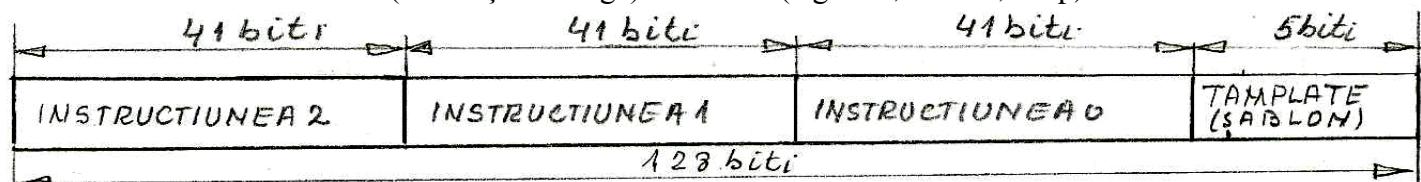
Un exemplu de arhitectură WLIW (EPIC) este cea dezvoltată în cooperare de Intel și Hewlett-Packard, referită **ARHITECTURA IA-64**, realizată comercial de Intel prin procesoarele Itanium 1 și Itanium 2, prezentată în continuare.

- Cerințe impuse arhitecturii IA-64
 - compatibilitate completă cu arhitectura IA-32;
 - scalabilitate pentru o familie de procesoare (într-o gamă largă de configurații și implementări);
 - calculabilitate completă în lungimea de (operanzi) 64 biți.
- Caracteristicile arhitecturii IA-64
 - explicitarea paralelismului la nivel de instrucțiune cod mașină, ILP (Instruction Level Parallelism);
 - suport (arhitectural) pentru compilator în realizarea explicitării la nivel cod mașină, ILP, constând din:
 - 1. predicare completă (pentru toate instrucțiunile), 2. controlul execuțiilor speculative;
 - resurse hardware pentru execuția paralelă: 1. 128 registre generale (de 64 biți), 2. 128 registre floating point (de 82 biți), 3. 64 registre predicative (de 1 bit), 4. 8 registre pentru calcularea salturilor (de 64 biți), 5. 128 registre suport pentru aplicații (de 64 biți).

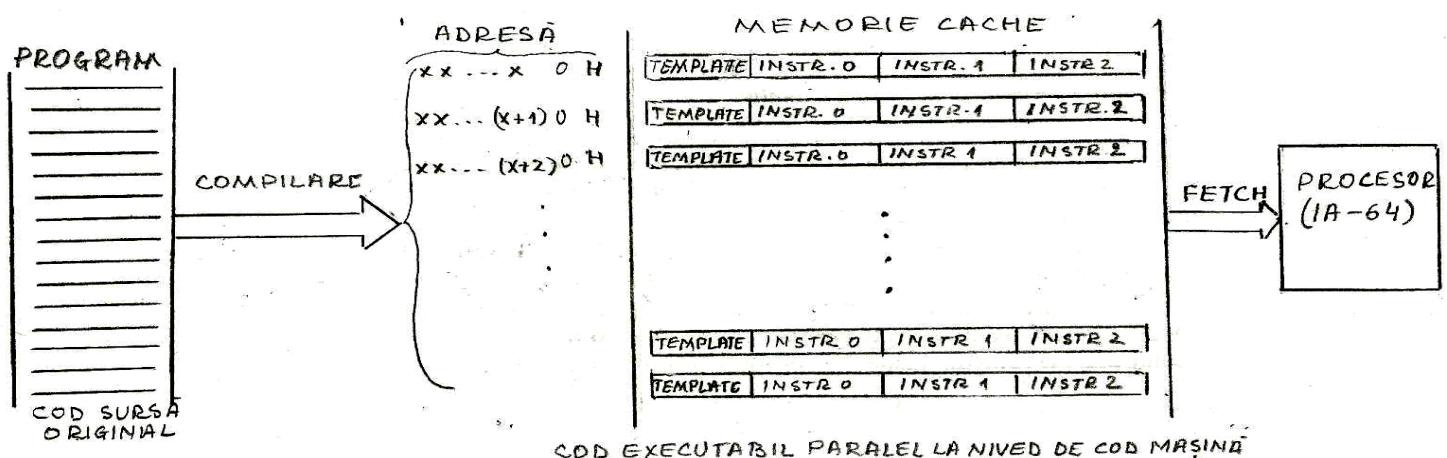
- Formatul instrucțiunii



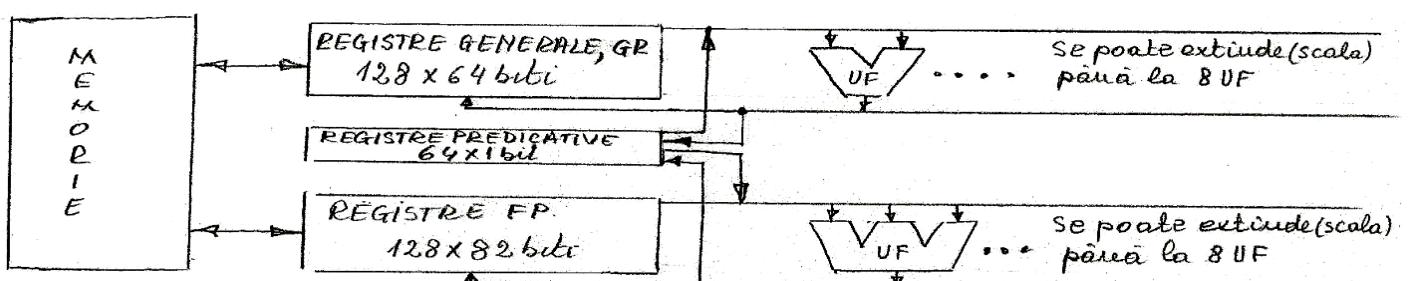
- Formatul cuvântului (instrucțiune lungă) – **Bundle** (legătură, maldăr, snop)



Deoarece un bundle în memorie are lungimea de 16 byte (128biți), adresarea se realizează din 16 în 16 respectând relația ADRESA(modulo 16) = 0. Pentru o adresare corectă cuvântul ADRESA are totdeauna ultimii patru biți egali cu zero, adresele accesate succesiv sunt: xx....xx0000, xx....x(x+1)0000, xx....x(x+2)0000; spațiul de adresare este de 64 biți (2^{64} adrese), cum este reprezentat în figura următoare



- Organizarea procesorului IA-64 (posibilitate de scalare)



- Aportul compilatorului

Procesoarele WLIW, respectiv arhitectura EPIC, au fost gădite ca o alternativă la procesoarele superscalare care prin rezolvările excesive de tip diamic în pipe au dus la foarte complexe implementări ale unităților de control. Undeva, se repetă istorie de la începutul anilor '80 când arhitectura CISC (o solicitare puternică a hard-ului "hardul a intrat în soft") a fost înlocuită cu arhitectura RISC (o utilizare extensivă a soft-ului (compilatorului) "softul a intrat în hard"); acum, la fel, se face o rezolvare că mai mult static, adică în etapa de compilare. Compilatorul, în funcție de maparea care se poate realiza între tipul de instrucțiune și unitatea funcțională, pe care instrucțiunea se poate procesa, cum este prezentat în tabelul din figura următoare, analizează instrucțiunile din codul sursă al programului și selectează instrucțiunile care se pot grupa către trei,

MAPAREA TIPURILOR DE INSTRUCȚIUNI PE TIPUL CORESPONDENT DE UNITATE FUNCȚIONALĂ

TIPUL DE UF	I sau M care este disponibilă	I	M	B	F	I sau B depinut de operatie
TIPUL DE INSTRUC.	A	I	M	B	F	X
DESCRISEREA OPERAȚIEI	Aritmetică, logică sau comparație	Operări cu intrări de tip nou ALU	Acces la memorie sau move GR/FR	Salt sau call	Operări în virgulă flotantă	Operări care se execută pe două sloturi în bundle

formând cuvântul lung, un bundle. Dar prin această grupare, compilatorul totodată generează și o informație – **Template** (cinci biți) – prin care îi comunică procesorului în funcție de poziția instrucțiunii(slot) în bundle pe ce tip de unitate de procesare se va executa insstrucțiunea . Tipurile de coduri de template (cuvinte de cinci biți), repectiv tipurile de unitățile funcționale (I-integer, M-acces memorie, A- aritmetice, B-branch, F virgulă flotantă) care sunt asignate pentru fiecare template sunt prezentate în tabelul următor.

TIPURI DE TEMPLATE IN CUVÂNTUL LUND (BUNDLE)

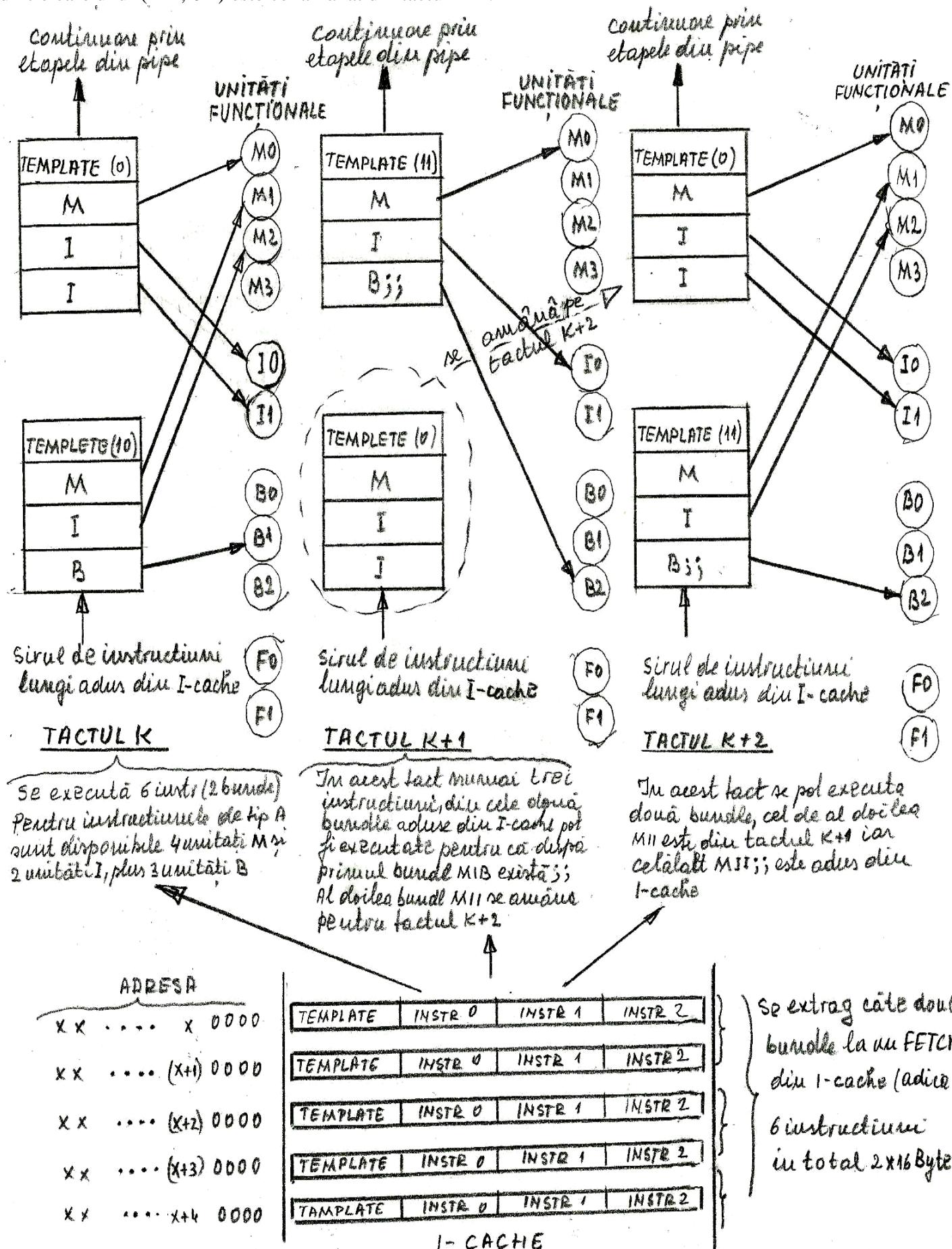
COD TEM- PLET (HEX)	SLOT	TIP UF	COD TEM- PLET (HEX)	SLOT	TIP UF	COD TEM- PLET (HEX)	SLOT	TIP UF	COD TEM- PLET (HEX)	SLOT	TIP UF	COD TEM- PLET (HEX)	SLOT	TIP UF	COD TEM- PLET (HEX)	SLOT	TIP UF	COD TEM- PLET (HEX)	SLOT	TIP UF
0 1	I	0 M	4 1	L	8 1	M	C 1	F	10 1	I	14 1	*	18 1	M	1C 1	M	0 1	M	1F 2	B
2 1		X	2		2	I	2		2	B;;	2		2		2	B;;	0 1	M	1D 2	B;;
0 1	I	0 M	0 1	L	9 1	M	D 1	F	11 1	I	15 1	*	19 1	M	1D 1	M	0 1	F	1 2	M
2 1	I;;	5	2	X;;	2	I;;	2	I;;	2	B;;	2		2		2	B;;	0 1	M	1E 1	B;;
0 1	T;;	0 *	0 1	M;;	A 1	M;;	E 1	M	12 1	B	16 1	B	1A 1	B	1E 1	*	0 *	0 1	*	0 2
2 1	T;;	6	1		2	I	2	F	2	B	2	B	2	B	2		0 *	0 1	1F 1	*
0 1	I;;	0 *	0 1	M;;	B 1	M	F 1	M	13 1	B	17 1	B	1B 1	B	1F 1	0 *	0 1	1F 2	*	
2 1	I;;	7	2		2	I;;	2	F;;	2	B;;	2	B;;	2	B;;	2		0 *	0 1	1F 1	

* Rezervat pentru extensii arhitecturale viitoare (acestea sunt la nivelul anului 2003).

Se extrag simultan din I-cache două bundle (6 instrucțiuni, 6x16 bytes) și sunt trimise spre execuție. Din tabelul anterior, cu codurile template, se observă că după unele unități funcționale, utilizate pentru acel template, este introdus simbolul ; ; (split) . Simbol **split** limitează/separă până la ce poziție (unde este plasat split-ul) instrucțiunile din cele două bundle introduse în execuție se pot procesa în parallel.

În figura următoare este exemplificat modul de procesare paralelă pe Itanium. Pe tactul k se extrag din I-cache două bundlele unul cu codul template 0 H iar celălat cu codul 10 H. Pentru toate cele șase instrucțiuni sunt disponibile unități funcționale, deci toate cele șase instrucțiuni se pot procesa în paralel. Pe tactul k+1 se extrag următoarele două bundle cu codurile template 11H și 0H, dar în codul MIB(11H) există și simbolul ; ; (split) ceea ce înseamnă că se execută în paralel doar primele trei instrucțiuni pe unitățile funcționale M, I, B iar instrucțiunile din al doilea bundle care necesită unitățile funcționale M, I, I așteaptă până la tactul următor (aceasta corespunde

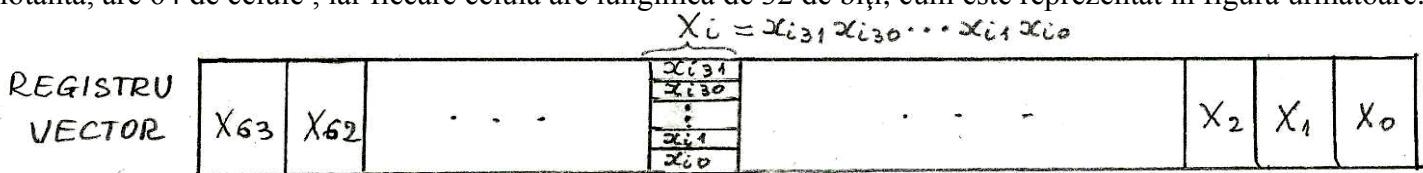
primelor implementări ale procesorului Itanium). În tactul k+2 se procesează din nou în paralel două bundle, dintre care unul (MII, 0H) este cel amânat din tactul k+1.



5.2.3 Procesoare vectoriale

Aceste tipuri de procesoare au fost concepute pentru lucrul cu vectori. În ultimii ani procesoarele vectoriale sunt utilizate ca și coprocesoare pe lângă un procesor în segmentul de aplicații multimedia, pentru procesarea strimurilor de eșantioane de date de tip imagine sau sunet; mai nou acest segment este acoperit din ce în ce mai mult de noile coprocesoare de tip GPU.

Considerăm un vector $X = (X_{63}, X_{62}, X_{61}, \dots, X_1, X_0)$ care este o matrice unidimensională (coloană), cu 64 de componente, fiecare componentă fiind un număr în virgulă flotantă exprimat sub forma unui cuvânt de 32 de biți, conform standardului IEEE754. Un registru –**registru vector**– în care se înscrie un astfel de vector, în virgulă flotantă, are 64 de celule, iar fiecare celulă are lungimea de 32 de biți, cum este reprezentat în figura următoare.



Tipurile de operații efectuate cu vectori sunt sistematizate în următorul tabel:

OPERAȚIA	EXAMPLE
$Y_i = f_1(X_i)$	$f_1 = \cos X_i, \sqrt{X_i}, e^{X_i}$ (funcții transcendente)
Scalar = $f_2(X_i)$	$f_2 = \sum X_i$ (intrare vector ieșire scalar)
$Z_i = f_3(X_i, Y_i)$	$f_3 = X_i + Y_i, X_i - Y_i, X_i \cdot Y_i$ (intrare vector ieșire vector)
$Y_i = f_4(\text{scalar}, X_i)$	$f_4 = \text{scalar} \times X_i$ (combinare scalar-vector)

De exemplu, pentru înmulțirea a două matrici, $Z = X \times Y$; pentru elementul rezultat, $z_{ij} = \sum_{k=1}^n x_{ik} \cdot y_{jk}$, se realizează întâi operație de tip f_3 (înmulțire) apoi operația de tip f_2 (ieșire scalar). De asemenea, reamintim efectuarea operației de adunare/scădere în virgulă flotantă (pentru un vector cu componentele exprimate în virgulă flotantă), se consideră $X_E < Y_E$

$$X + Y = X_s \cdot B^{X_E} + Y_s \cdot B^{Y_E} = [X_s \cdot B^{(X_E - Y_E)} + Y_s] B^{Y_E}$$

$$X - Y = X_s \cdot B^{X_E} - Y_s \cdot B^{Y_E} = [X_s \cdot B^{(X_E - Y_E)} - Y_s] B^{Y_E}$$

Pentru următoarele două numere $X = X_s \cdot B^{X_E} = 0,3 \cdot 10^2$; $Y = Y_s \cdot B^{Y_E} = 0,2 \cdot 10^3$ (sub forma normalizată)

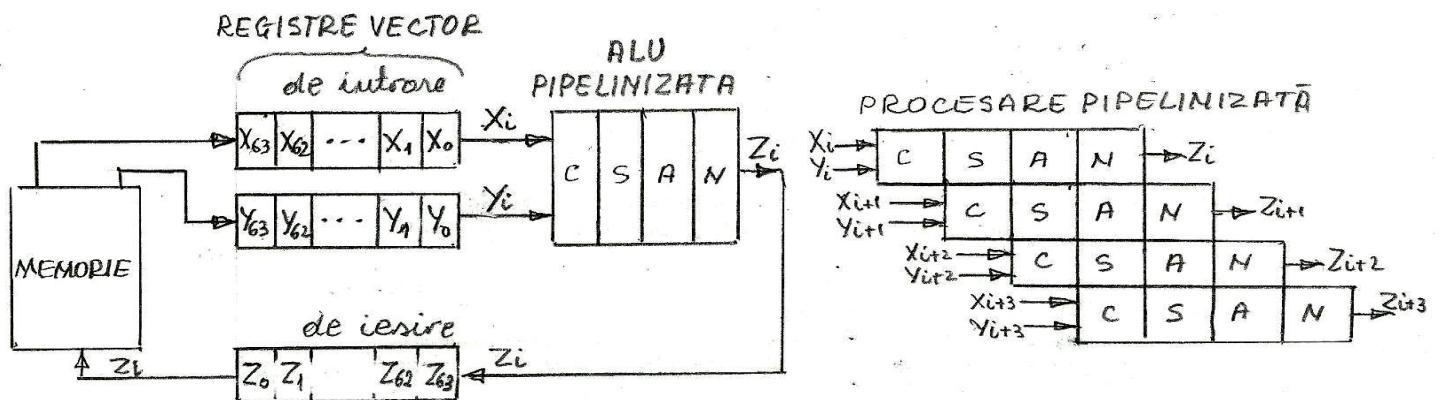
$$X + Y = [0,3 \cdot 10^{(2-3)} + 0,2] 10^3 = [0,3 \cdot 10^{-1} + 0,2] = 0,23 \cdot 10^3 (= 230)$$

$$X - Y = [0,3 \cdot 10^{(2-3)} - 0,2] 10^3 = [0,3 \cdot 10^{-1} - 0,2] = -0,17 \cdot 10^3 (= -230)$$

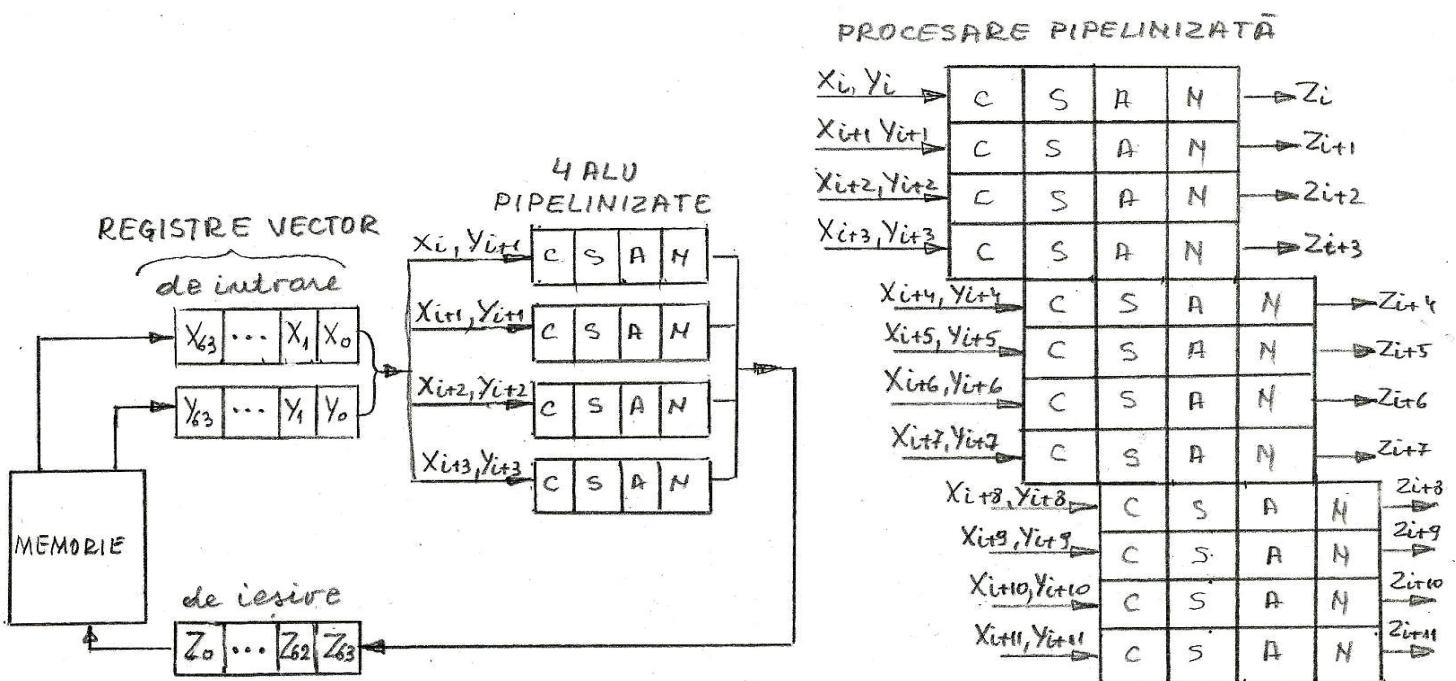
Etapele realizării operației de adunare, pentru efectuarea acesteia în pipeline, sunt [C], [S], [A], [N]:

1. **C** – *compară exponenții*, compară valoarea celor doi exponenți prin scăderea valorii lor ($X_E - Y_E$);
2. **S** – *shiftare mantisă*, se obține prin deplasarea mantisei care are exponentul mai mic ($X_E < Y_E$) spre dreapta cu un număr de poziții egal cu diferența ($X_E - Y_E$);
3. **A** – *adunarea mantiselor*;
4. **N** – *normalizarea rezultatului*, se obține prin deplasarea mantisei rezultante cu un număr de poziții până când aceasta are prima cifră după virgula zecimală diferită de zero, corespunzător se scade/crește exponentul.

- Organizarea de principiu a unui procesor vectorizat cu o singură ALU pipelinizată. Cei doi operanzi vector X și Y sunt citiți din memorie și depuși în registrele vector de intrare, cum este reprezentat în figura următoare (se consideră vectori cu 64 de componente). Apoi, pe fiecare tact componentele de același rang $X_i, Y_i, i = 0, 1, 2, \dots, 62, 63$ din registrele de intrare sunt trimise la unitatea funcțională pipelinizată (pe patru etape [C], [S], [A], [N]). După primele patru tacte (latența pipe-ului) se obține la ieșirea ALU, pe fiecare tact, câte o componentă a vectorului rezultat $Z_i, i = 0, 1, 2, \dots, 62, 63$ care este înscrisă în registrul vector de ieșire.



- Organizarea de principiu a unui procesor vectorial cu multiple ALU pipelinizate. Cu această organizare, spre deosebire de organizarea anterioară când era o singură unitate funcțională, fiind patru unități funcționale în paralel se trimit din registrele de intrare spre cele patru pipe-uri pe fiecare tact câte patru perechi de componente vector, $X_i, Y_i; X_{i+1}, Y_{i+1}; X_{i+2}, Y_{i+2}; X_{i+3}, Y_{i+3}$, obținându-se pe tact câte patru componente ale vectorului rezultat $Z_i; Z_{i+1}; Z_{i+2}; Z_{i+3}$, cum este reprezentat în figura următoare.



EXEMPLUL 5.2. Pe procesorul MIPS convențional să se realizeze procesarea buclei DAXPY, de asemenea să se proceseze și pe procesorul MIPS extins vectorial.

Pentru procesarea de tip vectorial, foarte frecvent, este întâlnită expresia DAXPY (vezi f4 și f3 din tabelul anterior)

$$Y = a \cdot X + Y; \quad (\text{DAXPY - Double precision } a \cdot X \text{ plus } Y)$$

ARHITECTURA ȘI ORGANIZAREA MICROPROCESOARELOR – Gh. Toacse

în care: X, Y – vectori (în acest caz fiecare cu 64 componente/cuvinte), fiecare cuvânt cu lungimea $2 \times 32\text{biți} = 64\text{ biți}$ (dublă precizie – d simbol în mnemonicul instrucțiunii);
a – scalar, dublă precizie ($2 \times 32\text{biți} = 64\text{ biți}$).

Extensia procesorului MIPS la un procesor MIPS vectorial necesită introducerea de registre vector, cum este prezentat în organizările anterioare, plus adăugarea în ISA a unor instrucțiuni vectoriale, care vor avea în opcode litera v, de exemplu:

addv.d – adună doi vectori în dublă precizie;
addvs.d – adună la toate componentele vectorului un scalar, în dublă precizie;
lv, sv – load, store un vector (toate componente, în acest caz 64 de cuvinte fiecare de 64 biți).

Adresa de început din memorie pentru vectorul X este în registrul \$s0, pentru vectorul Y în registrul \$s1, iar adresa din memorie pentru scalarul a este A.

Codul DAXPY pentru MIPS convențional:

l.d	\$f0, A(\$zero)	; se încarcă scalarul a în registrul floating-point \$f0
addiu	\$r4, \$s0, 512	; stabilește adresa limită superioară pentru componentele vectorului X
Buclă: l.d	\$f2, 0(\$s0)	; încarcă în \$f2 componenta X(i)
mul.d	\$f2, \$f2, \$f0	; a·X(i)
l.d	\$f4, 0(\$s1)	; încarcă în \$f4 componenta Y(i)
add.d	\$f4, \$f4, \$f2	; a·X(i) + Y(i)
s.d	\$f4, 0(\$s1)	; înscrive în memorie a·X(i) + Y(i)
addiu	\$s0, \$s0, 8	; adresa următoare din memorie pentru componenta X(i)
addiu	\$s1, \$s1, 8	; adresa următoare din memorie pentru componenta Y(i)
subu	\$t0, \$r4, \$s0	; calculează diferența până la limita superioară
bne	\$t0, \$zero, Buclă	; dacă nu s-a atins limita superioră procesează componenta următoare

Codul DAXPY pentru MIPS extins vectorial:

l.d	\$f0, A(\$zero)	; se încarcă scalarul a în registrul floating-point \$f0
lv	\$v1, 0(\$s0)	; în reg. vector \$v1 se încarcă toate cele 64 de componente ale vect. X
mulvs.d	\$v2, \$v1, \$f0	; \$v2 ← a·X
lv	\$v3, 0(\$s1)	; în reg. vector \$v1 se încarcă toate cele 64 de componente ale vect. Y
addv.d	\$v4, \$v2, \$v3	; \$v4 ← a·X + Y
sv	\$v4, 0(\$s1)	; înscrive rezultatul în memorie

Comparând cele două programe rezultă:

– programul pe procesorul convențional aduce (fetch) din memorie aproape 600 de instrucțiuni, pe când la varianta extinsă vectorial sunt aduse din memorie doar șase instrucțiuni, deci diferență de transfer pe magistrală. Aceasta rezultă deoarece varianta clasică iterează în buclă de 64 de ori; din fiecare iterație ultimile patru instrucțiuni (addiu, addiu, subu, bne) constituie regia buclei, nu realizează efectiv calculul pentru DAXPY.

– în programul vectorial operațiile de calcul, care se efectuează prin cele două instrucțiuni **mulvs.d** **\$v2, \$v1, \$f0 ; \$v2 ← a·X** și **addv.d** **\$v4, \$v2, \$v3 ; \$v4 ← a·X + Y**, ”curg” neîntrerupt de 64 de ori fără a se efectua 64 de iterații în buclă. Mai mult, în programul convențional trebuie introduse stall-uri între instrucțiunea a patra (mul.d \$f2, \$f2, \$f0) și a șasea (add.d \$f4, \$f4, \$f2), la fel între a șasea (add.d \$f4, \$f4, \$f2) și a șaptea (s.d \$f4, 0(\$s1)), deoarece există hazard RAW. În programul vectorial o dependență de tip RAW introduce stall-uri doar pentru prima componentă a vectorului, după care celelalte componente ale vectorului curg continuu în pipe, deci stall-uri pe un vector în raport cu un stall-uri pe interație la procesarea convențională; în acest exemplu sunt cam de 64 ori mai puține stall-uri la procesarea vectorială în raport cu cea convențională. Totuși, pentru programul vectorial trebuie considerat și timpul de încărcare din memorie în registrele de intrare a celor doi vectori; pentru o încărcare paralelă a vectorilor cu o componentă pe tact sunt necesare 64 tacte.

5.3 SISTEME MULTIPROCESOR

Visul, din totdeauna, al arhitecților din domeniul calculatoarelor a fost să realizeze calculatoare prin integrarea de procesoare, de preferat simple, astfel încât să se obțină un calculator scalabil care să ofere performanțe și preț la cerere. Până în prezent, dezvoltarea calculatoarelor a fost evolutivă și s-a bazat aproape exclusiv pe **procesarea secvențială**, iar organizarea realizată pe bază de monoprocesor (arhitectura de tip von Neumann) **este organizare de tip serie**. Performanțele de viteză ale procesoarelor s-au bazat în primul rând pe paralelismul la nivel de instrucțiune, ILP, prin pipelinizare: sub formă de execuție out-of-order la organizările superscalare (rearanjare dinamică), prin rearanjare statică la procesoarele VLIW (EPIC) sau, într-un fel, prin procesarea vectorială.

Sistemele multiprocesor necesită schimbarea modului de abordare deoarece prin existența a n procesoare rezultă o **organizare paralelă**, iar pentru acestea **softul este concurențial** (în general, programare paralelă). Sistemele multiprocesor când nu sunt realizate cu procesoare discrete ci sunt pe singur chip, în care sunt integrate procesoarele componente, sunt referite ca **procesoare multicore** (multicore processors) sau **CMP** (Chip MultiProcessors). Procesarea pe sisteme multiprocesor poate fi realizată în următoarele două variante:

1. **procesarea paralelă de program independent** (job-level parallelism sau process-level parallelism, TLP- Task Level Parallelism) – când pe fiecare procesor component sunt executate programe sau părți de program independente;
2. **procesarea paralelă a unui program** (parallel processing program, DLP- Data-Level Parallelism) – când date ale aceluiași program sunt procesate în paralel (simultan) pe mai multe procesoare componente.

Domeniul sistemelor multiprocesor, a fost explorat până în prezent mai mult sub forma unor abordări experimentale, dar cu certitudine acesta este domeniul care va deveni normă în viitor, și care, poate, va duce la o dezvoltare revolutivă. Pentru sistemele multiprocesor suportul tehnologic, conform legii lui Moore, există, dar mai trebuie ca și componenta de programare paralelă să devină curentă. Aspectele care sunt încă deficitare în elaborarea de soft concurențial, prin care să fie utilizate din plin cele n procesoare ale sistemului, sunt:

- repartizarea sarcinilor pe fiecare procesor;
- încărcarea echilibrată a procesoarelor;
- sincronizarea sarcinilor;
- comunicarea între procesoare (sarcini).

S-ar crede, și s-ar dori, ca creșterea performanței de viteză, CV (speed-up), să fie liniară cu creșterea numărului de procesoare din sistem, dar în realitatea această creștere este mult subliniară. Pentru oricare program cu execuție paralelă există și un anumit segment de execuție secvențială și de sincronizare. Presupunem că pentru execuția acelui program pe un singur procesor segmentul care se execută secvențial necesită un timp s , iar segmentul în care există potențialul de execuție paralelă, dar acum se execuă secvențial pe un singur procesor, necesită un timp p . Conform legii lui Amdahl, dacă timpul total de execuție al programului pe un singur procesor este $T_1 = s + p$, iar timpul total de execuție pe un sistem multiprocesor cu n procesoare este $T_2 = s + p/n$, atunci rezultă creșterea de viteză egală cu

$$CV_n = \frac{T_1}{T_2} = \frac{s + p}{s + p/n}$$

Această relație arată că pentru sistemele multiprocesor creșterea de viteză nu este o dependență liniară funcție de creșterea numărului n de procesoare componente, ci este o dependență subliniară; s-ar obține o dependență liniară, $CV_n = n$, numai când întreg programul poate fi procesat în paralel, adică nu există segmentul secvențial, $s = 0$. Dacă în această relație se normalizează timpul la timpul de execuție serială pe un sistem monoprocesor, $T_1 = s + p = 1$, adică se consideră procesarea secvențială pe monoprocesor ca unitate, iar numărul de procesoare este foarte mare ($n \rightarrow \infty$) atunci maximul creșterii de viteză (în raport cu unitatea, monoprocesorul) este

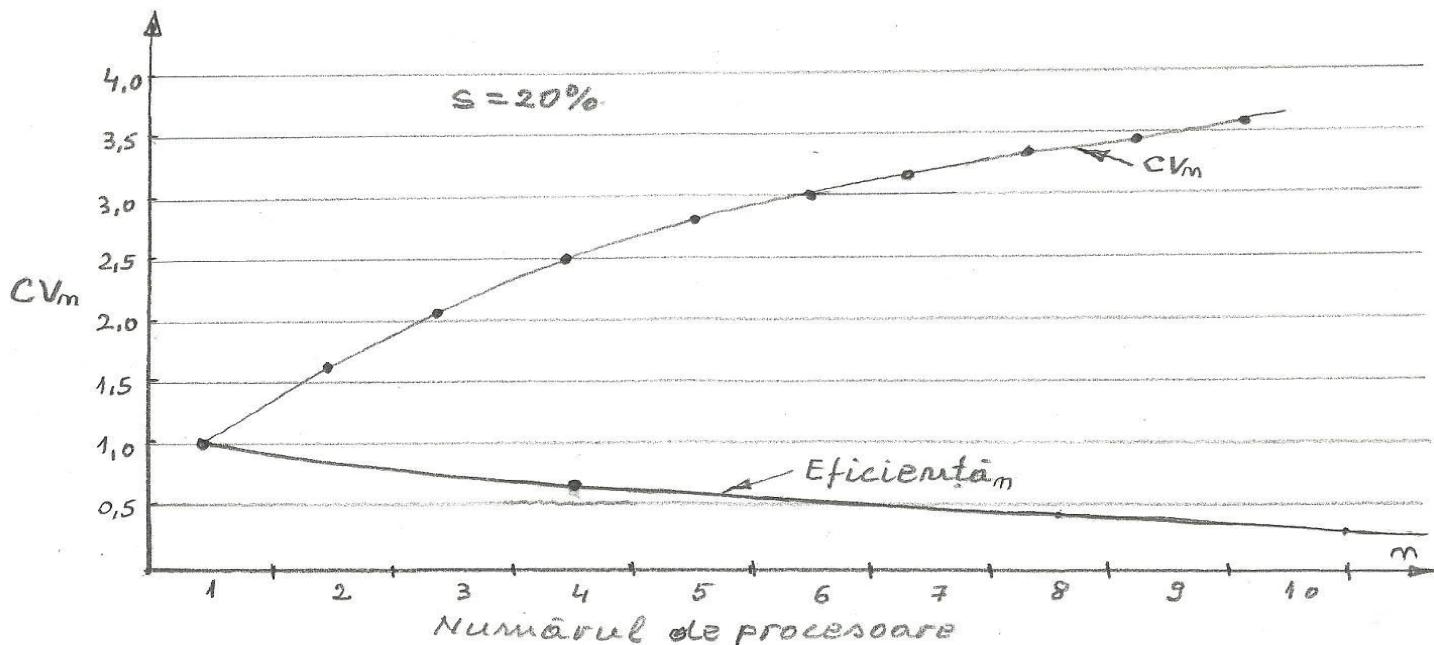
$$CV = \frac{1}{s + \frac{p}{\infty}} = \frac{1}{s}$$

De exemplu, dacă intervalul de timp din program pentru procesarea secvențială din timpul total de execuție al întregului program pe monoprocesor este $s = 0,2$ (20%), atunci creșterea maximă de viteză poate atinge valoarea $1/0,2 = 5$, când ($n \rightarrow \infty$). În figura următoare este reprezentată grafic dependența creșterii de viteză, CV , în funcție de numărul n de procesoare din sistem, când $s = 20\%$; panta pentru CV se reduce mult când n este mai mare de 6-7 procesoare. De exemplu, pentru a atinge un $CV = 4$ sunt necesare 16 procesoare, cu 64 de procesoare se obține $CV = 4,7$, iar pentru 128 de procesoare creșterea de viteză atinge doar valoarea de $4,85 (< 5)$. Aceste rezultate se obțin doar când toate accesările la memorie cache sunt cu succes și împărțirea segmentului p , de procesare paralelă din program, pe fiecare procesor se face în părți egale cu p/n (încărcare echilibrată), ceea ce în realitate se realizează greu. Să presupunem că segmentul de procesare paralelă (consumă un timp p pe un uniprocesor) este executat de 100 procesoare, dar unuia dintre procesoare i se alocă nu 1% (0,01p) din sarcină ci 2% (0,02p), deci fiecare dintre celelalte 99 de procesoare va procesa un timp egal cu $(1p - 0,02p)/99 = 0,98p/99 = 0,009899p$. Timpul cât procesorul mai încărcat va mai procesa, după ce procesarea paralelă s-a terminat pe celelalte 99 de procesoare, va fi egal cu $0,02p - 0,009899p = 0,010101p$, iar acest timp este, de fapt, un timp de procesare serială care se însumează cu timpul serial s al programului, rezultând un timp total de procesare secvențială mai lung ($s + 0,010101p$), deci o creștere de viteză mai mică.

Creșterea de viteză prin adăugarea de resurse (procesoare) reflectă creșterea de performanță a sistemului. Adar nu reflectă și eficiența utilizării acestor resurse. Se poate introduce o metrică și pentru eficiență, $Eficiență_n$, sub forma

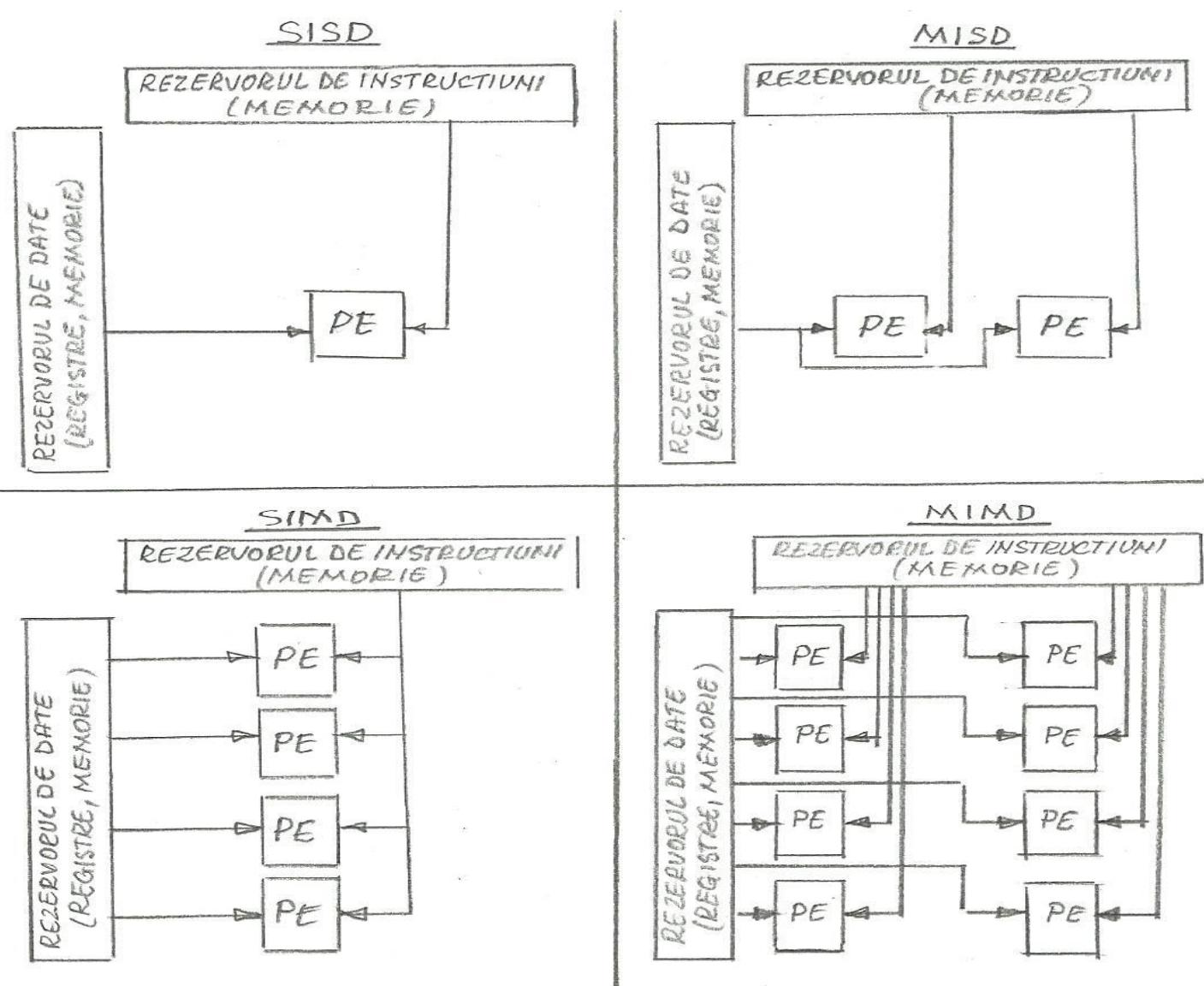
$$Eficiență_n = CV_n/n$$

a cărei reprezentare grafică, în funcție de numărul de procesoare, este suprapusă peste reprezentarea creșterii de viteză în figura următoare. Astfel se poate corela creșterea de viteză cu valoarea eficienței obținute prin adăugarea de procesoare în sistem, de exemplu: pentru patru procesoare creșterea de viteză este de 2,5 iar eficiența obținută pe procesor este de 0,6; pentru 16 procesoare creșterea de viteză este de 4 iar eficiența obținută pe procesor este de 0,25; iar pentru 64 de procesoare creșterea de viteză este de 4,7 iar eficiența obținută pe procesor este de 0,075. Rezultă că prin adăugarea de procesoare eficiența pe fiecare procesor scade.



Există o clasificare a organizării calculatoarelor, introdusă de M. Flynn în 1966, în funcție de numărul de instrucțiuni din fluxul de instrucțiuni și de numărul de date din fluxul de date procesate, care se repartizează la elementele de procesare, PE(Processing Element); astfel pot fi identificate următoarele patru organizări cu reprezentarea din Figura următoare:

- Single Instruction, Single Data (SISD);
- Single Instruction, Multiple Data (SIMD);
- Multiple Instruction, Single Data (MISD);
- Multiple Instruction, Multiple Data (MIMD).



Organizarea SISD, o singură instrucțiune și o singură data, corespunde organizării convenționale de uniprocesor. MISD, un stream de instrucțiuni și o singură dată, nu a fost construită, nu are aplicație comercială.

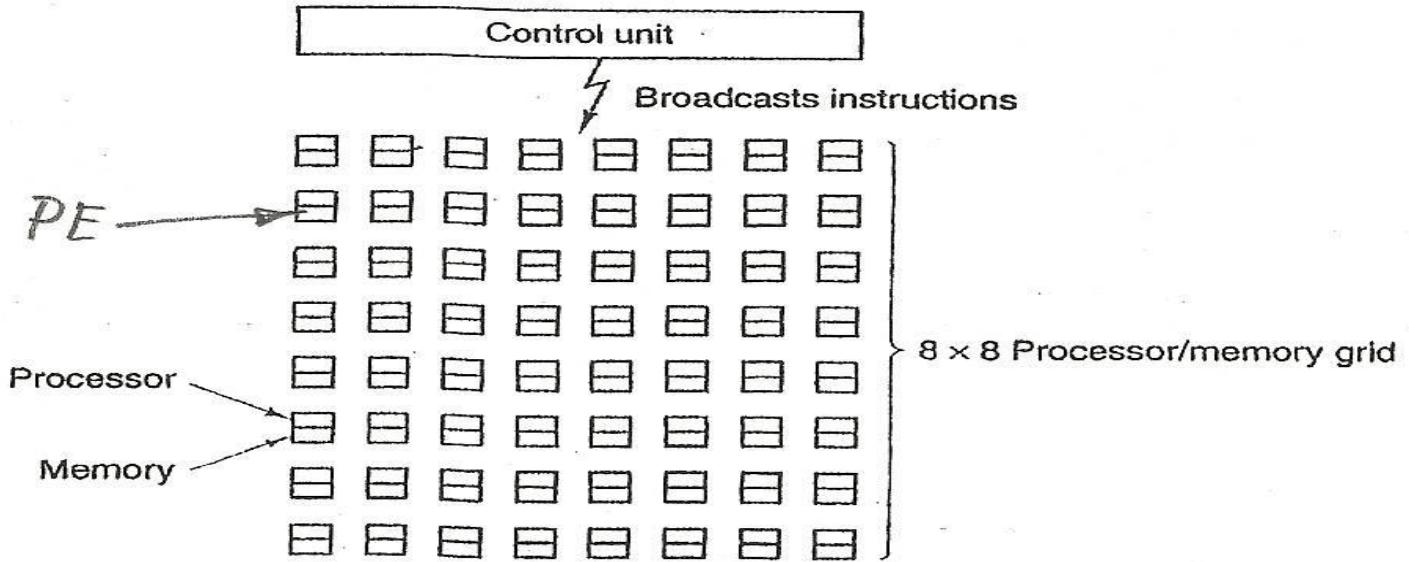
SIMD, aceeași instrucțiune este aplicată la mai multe PE-uri operând asupra mai multor strumuri de date, deci se exploatează paralelismul la nivel de date, DLP (Data-Level Parallelism), aceeași operație se execută simultan pe fiecare PE, dar cu alte date. Fiecare PE are memoria sa (de date), iar controlul calculatorului acceseează o memorie de instrucțiuni comună de unde acceseează (fetch) și dispecerează o singură instucțiune la toate PE-urile componente. Această organizare acoperă arhitecturile: vectoriale, superscalare cu extensie spre set de instrucțiuni multimedia și procesoarele GPU.

MIMD, fiecarui PE i se aplică propriul strum de instrucțiuni care oprează asupra propriului strum de date, deci un paralelism la nivel de task. MIMD-ul este mai flexibil și mai general aplicabil decât SIMD-ul, dar, evident, mai scump. MIMD-ul poate realiza și procesări de tipul paralelismului la nivel de date, specific organizărilor de tip SIMD, dar necesită soft (de regie) suplimentar.

Acestă taxonomie, în principiu, are în primul rând o utilitate didactică, pentru că în practică sistemele cu procesoare multiple sunt o hibridizare între organizările SISD, SIMD și MIMD.

5.3.1. Calculatoare de tip SIMD

Organizarea SIMD tipică constă într-o singură unitate de control și în **elemente de procesare, PE** (Processing Unit/Element). Unitatea de control realizează fetch, decodificarea și transmiterea aceleiași instrucțiuni, pentru a fi executată, la toate cele n elemente de procesare, care sunt CPU-uri dar fără unitate de control, referite și ca **procesoare matriceale**. Sincronizarea execuției pe toate PE-urile se realizează prin transmisia aceleiași



instrucțiuni de către unitatea de control; fiecare PE având propriile sale registre și propria sa memorie de date (nu și de program!). Motivația inițială pentru organizarea de tip SIMD a fost reducerea costului unităților de control pentru toate elementele de procesare, de asemenea reducerea dimensiunii memoriei de program, deoarece este necesar un singur program pentru toate elementele de procesare. Calculatoarele de tip SIMD sunt potrivite pentru structurile de date identice, care pot fi procesate în paralel, cum ar fi bucle *for* în structurile matriceale. În tipul de organizare SIMD se încadrează foarte bine procesoarele vectoriale, aceeași instrucțiune operează identic asupra tuturor componentelor unui vector.

În clasa de SIMD pot fi încadrate și procesoarele superscalare, cărora li s-a exins ISA cu instrucțiuni pentru procesare multimedia. Prin procesare multimedia se înțelege *crearea, codificarea și decodificarea, procesarea, vizualizarea și transmiterea informației digitale multimedia pentru: audio, imagini, video, și grafică*. Pentru eșantioanele de semnal audio (codificate pe 8 sau 16 biți) sau pentru pixelii unei imagini (codificați pe 4 cuvinte cu lungimea de 8 biți pentru cele trei culori fundamentale RGB(Red, Green, Blue) plus transparență) rezultă streamuri data cu următoarele particularități :

1. fiecare cuvânt data din stream este limitat ca lungime (de exemplu 8 biți);
2. volum mare de astfel de date, iar asupra fiecărei cuvânt data se aplică aceeași operație.

Cuvintele data de tip multimedia pot fi împachetate într-un cuvânt mai lung, de exemplu, opt astfel de date de un byte se formează un cuvânt lung de 64 biți. Mai mult, unitățile funcționale din unitățile de procesare pot fi divizate pentru a realiza, în paralel, aceeași operație pentru fiecare cuvânt data din cuvântul lung, de exemplu o unitate aritmetică ce realizează adunare pentru două cuvinte de 64 de biți este partajată pentru a efectua în paralel 8 adunării a opt perechi de operanzi de un byte. Pentru astfel de procesări multimedia, începând cu anii '90, ISA procesoarelor superscalare a fost extinsă, în acest scop, cu seturi de instrucțiuni specifice (această extensie uneori cuprinde mai mult de o sută de instrucțiuni !). De exemplu, pentru arhitectura x86 începând cu Pentium II s-au introdus 57 instrucțiuni referite MMX (Multimedia Extension), apoi încă 13 instrucțiuni pentru Pentium III, referite SSE (Streaming SIMD Extension), iar la Pentium 4 s-a ajuns, în total, la 144 instrucțiuni referite SSE2, dar în 2004 s-a mai adăugat încă 13 instrucțiuni, referite acum, toate instrucțiunile extinse, prin abreviația SSE3. În 2010 s-a introdus extensia multimedia Advance Vector Extention, AVX, care lucrează cu cuvinte multimedia împachetate în lungimi de 256 biți ; AVX include posibilitatea, pentru viitor, la lungimi de 512 și 1024 biți.

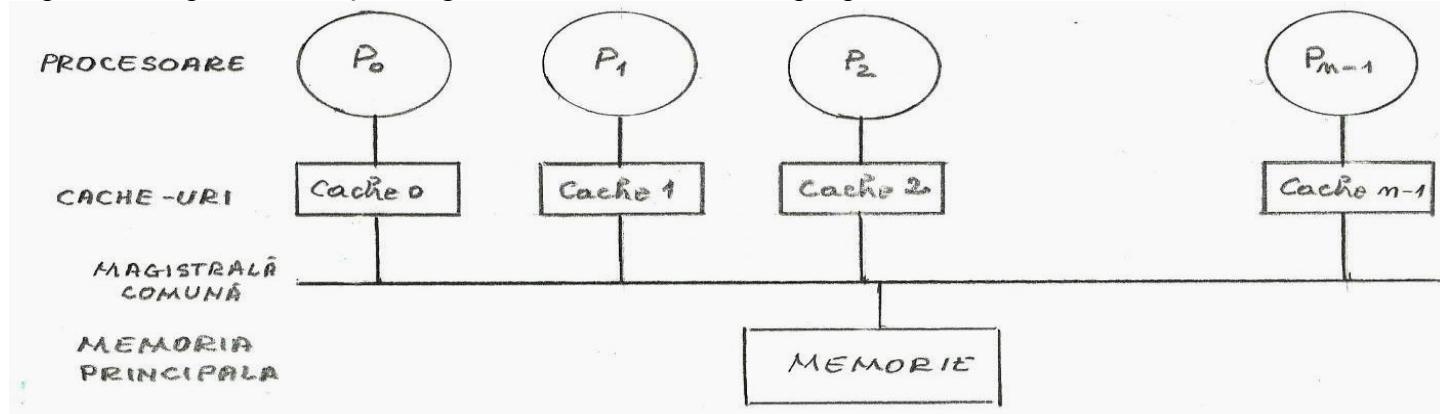
De asemenea, și procesoarele GPU pot fi considerate ca fiind o variantă de SIMD

5.3.2. Calculatoare de tip MIMD

Organizarea de tip MIMD este cea mai generală, mai multe procesoare, fiecare cu propriu cache, operează în paralel în mod asincron. O distincție între organizările de tip MIMD se poate face în funcție de accesul procesoarelor la memorie și de modul de comunicare între ele:

1. cu un spațiu de adresare comun (shared-memory) tuturor procesoarelor și comunicare prin instrucțiuni load-store (SMP);
2. cu spații de adresare independente și comunicare prin transfer de mesaje (Clustere)

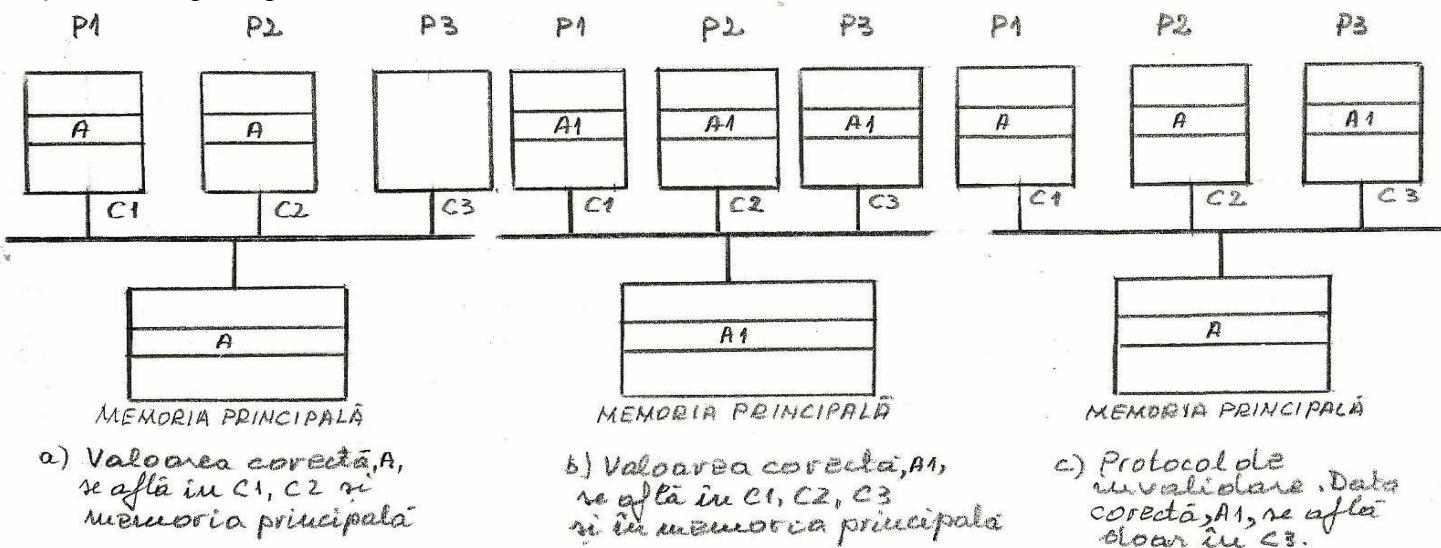
1. **Organizarea de tip SMP** (Shared Memory multiProcessors), cu o reprezentare de principiu ca în figura următoare, prezintă un singur **spațiu (comun) de adresare**, dar care este partajat pentru fiecare procesor. Toate cele n procesoare plus memoria principală sunt conectate împreună prin intermediul unei magistrale, în unele implementări poate exista și o magistrală de control, de exemplu pentru sincronizare.



Uzual, toate procesoarele sunt identice, dar pot fi și diferite dacă au aceeași ISA. Schimbul de date între procesoare este realizat prin intermediul memoriei principale cu instrucțiunile de tip load-store. Deoarece timpul de acces al tuturor procesoarelor la memoria principală partajată este același (uniform), această organizare mai este referită și prin abreviația **UMA** (Uniform Memory Access).

Organizarea de tip UMA, inclusiv un număr nu prea mare de procesoare identice, a constituit prima structură pe baza cără s-a implementat primele microprocesoare multicore.

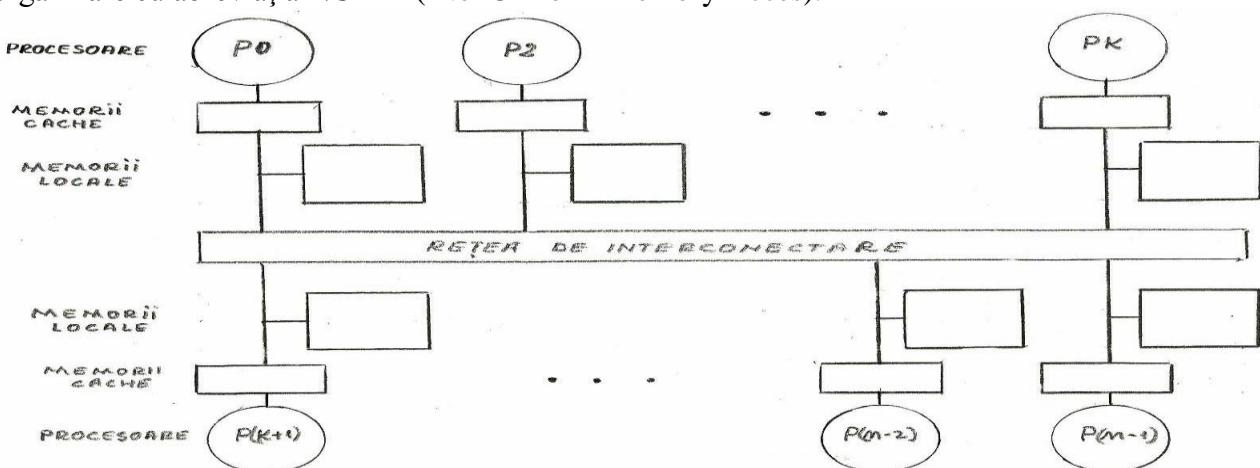
Deoarece memoria principală este comună pentru toate procesoarele, la un moment dat, poate apărea inconsistență datelor între diferențele cache-uri, ceea ce este referit prin **coerență cache**. Pentru explicarea apariției incoerenței între datele dintre diferențele cache-uri se va considera, în figura următoare, o structură UMA din trei procesoare, în care s-a reprezentat doar cele trei memorii cache (C1, C2, C3) corespunzătoare procesoarelor P1, P2, P3 și memoria principală.



Inițial toate cele trei cache-uri, C1, C2, C3, sunt goale. Apoi procesorul P1 accesează variabila A (de valoare A), care nefiind în cache (miss) este adusă și înscrisă în cache C1. De asemenea procesorul P2 citește variabila A, dar fiind miss în cache C2 va citi această valoare, fie din memorie, fie din cache C1, depinde de protocolul de coerență, și o va înscrie în C2. În acest moment C1, C2 și memoria principală au câte o copie corectă a valorii variabilei A, Figura a). Apoi, procesorul P3 înscrise în C3 pentru variabila A valoarea nou produsă A1, dar deoarece această variabilă nu a fost adusă încă în cache apare evenimentul de miss la înscrisere, pentru care se pot genera următoarele două cazuri. Primul caz, procesorul pe lângă înscriserea valorii A1 în C3 va trimite valoarea A1 la C1, C2 și memoria principală, deci toate cache-urile și memoria principală au o copie validă pentru variabila A, Figura b); acest protocol de up-datare este similar cu tehnica de înscrisere write-through de la un monoprocesor cu memoria cache. Al doilea caz, procesorul P3 printr-un protocol de invalidare înscrisere trimite un mesaj de invalidare la C1, C2 și memoria principală, deci în acest moment copia validă A1, pentru valoarea variabilei A, se află doar în C3, ceea ce este similar cu tehnica writeback de la un monoprocesor cu memoria cache. Iată că, în acest caz în sistem există două valori pentru variabila A, valoarea A1 în C3 și valoarea A în C1, C2 și memoria principală, deci cache-urile nu sunt consistente! ceea ce este reprezentat în Figura c). Pentru astfel de sisteme multiprocesor care partajează o magistrală comună, pentru rezolvarea situațiile de incoerență cache, este implementat **snoopy protocol**. Pe baza acestui protocol, toate mesajele transmise pe magistrala comună partajată sunt interceptate și ascultate de către controlerile cache ale tuturor procesoarelor, și, în consecință, se acționează după cum în cache-ul respectiv există sau nu o copie corectă a valorii datei care a fost procesată.

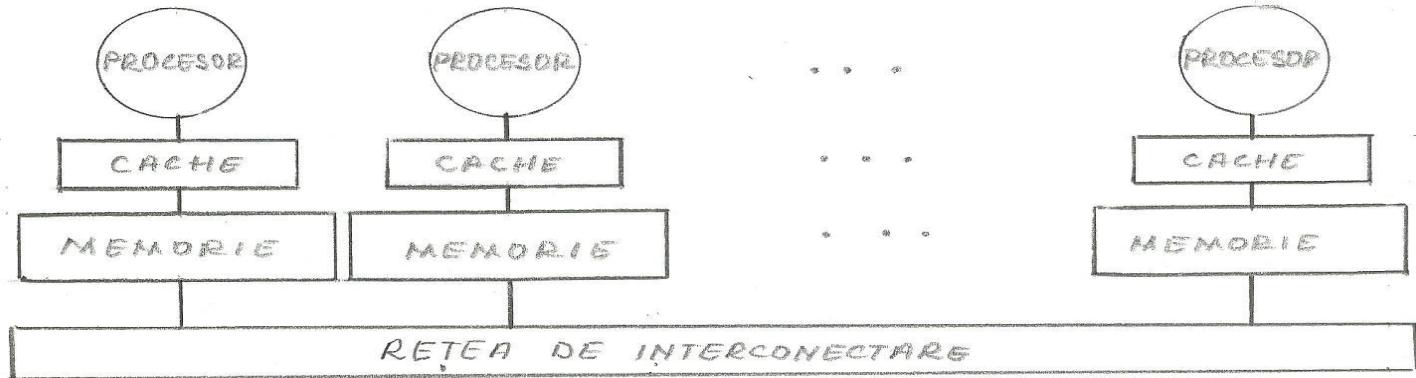
Utilizarea partajată a unei singure magistrale comune pentru organizările multiprocesor de tip UMA prezintă următoarele avantaje: simplitate, cost redus, ușurință în utilizare, precum și posibilitatea implementării unui protocol simplu de coerență cache – snoopy protocol. Dar pe de altă parte, magistrala comună partajată are și următoarele limitări: restricționări de natură electrică (lungime, încărcare), număr redus de dispozitive care se pot conecta, scalabilitatea redusă și posibilitatea apariției de strangulări datorită concurenței dispozitivelor conectate; toate acestea au ca și consecință directă reducerea vitezei de transfer. În consecință, pentru eliminarea acestor limitări se substitue comunicare pe magistrală cu o comunicare pe o rețea de interconectare.

Organizare pe baza unei rețele de interconectare. Pentru unele organizări de multiprocesoare memoria principală (comună) existentă la organizarea de tip UMA este repartizată, ca memorii locale, la fiecare din procesoarele din sistem, iar conectarea se realizează printr-o rețea de interconectare, cum este reprezentat în figura următoare. Toate memorile locale sunt parteal aceluiași spațiu de adresare! În această organizare, cu memorii locale, dacă la execuția unei instrucțiunii de tip load-store apare o situație de miss la propriul cache, atunci primul acces se realizează la memoria locală a procesorului respectiv, dar dacă și acest acces este miss se continuă cu acces, prin rețeaua de interconectare, la memoriile de la celelalte procesoare. Evident că în funcție de calea de acces pentru un procesor, la locul unde se află sursa (în cache-ul sau memoria locală proprie sau în memorile locale de la celelalte procesoare), depind și timpii de acces, care nu sunt egali, de unde și denumirea pentru această organizare cu abreviația **NUMA** (NonUniform Memory Access).



2. Organizare de tip message passing. Alternativă la organizarea de tip SMP este cea în care fiecare procesor are propriul său spațiu de adresare, o organizare clasică pentru un astfel de multiprocesor este prezentată în figura următoare. Procesoarele comunică între ele, nu printr-un spațiu de memorie comun/partajat, ci prin trimiterea și primirea de mesaje, de unde și denumirea de *organizare de tip message passing*. Fiecare procesor component are

rutină de trimis mesaje și rutină de recepționat mesaje. Dacă procesorul emițător necesită confirmarea că un mesaj a ajuns la destinație, procesorul receptor trimite o confirmare la procesorul emițător. Programarea pentru acest tip de organizare ridică anumite dificultăți, deoarece orice comunicare în program, de tip message passing, trebuie identificată în avans de programator. Astfel de organizări sunt recomandate pentru procesarea paralelă de programe independente (job-level parallelism) sau aplicații cu mic transfer de comunicație între procesoare, cum ar fi: căutare Web, servere mail sau servere de fișiere.



O variantă foarte curentă, de organizare de tip message passing, o constituie **clusterele**. Un cluster este, în general, o colecție de calculatoare obținabile comercial, conectate între ele prin I/O pe baza unei rețele locale (LAN) standard de switch-uri și cabluri, care pentru sarcini ce nu necesită comunicații intensive între calculatoare oferă un raport cost/performanță mult mai bun decât organizările cu memorie partajată

Organizarea de tip cluster în raport cu organizarea de tip SMP prezintă următoarele dezavantaje:

- costul administrării unui cluster compus din n mașini este aproximativ același cu cel al administrării a n mașini independente, pe cind costul de administrare unei organizări cu memorie partajată de n procesoare este aproximativ egal cu cel al administrării unei singure mașini;
- deoarece conectarea calculatoarelor într-un cluster se face prin I/O al fiecărui calculator, pe baza unei rețele standard, pe când procesoarele într-un SMP sunt legate printr-o rețea de interconectare, rezultă pentru clustere o mai redusă performanță de comunicare;
- deoarece un cluster este compus din n mașini independente necesită n memorii și n copii de sisteme de operare, pe când o organizare SMP necesită doar o singură memorie partajată și o singură copie de sistem de operare.

În schimb:

- pentru un cluster, care este format din mașini independente conectate într-o rețea locală, este mai ușor ca o mașină să fie izolată decât un procesor în SMP;
- este mult mai ușor de realizat extensia unui cluster decât a unui sistem SMP.

Din organizările de tip cluster, foarte populare în anii '90 au rezultat: centrele de date, supercalculatoarele și WSC [1].

Centre de date (datacenter) sunt o colecție (cluster) de mașini și produse software, centralizată, pentru necesitățile IT ale unei firme. Prin această centralizare, de suport hard și soft, se asigură o eficiență și protecție pentru nevoie IT.

Supercalculatoare, HPC (High Performance Computers), sunt un cluster de milioane de procesoare de viteză ridicată, conectate printr-o rețea rapidă de comunicare între noduri, care realizează aplicații de calcul cu o largă comunicație între noduri și la o frecvență foarte ridicată. Uzual, utilizarea nu este de programe în paralel ci de programe științifice lungi și independente care pot ocupa intervale chiar și săptămâni, fără un schimb cu rețelele din exterior. Astfel de HPC costă de ordinul sute de milioane de dolari.(vazi Clase (de aplicații) pentru calculatoare în cursul introductiv).

WSC (Warehouses Scale Computers), sunt componente fundamentale pentru serviciul Internet (search, online maps, online shopping, email services etc) și Cloud Computing (Software as a Service- SaaS). Un WSC trebuie privit ca un singur (cluster) calculator compus dintr-un număr mare (50 000- 100 000) de servere concentrate într-o singură hală cu o puternică rețea electrică de alimentare (zeci-sute de MW) și o infrastructură de

răcire (costul pentru un WSC este peste \$150 000 000). S-ar putea considera un WSC ca un datacenter extins la o scară foarte mare, în principiu da, dar într-un WSC există o uniformitate a echipamentelor hardware (servere) chiar și a soft-ului pe când într-un datacenter există o neuniformitate a echipamentelor hardware și soft-ului utilizat. În imaginile următoare, pentru Un WSC Google sunt prezentate: 6.21- fotografia serverului; 6.19- secțiune prin containerul cu servere; 6.20- circulația aerului de răcire prin container; iar ultima reprezintă imaginea halei în care sunt plasate 45 containere.

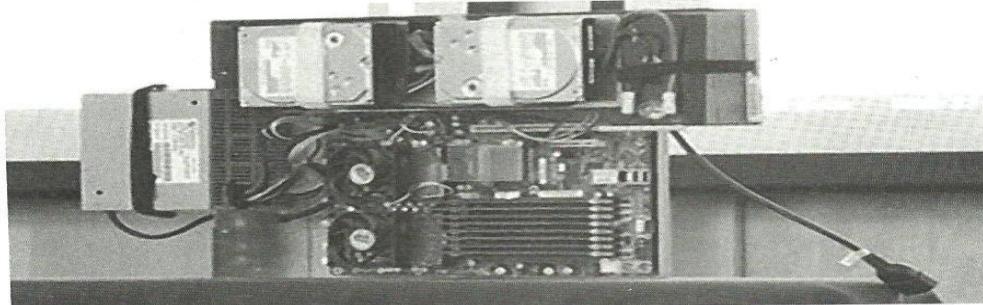


Figure 6.21 Server for Google WSC. The power supply is on the left and the two disks are on the top. The two fans below the left disk cover the two sockets of the AMD Barcelona microprocessor, each with two cores, running at 2.2 GHz. The eight DIMMs in the lower right each hold 1 GB, giving a total of 8 GB. There is no extra sheet metal, as the servers are plugged into the battery and a separate plenum is in the rack for each server to help control the airflow. In part because of the height of the batteries, 20 servers fit in a rack.

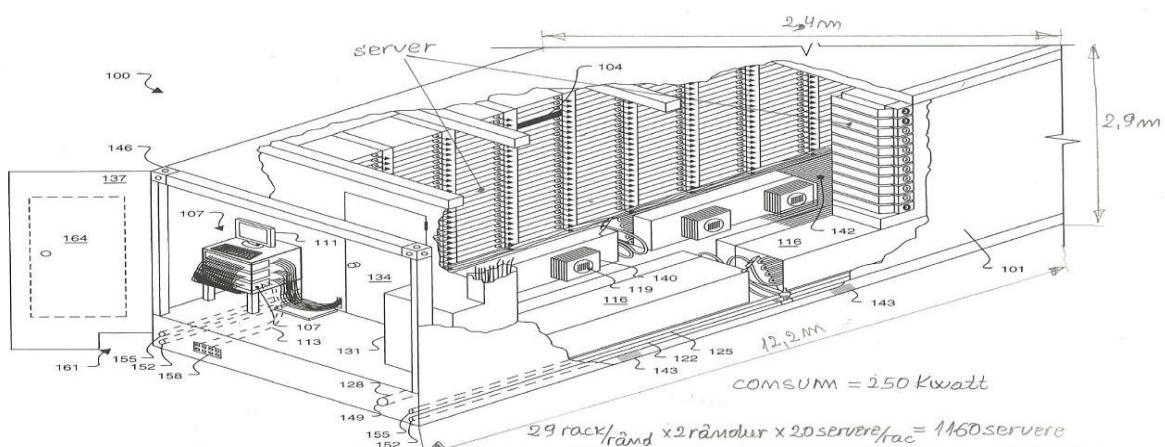


Figure 6.19 Google customizes a standard 1A AAA container: 40 x 8 x 9.5 feet (12.2 x 2.4 x 2.9 meters). The servers are stacked up to 20 high in racks that form two long rows of 29 racks each, with one row on each side of the container. The cool aisle goes down the middle of the container, with the hot air return being on the outside. The hanging rack structure makes it easier to repair the cooling system without removing the servers. To allow people inside the container to repair components, it contains safety systems for fire detection and mist-based suppression, emergency egress and lighting, and emergency power shut-off. Containers also have many sensors: temperature, airflow pressure, air leak detection, and motion-sensing lighting. A video tour of the datacenter can be found at <http://www.google.com/corporate/green/datacenters/summit.html>. Microsoft, Yahoo!, and many others are now building modular datacenters based upon these ideas but they have stopped using ISO standard containers since the size is inconvenient.

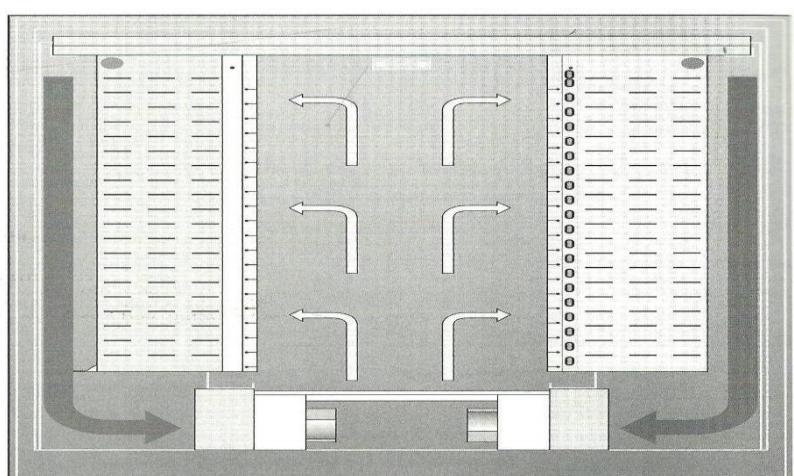


Figure 6.20 Airflow within the container shown in Figure 6.19. This cross-section diagram shows two racks on each side of the container. Cold air blows into the aisle in the middle of the container and is then sucked into the servers. Warm air returns at the edges of the container. This design isolates cold and warm airflows.



+

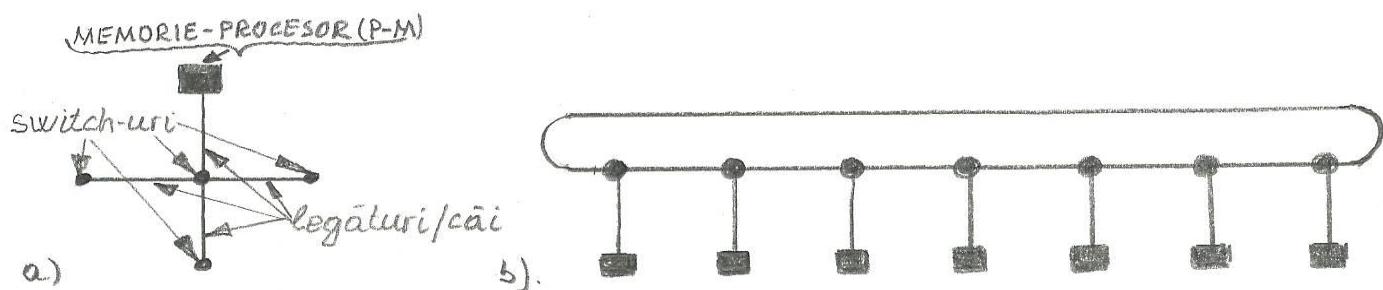
La adresa <http://research.google.com/pubs/HardwareandArchitecture.html> se poate vizualiza un film de prezentare a unui WSC Google.

Pentru Personal Mobile Device, PMD, organizările de tip SIMD par să fie mai atractive decât cele de tip MIMD, argumentele în favoarea SIMD sunt:

1. Extensia aplicațiilor actuale cu paralelism la nivel de date, DLP (Data-Level Parallelism) de la domeniul științific, puternic bazat pe calcul matricial, și la domeniul multimedia pentru procesare de sunet și imagine.
2. Din punct de vedere al eficienței energetice este mai avantajos lansarea unei singure instrucțiuni care efectueză aceeași operație asupra mai multor date decât aducerea și executarea a câte unei instrucțiuni pentru fiecare dată.
3. Și, poate cel mai important, programatorul continuă să gândescă secvențial, dar se obține creștere de viteză prin operații paralele asupra datelor.

5.3.2. Rețele de intreconectare multiprocessor

Conecțarea procesoarelor în organizarea de tip UMA, de regulă, este realizată pe o magistrală comună partajată, ceea ce impune ca la un moment dat doar un singur procesor să comunice pe magistrală. Pentru alte organizări și un număr ridicat de procesoare conejarea acestora se realizează prin intermediul unei rețele de interconectare. O rețea de intreconectare se reprezintă printr-un graf în care (în figurile următoare): un arc reprezintă o legătură/cale de comunicație, un nod (reprezentat ca un punct) este constituit dintr-un switch, iar percheia procesor – memorie, P-M, este reprezentată ca un pătrat plin. De la un switch (nod) legăturile sunt la perechea P-M și la alte switch-uri, cum este reprezentat în Figura a.



Performanța unei rețele de intreconectare se apreciază după:

- latență pentru transmiterea și receptia unui mesaj, când rețeaua nu este încărcată;
- throughput, numărul maxim de mesaje într-o perioadă dată de timp;
- întârzierea cauzată de concurență pe o anumită legătură;
- variația de performanță în funcție de tipul de mesaj transmis;
- toleranța la defecte (sistemul să poată opera în prezența defectării unor componente);
- puterea disipată pe rețea.

Costul implementării unei rețele pe chip este funcție de:

- numărul de switch-uri;
- numărul de căi conectate la un switch;
- lățimea unei legături (nr de biți);
- lungimea unei legături pe suprafața de siliciu.

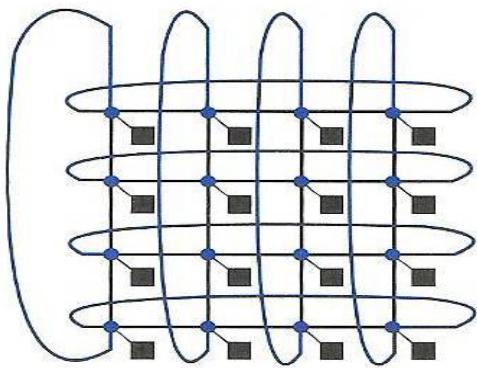
O simplă topologie de rețea de intreconectare este prezentată în Figura b (anterioară), care s-a obținut prin conectarea împreună a unei succesiuni de noduri, realizând **configurația inel (ring)**. Configurația ring ar părea similară cu o magistrală ale cărui capete au fost unite. În rețeaua ring spre deosebire de o magistrală, în care la un moment dat doar un singur procesor comunică pe magistrală, pot exista simultan mai multe transferuri.

Transferurile în ring nu totdeauna se realizează între două noduri vecine, mesajul ca să ajungă la nodul final trebuie să treacă prin mai multe noduri întâlnite pe traseu.

În extrema opusă față de simpla topologie de rețea ring se situează **rețeaua cu conectivitate totală**, care implementează legături bidirectionale de la fiecare procesor cu oricare din celelalte $(n-1)$ procesoare din sistem; numărul total de legături bidirectionale este egal cu $n(n-1)/2$.

Între costul scăzut oferit de configurația ring și performanțele obținute cu rețeaua cu conectivitate totală, există în literatura de specialitate foarte multe variante de rețele de conectare. Performanțele unei rețele de conectare depind de natura comunicațiilor necesare în procesarea paralelă a programelor pe sistemul multiprocesor respectiv. În continuare sunt prezentate următoarele variante de rețele de intreconectare:

1. rețeaua 2-D, cu 16 noduri;
2. rețeaua cub, cu dimensiunea $n = 3$;
3. rețea matrice (crossbar);
4. rețeaua omega.



a. 2-D grid or mesh of 16 nodes

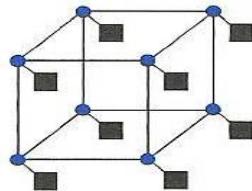
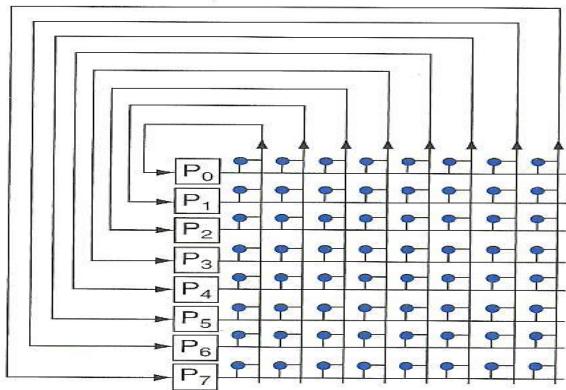
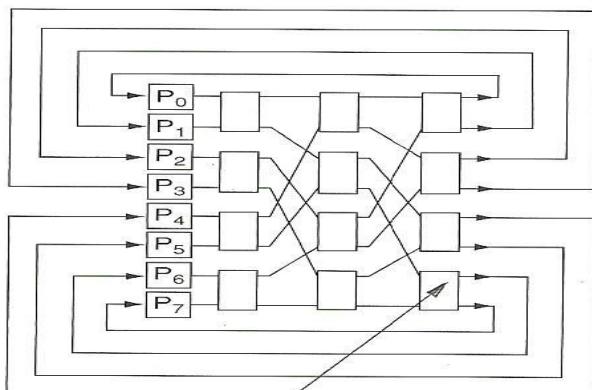
b. n-cube tree of 8 nodes ($8 = 2^3$ so $n = 3$)

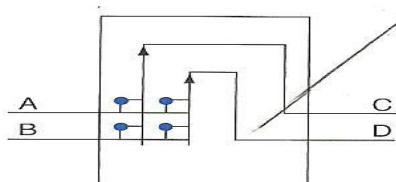
FIGURE 7.9 Network topologies that have appeared in commercial parallel processors. The colored circles represent switches and the black squares represent processor-memory nodes. Even though a switch has many links, generally only one goes to the processor. The Boolean n -cube topology is an n -dimensional interconnect with 2^n nodes, requiring n links per switch (plus one for the processor) and thus n nearest-neighbor nodes. Frequently, these basic topologies have been supplemented with extra arcs to improve performance and reliability.



a. Crossbar



b. Omega network



c. Omega network switch box

Comunicatia $P_0 \rightarrow P_6$
nu se poate realiza
simultan cu
comunicatia $P_1 \rightarrow P_7$

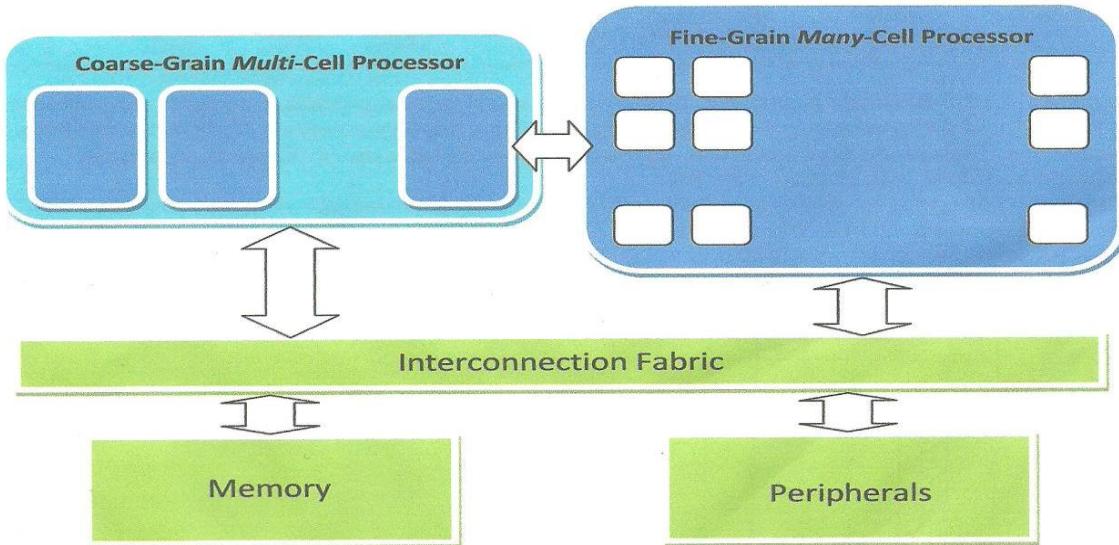
FIGURE 7.10 Popular multistage network topologies for eight nodes. The switches in these drawings are simpler than in earlier drawings because the links are unidirectional; data comes in at the bottom and exits out the right link. The switch box in c can pass A to C and B to D or B to C and A to D. The crossbar uses n^2 switches, where n is the number of processors, while the Omega network uses $2n \log_2 n$ of the large switch boxes, each of which is logically composed of four of the smaller switches. In this case, the crossbar uses 64 switches versus 12 switch boxes, or 48 switches, in the Omega network. The crossbar, however, can support any combination of messages between processors, while the Omega network cannot.

Încheiem subcapitolul de sisteme multiprocesor cu următoarele conceptualizări despre soft și hard. Hardware-ul poate funcționa serial (uniprocesor) sau paralel (multiprocesor), iar software-ul poate opera secvențial sau

concurrent. Cu aceste variantele de funcționare pentru hardware și software rezultă următoarele patru combinații exemplificate în tabelul următor

		Software	
		Secvențial	Concurrent
Hardware	Serial	Un program în Matlab, de înmulțire a două matrice, rulat pe un procesor clasic, de exemplu Intel Pentium 4	Sistem de operare (de exemplu Vista) rulat pe un procesor clasic, de exemplu Intel Pentium 4
	Paralel	Un program în Matlab, de înmulțire a două matrice, rulat pe un procesor multicore, de exemplu Intel Xeon e5345 (patru cores)	Sistem de operare (de exemplu Vista) rulat pe un procesor multicore, de exemplu Intel Xeon e5345 (patru cores)

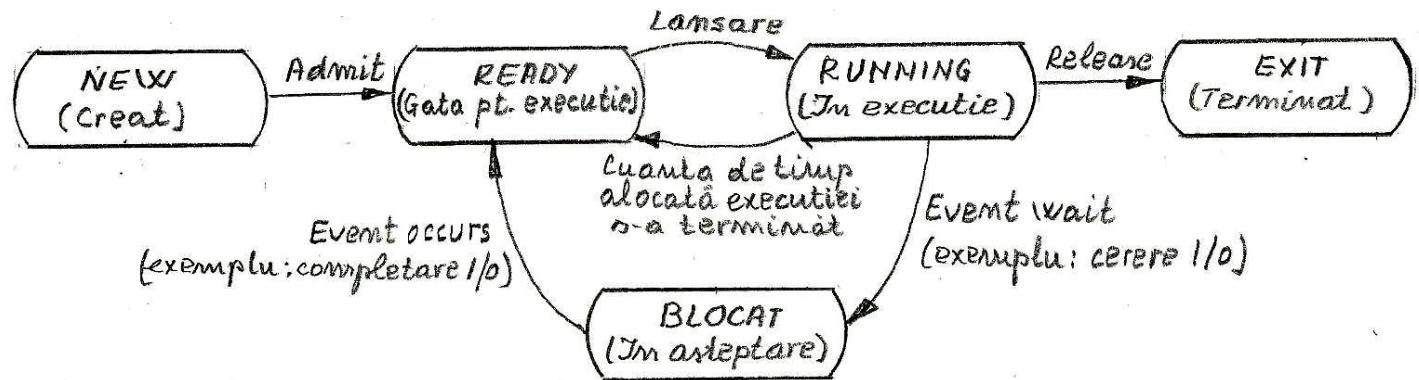
Cum va evoluă arhitectura și organizarea microprocesorului în următorii ani? Încă greu de spus, dar cu siguranță direcția în dezvoltarea sa va fi realizarea unui suport pentru procesarea paralelă. Ori, pentru această dezvoltare organizarea procesorului va fi, probabil, de **tipul multiprocesor/multicore, cluster sau multiprocesor de multiprocesoare** (veri procesoare grafice). Un mod de organizare pentru un astfel de viitor multiprocesor ar putea fi ca cel din figura următoare [9]. Sarcina de procesare în procesor este divizată în realizarea de calcule complexe (**complex computing**) și realizarea de calcul intensiv (**intense computing**). Suportul hardware pentru complex parallel computation este o rețea **Coarse-Grain Multi-Cell Processor**, scalabilă de 2-8 core (de exemplu superscalare); iar pentru calcule intensive o a doua rețea **Fine-Grain Many-Cell Processor**, scalabilă formată din 64-4096 elemente de procesare, PE.



5.4 PROCESAREA DE TIP MULTITHEAED

Proces. Definiție (cea mai frecventă) a noțiunii de proces (uneori referit **task**) este: *un program în execuție*. Execuția unui proces necesită alocarea unor resurse: cuantă de timp alocată pentru execuție (T_{CPU}), registre, memorie (spațiul alocat), fișiere, I/O etc; aceste resurse îi sunt alocate procesului de către sistemul de operare

atunci când procesul este creat; fiecare proces are propriile sale resurse, care nu se împart cu alt proces. Stările pe care le parcurge un proces, sub supravegherea sistemului de operare, sunt prezentate în organigrama următoare:

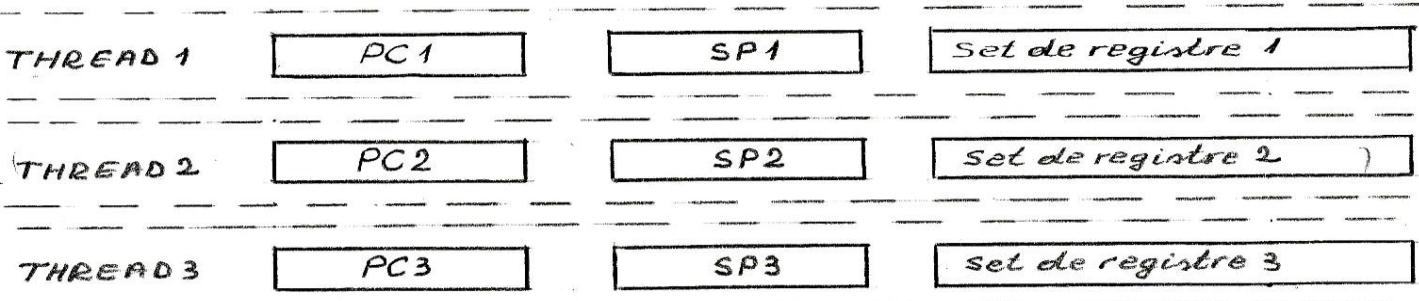


Un proces în starea de execuție (rulează pe CPU) poate fi substituit cu un alt proces, această substituire/comutare este referată prin sintagma **schimbarea contextului**. La schimbarea contextului este necesar salvarea celor resurse ale procesului prezent (conținutul regisrelor, registrul de stare, PC, SP, care sunt necesare la reluarea procesului), iar în CPU este introdus contextul noului proces. Procesele interacționază între ele numai explicit, dacă sistemul are un mecanism de comunicare între procese. Schimbarea contextului este o operație consumatoare de timp, se consumă de ordinul sutelor sau chiar mii de tacte.

Thread (fir, ață, a însira mărgele, engl.). *Un thread în execuție este cea mai mică unitate din program care poate fi procesată independent, care poate fi planificată de sistemul de operare.* Mai concret, succesiunea de instrucțiuni a unui program poate fi divizată în părți de succesiuni de instrucțiuni de dimensiuni mai mici, care pot fi executate independent una de alta, aceste părți sunt referite prin termenul de **thread (fir de execuție)**. Thread-ul rezultă în urma unei ramificații (fork) a programului în două sau mai multe taskuri concurente. Într-un proces pot exista multiple thread-uri, resursele procesului sunt împărtășite între thread-uri, pe când între diferite procese nu se împart resursele.

Programele în execuție pot fi divizate în thread-uri, uneori de ordinul sutelor, de exemplu în sarcinile unui server fiecare aplicație utilizator poate fi considerată ca fiind unul sau mai multe thread-uri. Recent procesarea de tip threading este introdusă și pentru domeniul embedded chiar și pentru smartphone (ca suport pentru rularea simultană a mai multor aplicații).

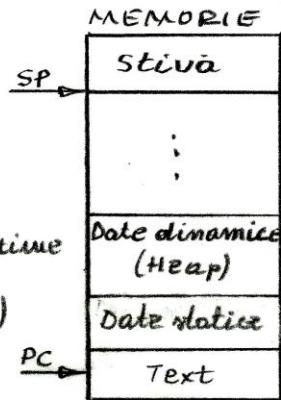
Logic, unui thread în cadrul procesorului i se alocă următoarele resurse: un PC, un SP și un set de registre proprii, cum este prezentat în Figura următoare, restul de resurse ale procesorului sunt disponibile/partajate pentru celelalte thread-uri ale procesului; memoria procesului, utilizată de un thread, poate fi ușor partajată prin mecanismul de memorie virtuală, care suportă multiprogramare.



COMPARATIE PROCES- THREADPROCES

- Resurse pentru execuția procesului:

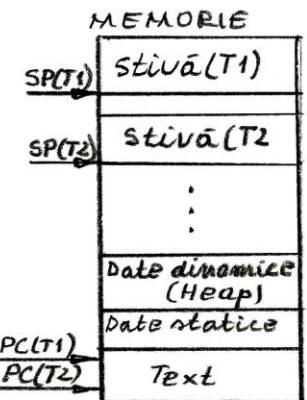
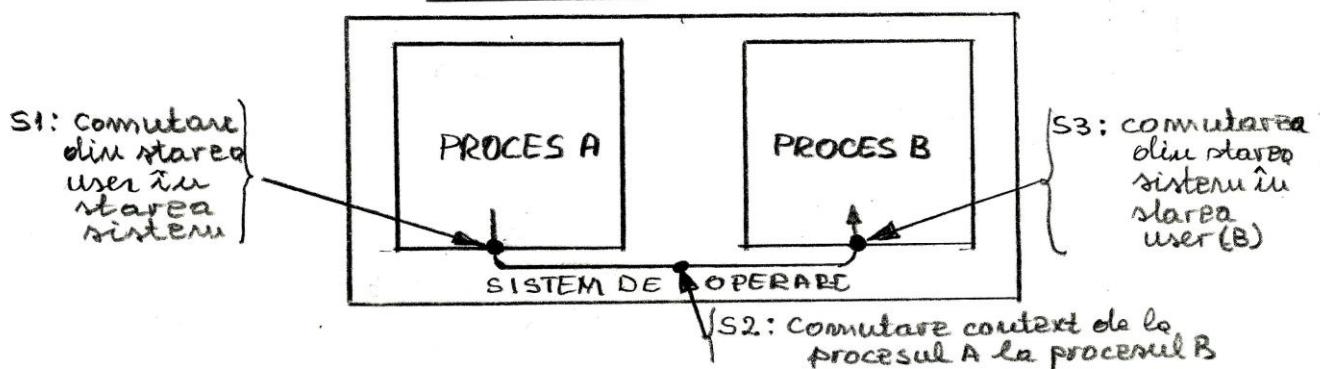
- Program Counter, PC
- Stack Pointer, SP
- Registre de date
- Timp de execuție, CPU time
- Memorie:
 1. Cod (text/program)
 2. Date
 3. Stivă
- I/O

THREAD

- Resurse pentru execuția thread-ului:

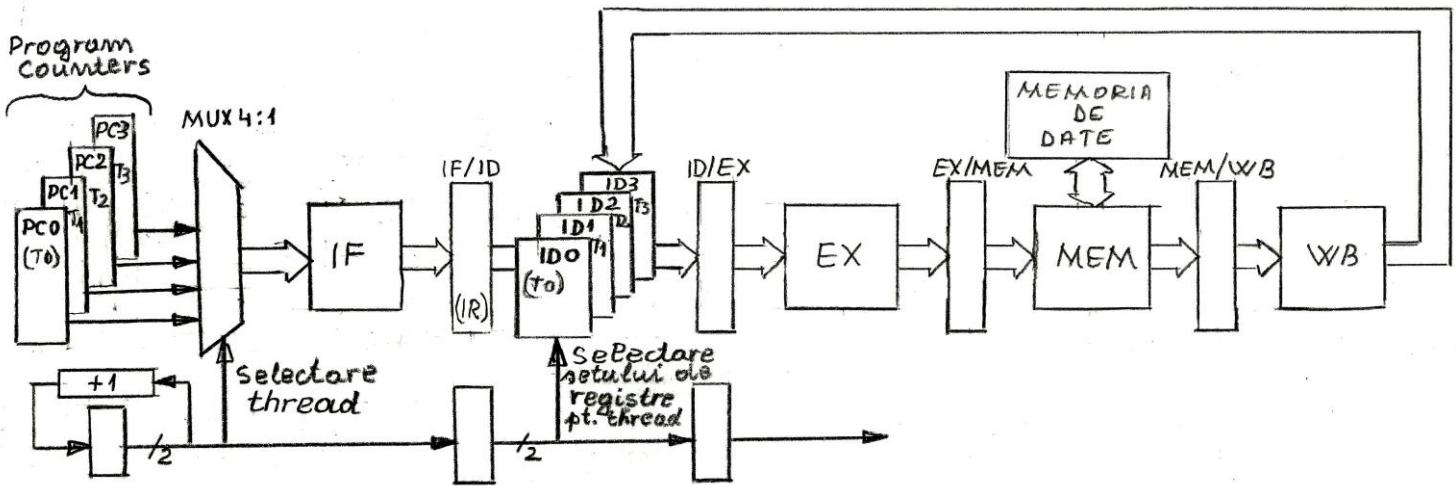
- Program counter, PC
- Stack pointer, SP
- Registre de date

T1 - Thread 1
T2 - Thread 2
...

COMUTAREA PROCESELOR

- Comunicare:
 - Comunicarea intre procese este costisitoare (necessită comutare de contexte);
 - Comunicarea intre thread-uri este ieftină (utilizează memoria comună alocată procesului)
- Securitate:
 - Un proces nu poate corupe un alt proces;
 - Un thread poate înscrie (corupe) memoria utilizată de către un alt thread.
- Comparatie tread-proces:
 - Un proces este o unitate de alocare, necesită: resurse, privilegii etc.
 - Un thread este o unitate de execuție, necesită: PC, SP și registre;
 - Fiecare proces are unul sau mai multe thread-uri;
 - Fiecare thread aparține unui proces.

Modul de principiu de transformare/completare a unui pipeline de la un procesor clasic, încât să poată procesa thread-uri, este schițat în figura următoare. Se observă posibilitatea de a se selecta, în etapa Fetch, thread-ul necesar (PC_i , $i = 0, 1, 2, 3$) prin intermediul unui MUX 4:1 și de asemenea, în etapa ID, selectarea setului de registre alocat thread-ului respectiv (ID_i , $i = 0, 1, 2, 3$).



Paralelismul la nivel de instrucție, ILP, într-un program este limitat, de aceea pentru procesarea în pipeline nu se pot găsi întotdeauna instrucții successive independente, ceea ce impune introducerea de etape goale (stall) în fluxul procesării, cum este reprezentat în Figura a) următoare. Soluția? să se exploateze un paralelism de

- un singur thread (T_1) în pipeline

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}
T_1	$lw \ $r1, 0($r2)$													
T_1														
T_1	$lw \ $r5, 12($r1)$													
T_1														
T_1	$add \ $r5, $r5, 12$													
T_1														
T_1	$sw \ $r5, 12($r1)$													
a)														

- Patru thread-uri (T_1, T_2, T_3, T_4) întrepătrunse în pipeline.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
Thread T_1	$lw \ $r1, 0($r2)$								
Thread T_2									
Thread T_3	$add \ $r7, $r1, $r4$								
Thread T_4									
Thread T_1	$xori \ $r5, $r4, 12$								
Thread T_2									
Thread T_3									
Thread T_4	$sw \ $r5, 0($r7)$								
Thread T_1	$lw \ $r5, 12($r1)$								
b).									

sunt eliminate etapele de stall!

granulație mai grosieră (coarse-grained parallelism) adică un **paralelism la nivel de thread, TLP** (Tread-Level Parallelism). Aceasta înseamnă execuție concurentă de instrucții de la diferite thread-uri, fie întrețesute în același pipeline, fie în paralel pe pipeline-uri diferite. Controlul procesorului comută de la un thread la altul, comutare care se poate realiza în mult mai puține tacte de ceas, chiar zero, decât comutarea proceselor, deoarece nu este nevoie de schimbare de context. De exemplu, în Figură a) anterioară, procesarea segmentului de instrucții din thread-ul T_1 impune introducerea de șase stall-uri în pipe, dar aceste stall-uri sunt eliminate dacă controlul procesorului selectează pe fiecare tact o instrucție de la unul din cele patru thread-uri (T_1, T_2, T_3, T_4) ale programului, cum este reprezentat în Figura b) anterioară, în acest exemplu comutarea de la thread la thread se face cu zero tacte consumate.

Modalități de comutarea thread-urilor. Există patru modalități/tehnici de comutare a thread-urilor implementabile pe procesoarele cu execuție multithread.

1. *Multithreading întrepătruns* (intreleaved multithreading), referit și prin termenul de multithreading de granulație fină (fine-grained multithreading). Pentru realizarea unei întrepătrunderi este necesar să existe, evident, cel puțin două thread-uri, atunci pe fiecare tact de ceas procesorul poate execuția o instrucție de la un alt thread; uzual, thread-urile se comută printr-o parcursare circulară (round – robin), de exemplu $T_1, T_2,$

T3, T4, T1, T2,...pentru patru thread-uri, evident, sărind acel thread care este în stall în momentul când este rândul său. Pentru a se realiza o întrepătrundere de granulație fină procesorul trebuie să poată comuta la alt thread la fiecare tact de ceas. Dezavantajul multithreadingului întrepătruns constă în prelungirea executării individuale a unui thread, pentru că executarea unui thread, care deși este disponibil de a fi executat fără etape de stall, este este prelungită cu timpul de executarea de instrucțiuni din celelalte thread-uri.

Această tehnică este eficientă la procesoarele scalare pipelinizate și la cele de tip VLIW.

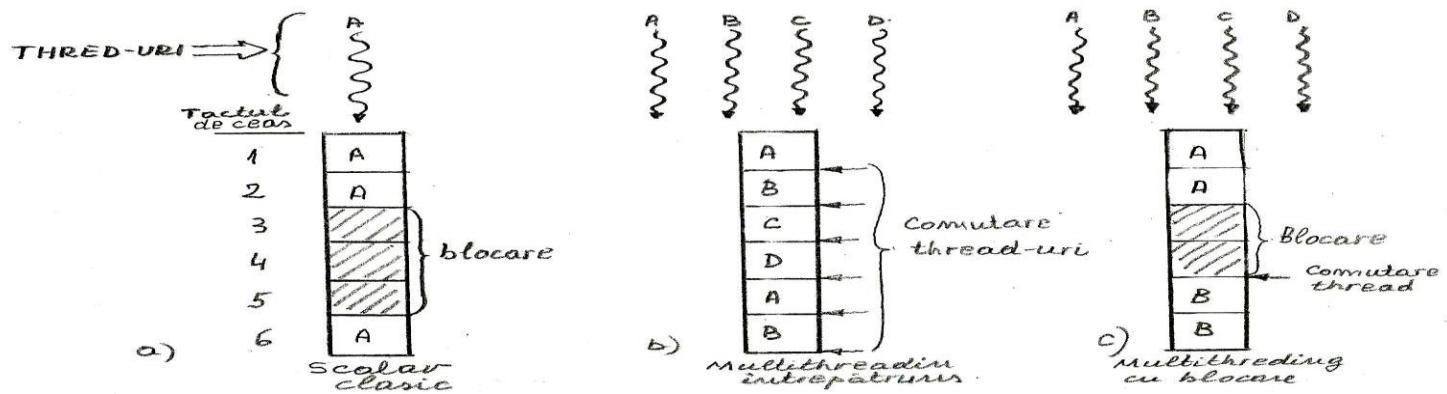
2. *Multithreading cu blocare* (blocked multithreading), referit și prin termenul de multithreading de granulație grosieră (coarse-grained multithreading). Spre deosebire de granulația fină unde există comutăție între threaduri de la tact la tact, la granulația grosieră se execută succesiv instrucțiunile dintr-un singur thread până când în thread-ul respectiv apare un eveniment care ar introduce foarte multe stall-uri (lungi), cum ar fi pagină lipsă ce necesită 10-20 de tacte dacă se face un acces la L2 cache sau chiar sute de tacte dacă accesul se continuă spre memoria principială. Numai la apariția unui eveniment cu stall-uri lungi acel thread este blocat și procesorul comută la un alt thread, la un eveniment doar cu puține stall-uri (scurte), de exemplu RAW, procesarea rămâne în cadrul thread-ului respectiv. Dezavantajul principal al multithreadingului cu blocare constă în micșorarea vitezei de procesare datorată evenimentelor cu stall-uri scurte din cadrul thread-ului. Nu se comută la un alt thread la fiecare stall scurt apărut, deoarece la comutare pipe-ul trebuie golit și apoi umplut cu instrucțiunile doar din nouă thread, ceea ce necesită un timp de umplere a pipe-ului (latență pipe) destul de lung, timp mult mai lung decât se consumă cu stallurile scurte din thread – ul care are aceste stall-uri. Această tehnică este eficientă la procesoarele scalare pipelinizate și la cele de tip VLIW.
3. *Multithreadingul simultan, SMT* (Simultaneous Multithreading). Acest mod de comutare presupune existența a mai multor unități de execuție, adică un procesor cu execuții multiple (superscalar). Pot fi lansate în execuție, în același tact, mai multe threaduri, Thread-Level Parallelism, cât și mai multe instrucțiuni din același thread, Instruction Level Parallelism.
4. *Multithreading pe chip multiprocesor* (Chip Multiprocessor Threading). Fiecare core, al unui procesor multicore, îl se repartizează către un thread.

Procesarea de tip multithread. Modalitățile de comutare a thread-urilor prezentate anterior se vor exemplifica pe următoarele organizări de microprocesoare studiate:

1. *Procesoare scalare pipelinizate*, Figura următoare. În Figura a) este prezentat un procesor scalar pipelinizat clasic, pe care rulează doar un program (thread-ul A); după două instrucțiuni introduse în pipe, pe tactul 3, apare o situație de hazard care se rezolvă cu trei etape de stall, tactele 3, 4, 5, după care se reia procesare tot din thread-ul A pe tactul 6.

Figura b) corespunde unei mașini scalare pipelinizată multithreading, pe care se procesează patru thread-uri (A, B, C, D) în modul întrepătruns (multithreading cu granulație fină), care, evident, trebuie să aibă, față de o mașină scalară pipelinizată clasică, patru PC și patru SP precum și patru seturi de registre, pentru fiecare thread un set de registre. Pe fiecare tact se introduce, în mod circular A, B, C, D. A, B,.... de la cele patru thread-uri, câte o instrucțiune; dacă unul din thread-uri este în starea de stall, atunci se introduce în pipe pentru acel thread o etapă goală. Dacă hardul este capabil să comute pe fiecare tact, și thread-urile nu prezintă etape de stall, atunci se procesează cu pipe-ul full.

Procesarea de tip multithreading cu blocare (multithreading cu granulație grosieră) este prezentat în Figura c). Se introduc în pipe instrucțiuni doar de la un singur thread, în acest caz de la thread-ul A, pe tactele 1 și 2 (după care acest thread trebuie blocat), apoi pe tactul 5 se comută la instrucțiuni numai de la thread-ul B. Deoarece pipe-ul trebuie întâi golit de thread-ul A și apoi umplut cu instrucțiunile de la thread-ul B tactele de neprocesare pot fi destul de numeroase (în desen s-au considerat numai două, tactele 3 și 4, în realitate pot fi mult mai multe tacte). Comutarea de la thread-ul A la B s-a realizat pentru că thread-ul A s-a blocat deoarece pe tactul 3 a întâmpinat, probabil, un eveniment de page fault (acces la L2 cache).



2. Procesoare cu execuții multiple. În figura următoare sunt prezentate variante de procesare multithread pe procesoarele superscalare și pe procesoarele de tip VLIW. Pentru ambele tipuri de procesoare se consideră că prezintă patru unități funcționale, deci se pot lansa în paralel, doar de la același thread, maximum patru instrucțiuni, evident, numai dacă aceste instrucțiuni sunt independente (parallelism la nivel de instrucțiune, ILP). În reprezentările dreptunghiulare, pe orizontală, sunt reprezentate sloturile mașinii (unități funcționale), adică numărul maxim de instrucțiuni care, în același tact, pot fi lansate în execuție paralelă (în acest caz Bandwidth = 4 instrucțiuni/tact); sloturile care nu pot fi acoperite cu instrucțiuni în același tact sunt referite ca pierderi orizontale. Pe verticală este reprezentată succesiunea de acoperire a sloturilor în fiecare tact aplicat; pot există tacte în care nu se acoperă niciun slot, nu se lansează în execuție nicio instrucțiune din thread, din cele patru posibile, acestea sunt referite ca pierderi verticale.

În Figura a) este prezentată mașina superscalară, fără multithreading, se observă că apar atât pierderi orizontale (nu se găsesc în același tact suficiente instrucțiuni din thread-ul A ca să acopere cele patru sloturi), dar apar și pierderi verticale, adică sunt tacte în care nu se poate lansa în execuție nici o instrucțiune (de exemplu pe tactele 3 și 5). În Figura a'), similar cu Figura a), este prezentată mașina VLIW, fără multithreading, se observă că apar atât pierderi orizontale cât și verticale, dar spre deosebire de mașina superscalară (a), de data aceasta, pierderile pe orizontală sunt introduse prin instrucțiuni NOP, acestea fiind împachetate în cuvântul lung de patru instrucțiuni.

Figura b) și b'), fiind patru threaduri A, B, C, D, aplicând modalitatea de comutare prin multithreading intrepătruns, atât la mașina superscalară cât și la mașina VLIW, în fiecare tact sunt introduse în sloturi cât se poate de multe instrucțiuni independente (până la Bandwidth = 4), din thread-ul respectiv. Procesarea prin aplicarea multithreading cu blocare este prezentată în Figura c) și c'). În fiecare tact sunt lansate în execuție până la patru instrucțiuni independente, dacă se găsesc în thread, operație care se continuă și pe tactele următoare cu lansare de instrucțiuni din același thread până când apare un eveniment care induce o latență (de exemplu, page fault, acces la L2 cache). Evenimentul care induce o latență generează o blocare a thread-ului curent și o comutare la

un alt thread, comutare care poate fi cu sau fără pierdei pe verticală.

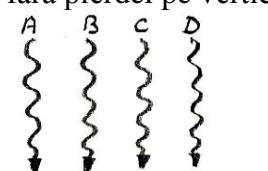
SUPERSCALAR

Tactul de ceas

1	A	A		
2	A			
3				
4	A	A	A	A
5				
6	A	A		A

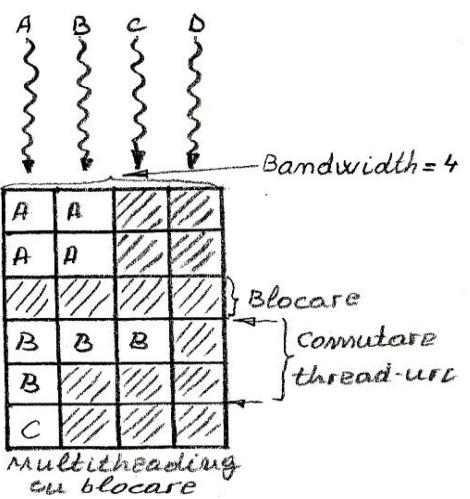
a)

Superscalar clasice



A	A		
B	B	B	D
C			
D	D	D	D
A	A		
B			

b) Multithreading întrepătruns



c)

A	A		
A	A		
B	B	B	
B	B	B	
B	B	B	
C			

Multithreading cu blocare

VLIW

Tactul de ceas

1	A	A	NOP	NOP
2	A	NOP	NOP	NOP
3				
4	A	A	A	A
5				
6	A	A	A	NOP

a')

VLIW

A	A	NOP	NOP
B	B	B	NOP
C	NOP	NOP	NOP
D	D	D	D
A	A	NOP	NOP
B	NOP	NOP	NOP

b) Multithreading întrepătruns

A	A	NOP	NOP
A	A	NOP	NOP
B	B	B	
B	NOP	NOP	NOP
C	NOP	NOP	NOP

c')

Multithreading cu blocare

3. Procesoare cu multithreading simultan. În acest caz se lansează simultan în execuție un număr de instrucțiuni egal cu patru (=Bandwidth), dacă se găsesc astfel de instrucțiuni independente. Spre deosebire de celelalte două tipuri de procesări anterioare, de data aceasta instrucțiunile lansate pot fi selectate nu numai dintr-un singur thread ci simultan din toate thread-urile procesului, deci se exploatează atât paralelismul la nivel de instrucțiune, ILP (din același thread) cât și paralelismul la nivel de thread, TLP (din thread-uri diferite).

Procesorul din Figura a) următoare este un superscalar cu Bandwidth = 8 (unități funcționale, sloturi), deci se pot lansa simultan în execuție până la opt instrucțiuni/tact selectate din cele patru thread-uri ale procesului. Cele opt sloturi pot fi acoperite cu instrucțiuni de la un singur thread, dacă acesta prezintă un ILP ridicat, sau se pot acoperi cu instrucțiuni selectate din mai multe thread-uri.

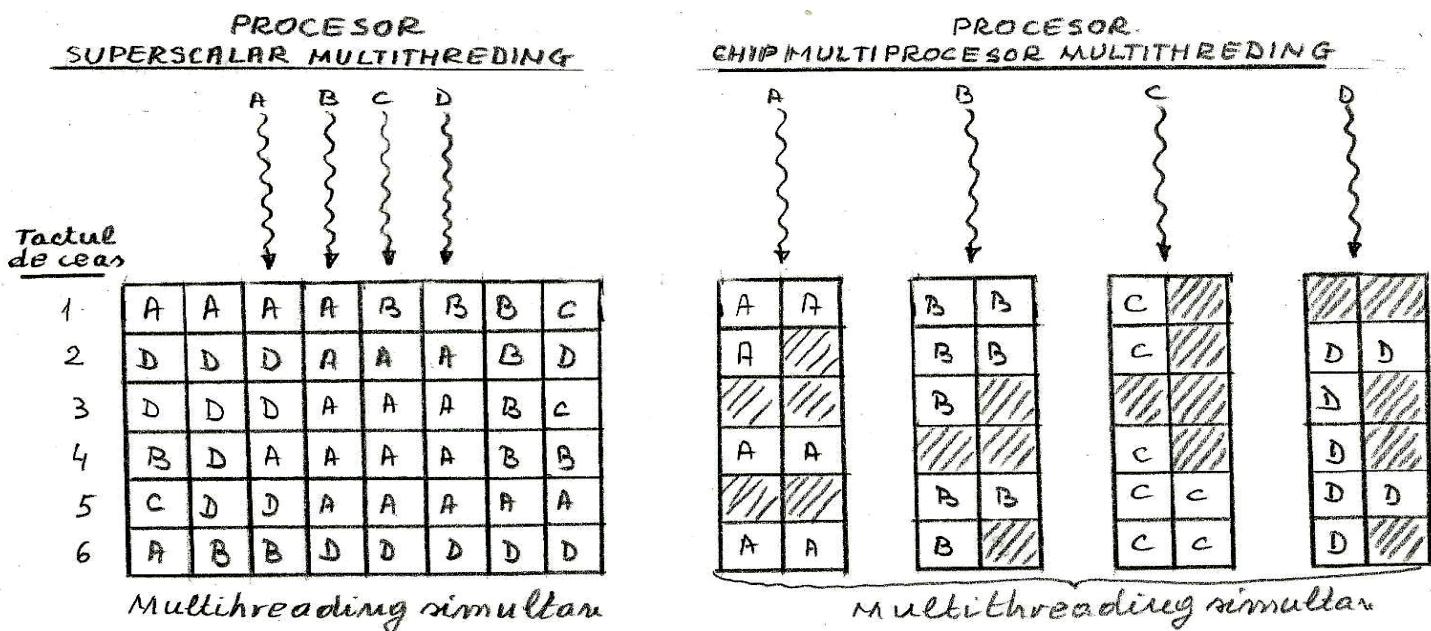
Procesorul multicore din Figura b), are patru procesoare/core identice, fiecare core fiind un 2-way superscalar (rezintă două unități funcționale), iar fiecărui core i se repartizează câte un thread. Este mult mai ușor să se acopere cu instrucțiuni independente de la un thread sloturile unui core 2-way superscalar decât pentru unul 8-way superscalar; rezultă că pierderile pe orizontală sunt mai mici la un procesor 4-core, fiecare core 2-way superscalar, decât la un 8-way superscalar.

La procesoarele superscalare, în cursa de a crește performanța de viteză, s-a ajuns în stadiul în care:

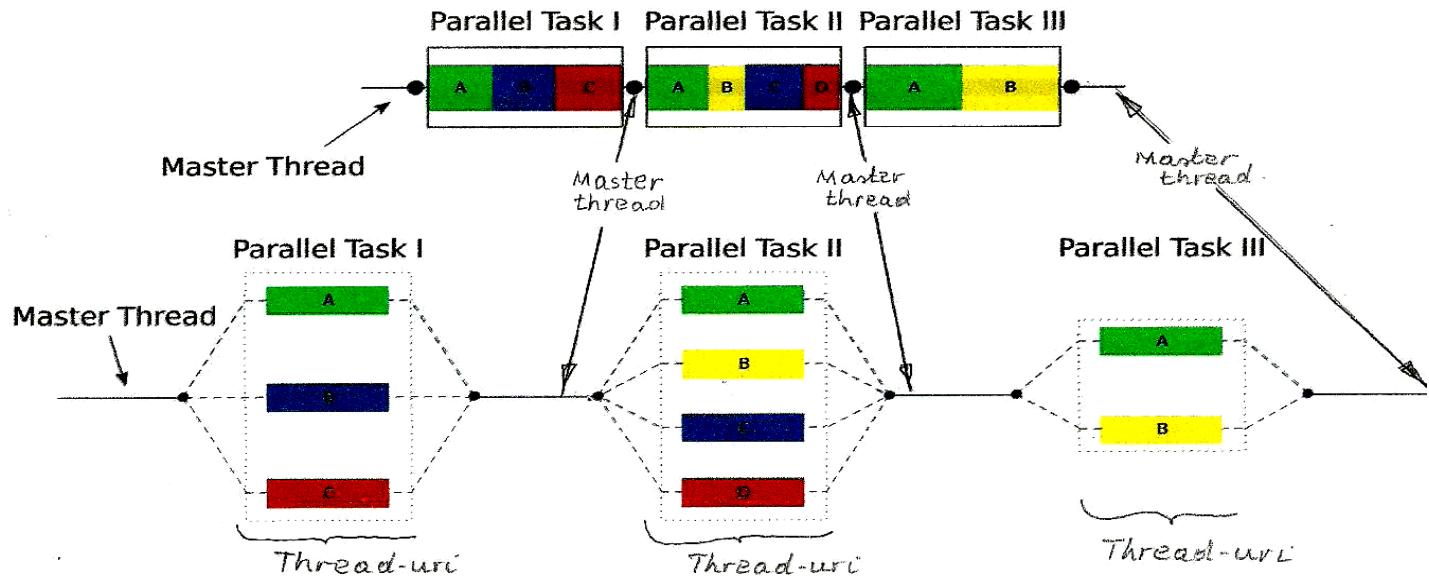
- puterea disipată, datorată creșterii peste 3-3,5GHz, să nu poată fi suportată de chip;
- multitudinea de resurse hardware (circuistică) de pe chip să fie subutilizate;
- complexitatea părții de control necesită un efort prea ridicat în proiectare și chiar consum de arie.

Acste aspecte au îndreptat atenția firmelor spre implementări sub formă de procesoare multicore, cu core simple care să nu aibă dezavantajele amintite ale procesoarelor superscalare. S-ar părea că numărul de core pe chip se dublează la o distanță în timp de doi ani, ceea ce conexează cu legea lui Moore referitoare la densitatea de componente integrate pe chip. Numărul de core pe chip, tipul de core, modul lor de organizare și de

intreconectare, încât să rezulte procesoare performante, sunt încă aspecte care se cercetează. Oricum, se pare că arhitectură de tip von Neumann, cu procesare secvențială, de tip uniprocesor, după mai bine de 50 de ani va preda ștafeta procesoarelor de tip multicore. Dar, pentru impunerea procesoarelor multicore, nu hardware-ul este componenta reluctantă ci componenta de software nu este încă suficient dezvoltată, care ar trebui fie să modifice softul existent pentru o procesare paralelă (parallel processing program), fie mai ales să genereze soft paralel.



Suport pentru rescrierea programelor existent pentru o procesare paralelă pe bază de thread-uri este programul **OpenMP** (Open Multi-Processing), www.openmp.org. OpenMP este un API (Application Programming Interface) portabil, scalabil, pentru dezvoltare de aplicații paralele pe platforme multiprocessor, SMP, programate în C/C++ și Fortran sub sistemele Unix & Windows NT. Progamaratorul prin directivele de compilare “croiește” aplicația pentru parametrii sistemului multiprocesor. În fond, OpenMP este o metodă de paralelizare a unui program secvențial, prin ramificare (“fork”) într-un număr specificat de thread-uri (vezi componentele s și p din introducerea de la subcapitolul 5.3), care apoi sunt alocate pe procesorul multicore sau pe sistemul multiprocesor și rulate în paralel. Reprezentarea din figura următoare schițează modul prin care după un segment de program secvențial (“master thread”) urmează un segment ce poate fi paralelizat, deci este divizat într-un număr de thread-uri cu procesare paralelă, iar la terminarea cărora poate urma din nou un segment secvențial (master thread) care în continuare iarăși poate fi ramificat în thread-uri.



5.5 PROCESOARE GRAFICE

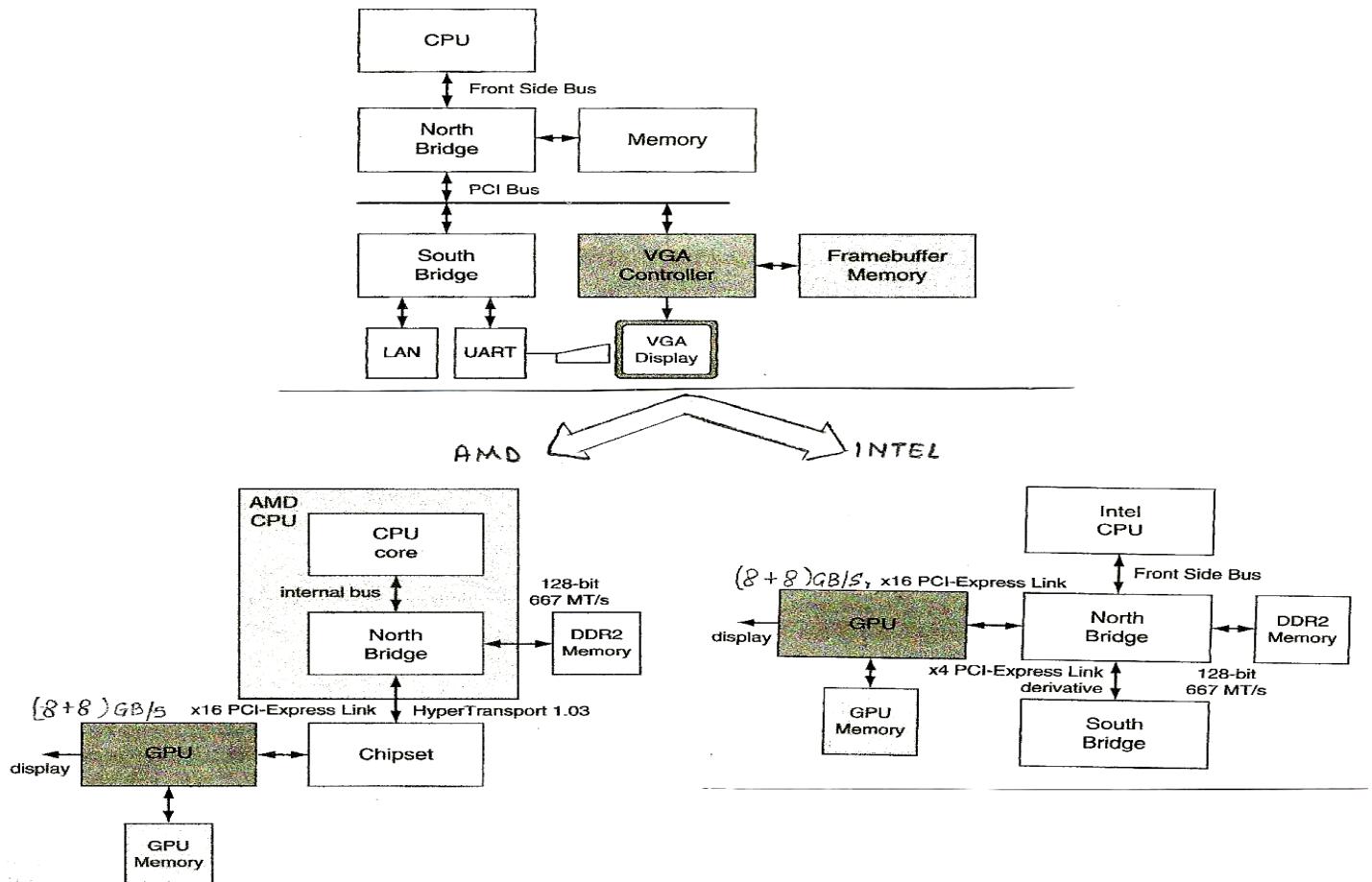
Unitatea de procesare grafică **GPU** (Graphical Processing Unit) există în oricare PC, laptop, desktop sau stație de lucru. Fundamental, GPU generează grafică 2D sau 3D, imagine sau video prin **întrepătrunderea procesării grafice cu calculul paralel**, realizând ceea ce este referit cu termenul **visual computing**; astfel având aceste facilități GPU a devenit un procesor paralel programabil. GPU împreună cu CPU formează un sistem heterogen, în care GPU, ca și coprocesor, are rolul de a degrevă procesorul, CPU, de toate sarcinile de procesare grafică.

În figura următoare sunt prezentate, sumar, două stadii din evoluția componentei grafice a unui sistem; în primul desen la nivelul anului 1990, iar în următoarele două reprezentări sunt structurile de principiu pentru implementările, la nivelul anului 2008, ale firmelor Intel și AMD. Inițial, grafica pe un PC era realizată de un controller VGA (Video Graphics Array) care era, în fond, un simplu controller pentru memoria (buffer) de cadre/frame și generator pentru display, conectat la magistrala PCI (vezi structura următoare la nivelul 1990).

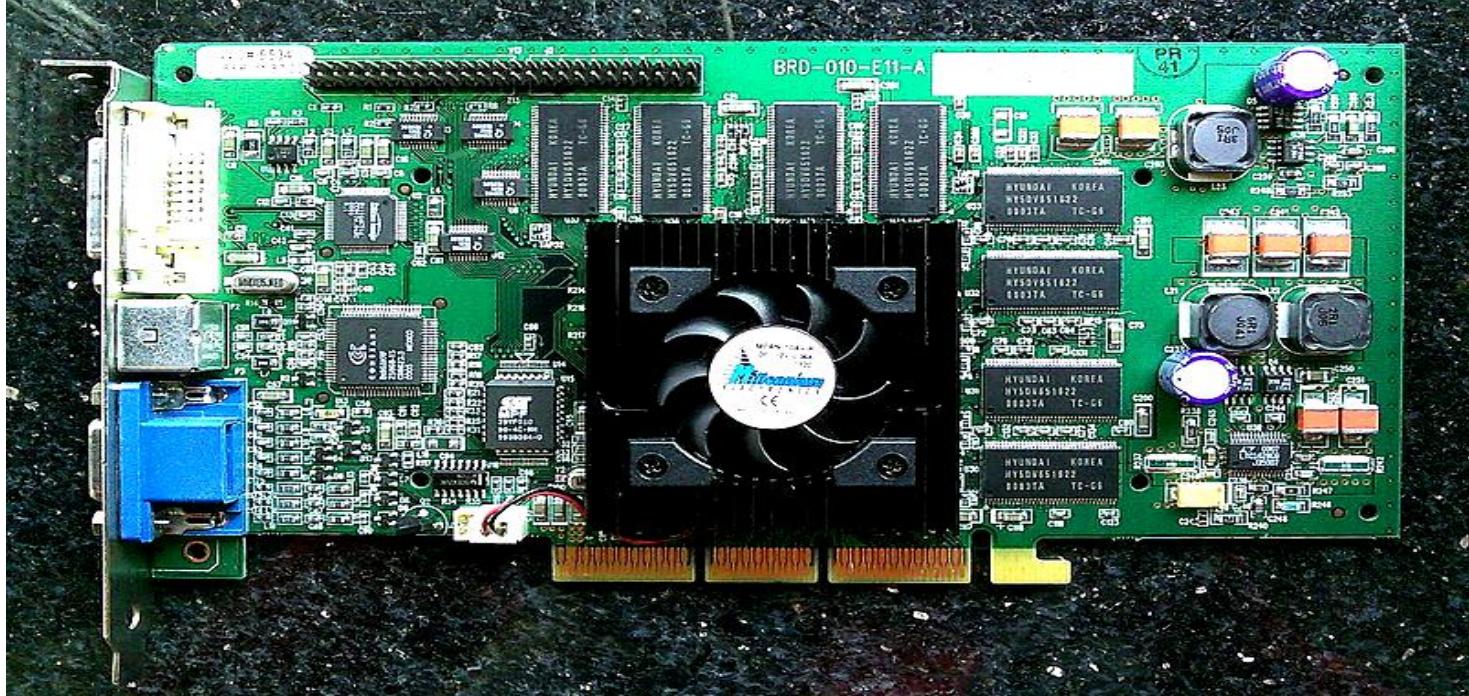
Actual, există două configurații uzuale de conectarea GPU la sistemul procesor + memorie sistem:

- configurația pe bază de procesor Intel, GPU+memoria locală (GPU memory) este conectat la north-bridge printr-o magistrală PCIe.2 cu 16 lane-uri realizând un bandwidth 16 GB/s (8GB în fiecare direcție); la următoarea generație GPU și north bridge vor fi intregreate împreună;
- configurația pe bază de procesor AMD, GPU+memoria locală (GPU memory) este conectat la chipset (south bridge) printr-o magistrală PCe.2 cu 16 lane-uri realizând un bandwidth 16 GB/s (8GB în fiecare direcție); la următoarea generație GPU și chipset vor fi intregreate împreună;

CPU și GPU au fiecare acces la memoria celuilalt, evident cu un bandwidth mai mic decât prin accesul direct la memoria proprie. Există și o variantă de low-cost, UMA (Unified Memory Architectures), cu o singură memorie comună, memoria sistemului, dar în această variantă accesul GPU la memoria sistem se realizează cu un bandwidth mult mai mic decât accesul la o memorie proprie; sau, în opziție, există și varianta de înaltă performanță, în care sunt în sistem 2 – 4 GPU-uri în paralel. CPU+GPU formează un sistem heterogen, CPU adecvat pentru procesare secvențială/serială, iar GPU eficient pentru grafică și calcul paralel.



Imagine placă video Nvidia, Quadro (SGI VPro VR3)



Sarcina unui GPU în sistem este generarea componetei grafice, ceea ce presupune o procesare rapidă și masivă de date, o sumară estimare rezultă din următorul calcul simplu. Pentru un ecran cu dimensiunea de 2560x1600 pixeli, cu o frecvență de înscriere de 24 cadre(frame)/s, la o frecvență de clock de 1 GHz, procesorul trebuie să calculeze un pixel la fiecare 10 tacte de ceas, $(10^9 \text{ tacte/s}) : (2560 \times 1600 \times 24) \text{ pixel/s} = 10 \text{ tacte/pixel}$, iar la o frecvență de înscriere de 60 cadre (frame)/s trebuie să calculeze un pixel la fiecare 4 tacte de clock.

Dacă inițial GPU avea ca utilitate numai programare de grafică pe baza unei API (Application Programming Interface) dedicate pentru grafică, mai nou pe baza platformei CUDA sunt rezolvate și sarcini de calcul paralel masiv utilizând limbajele C, C++, Fortran, OpenCL, DirectComput și altele. Aceste noi procesoare de tip "GPU Computing" sunt referite prin abreviația **GPGPU** (General-Purpose computation on GPU).

Imaginea unei plăci de bază în care este introdus cardul (GeForce) pentru componenta grafică a sistemului este prezentată în figura următoare

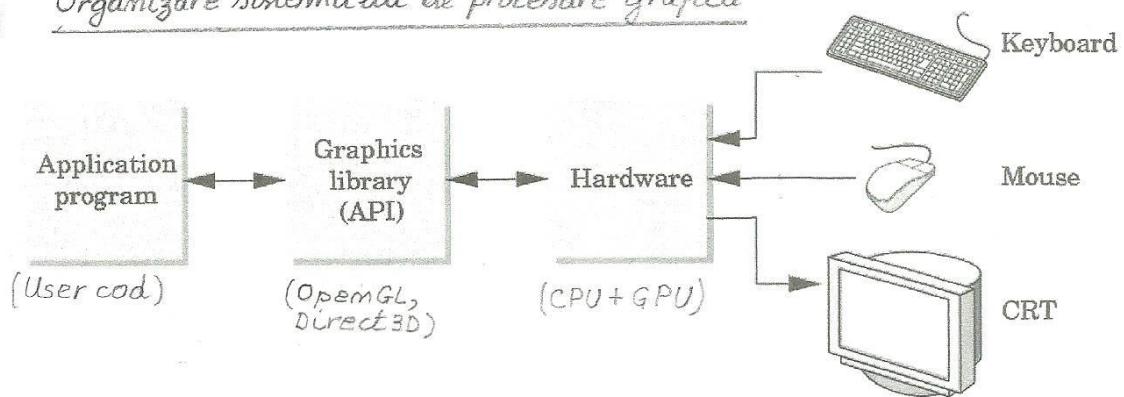


1. Noțiuni de procesarea grafică 3-D

Obiectul procesării grafice 3D constă în transformarea unui obiect în imagine sa plană 2D pentru a fi reprezentată pe ecran. Structurarea unui astfel de sistem pentru procesare grafică este reprezentată în Figura a).

Pentru reprezentarea imaginii pe ecran trebuie întâi ca obiectul să fie descris, utilizând un software de modelare 3D, într-un model de reprezentare a formei obiectului respectiv. Soft-ul de interfață, API (Application Programming Interface) din acest sistem de procesare, larg utilizat actual, de facto standard, este reprezentat de platformele **OpenGL** (Open Graphics Library, inițiat de Silicon Graphics) și **Direct3D** (inițiat de Microsoft). Structurarea logică (simplificată) a etapelor de procesare grafică pe care se bazează aceste platforme, care realizează o procesare de tip pipeline, este redată în figura b). Acest pipeline pentru procesare grafică, referit ca **rendering** (redare, generare) **pipeline**, este compus din etape/componente fiecare având o funcție specifică. Rendering este procesul de generarea unei imagini, cu ajutorul unui program de calculator, pornind de la un model. Modelul este descrierea într-un limbaj a unui obiect tridimensional sau o structură de date. Imaginea obținută, conform acestui pipeline de procesare, poate fi o imagine digitală sau o imagine grafică pe ecran. Transformările realizate în etapele din pipeline sunt prezentate (simplificat) în Figura c)

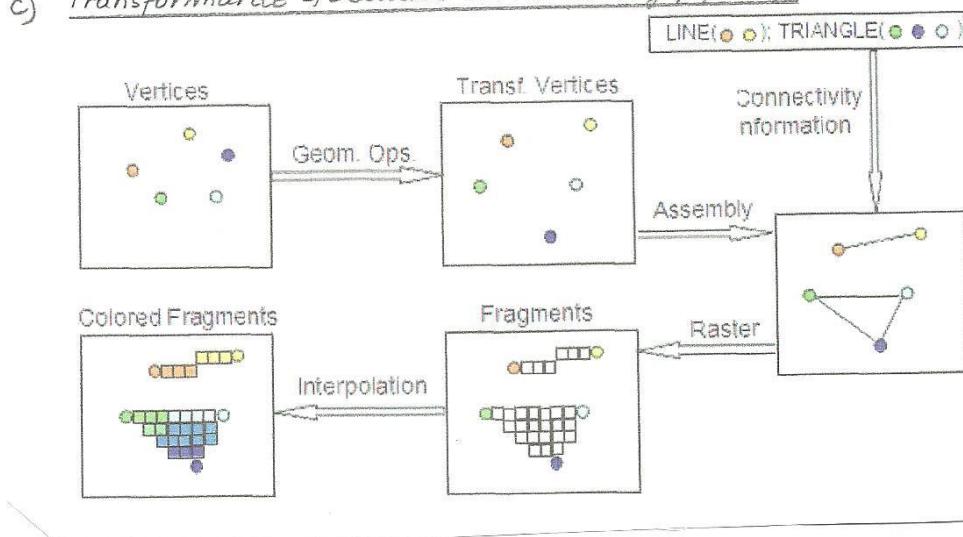
a) Organizare sistemului de procesare grafică



RENDERING PIPELINE

b) Etapele în rendering pipeline

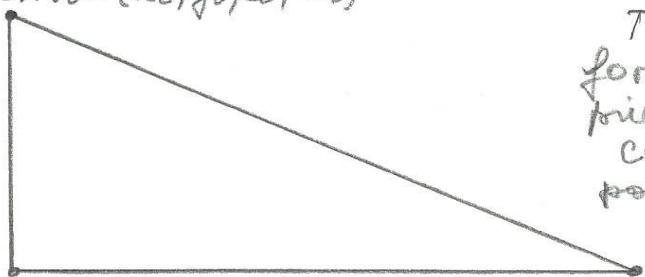
c) Transformările efectuate în rendering pipeline



Modelul de reprezentare (*descrierea scenei*) al unei forme se bazează pe primitive geometrice de reprezentare care sunt: punctul, linia dreapă și triunghiul. Triunghiul este forma geometrică plană cea mai simplă care poate fi realizată doar din trei linii drepte. Cu această formă simplă – triunghi – se poate compune modelul 3D “triunghiularizat” pentru reprezentarea oricărei forme spațiale a unui obiect, utilizând un număr mai mic sau mai

mare de triunghiuri, în funcție de acuratețea cu care se dorește reprezentarea obiectului respectiv. În figururile următoare sunt exemplificate modelele compuse din triunghiuri a trei obiecte: cub, sferă și un ceainic.

vertex $v_0 = (x_0, y_0, z_0, w_0)$



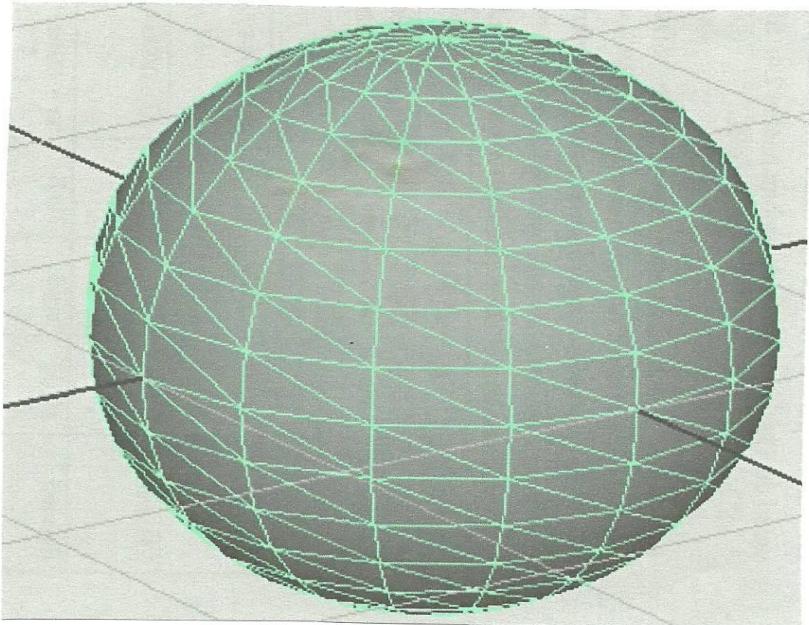
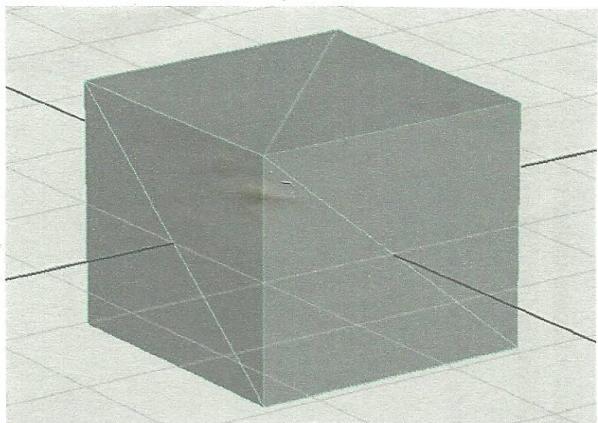
Triunghiul este cea mai simplă formă plană care poate fi definită prin liniile drepte.
Cu suficiente triunghiuri se poate compune modelul 3-D al oricărui corp.

vertex $v_2 = (x_2, y_2, z_2, w_2)$

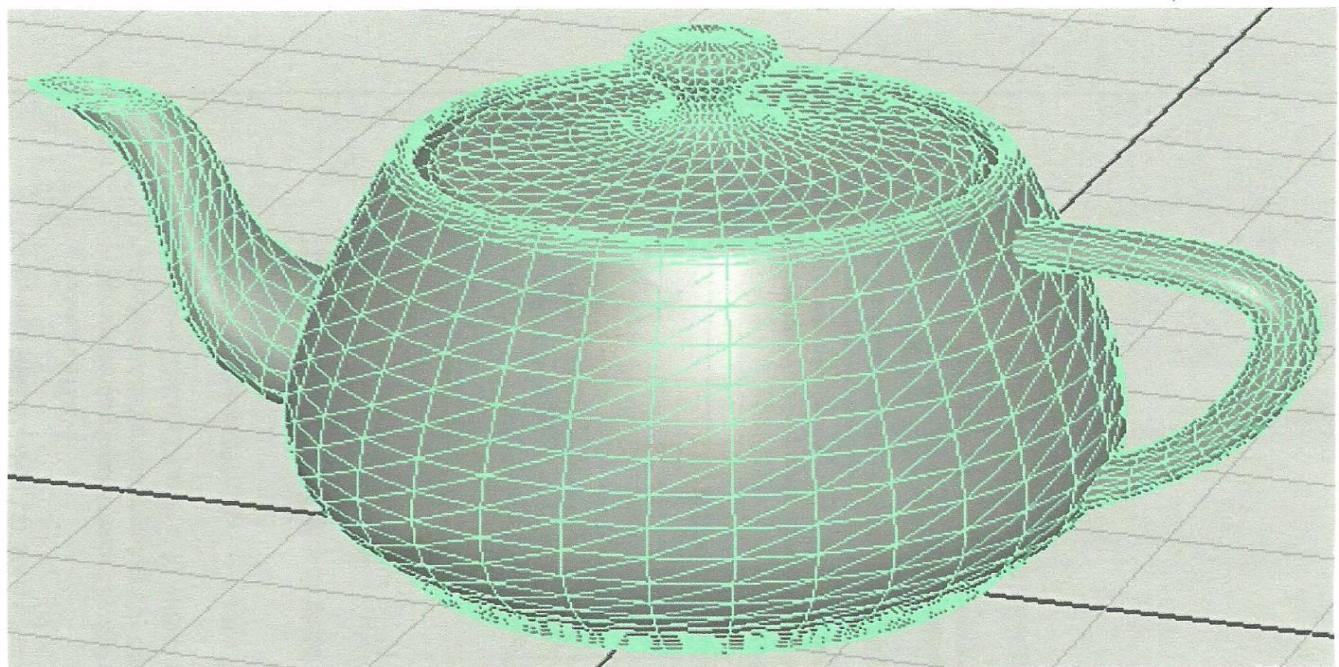
vertex $v_3 = (x_3, y_3, z_3, w_3)$

Modelul 3-D pentru o sferă compus din triunghiuri; cu suficiente triunghiuri curba sferică poate fi realizată suficient de ușor

Modelul 3-D pentru un cub ale cărui fețe sunt realizate din triunghiuri; modelul include și fețele care nu se văd



Cadrul "din săru" 3-D pentru un ceainic, realizat din triunghiuri



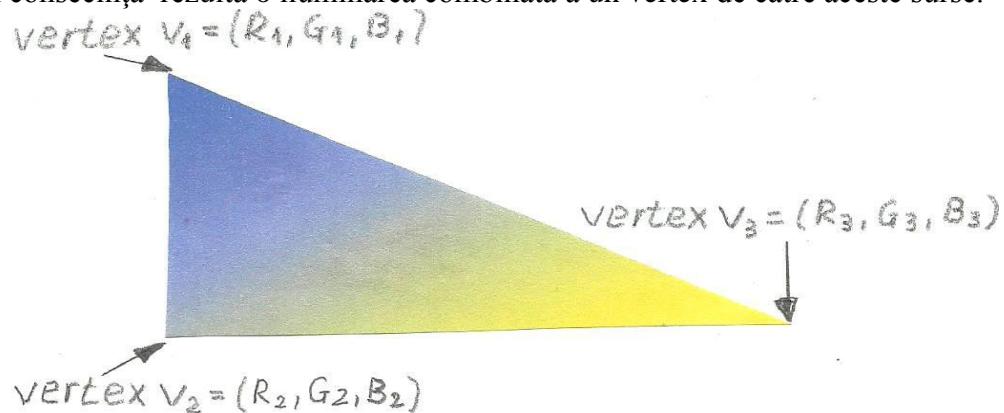
Fiecare vârf al triunghiului – **vertex** (punct vertical) – este caracterizat prin patru coordonate (x, y, z) și distanța w a vertex-ului față de cameră (ochi). Valorile acestor coordonate (x, y, z, w) pot fi exprimate ca numere în simplă precizie în virgulă flotantă, sub forma unui cuvânt binar de 32 biți (în standardul IEEE-754). În reprezentarea sub forma unui “cadru din sărmă”, pentru ceainicul din figura anterioară, fiecare triunghi component are propriile-i cordonate ale vârfurilor (vertex-urilor).

Prin *transformările/ procesarea vertex-urilor*, de forma $V' = f(V)$, pentru toate triunghiurile, care compun un obiect 3D, modelul obiectului respectiv poate fi deplasat, poate fi rotit, scalat etc. Aceste transformări au o exprimare sub forma unor ecuații matriceale, de exemplu pentru o translație cu distanța t și o rotație cu unghiul ϕ ecuația este următoarea:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \cos \phi & -\sin \phi & 0 \\ \sin \phi & \cos \phi & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$$

Dacă modelul triunghiularizat al ceainicului din figură este compus din 10 000 de triunghiuri, rezultă că modelul său de pe ecran pentru un cadru necesită 30 000 de transformări de vertex-uri, ceea ce înseamnă calculul a 120 000 de cordonate (x, y, z, w), fiecare coordonată fiind un cuvânt de 32 biți (precizie simplă în IEEE 754). Programele care operează asupra datelor grafice (vertex, pixel, fragment) în limbajul procesărilor grafice sunt referite ca programe **shader** (shader programs).

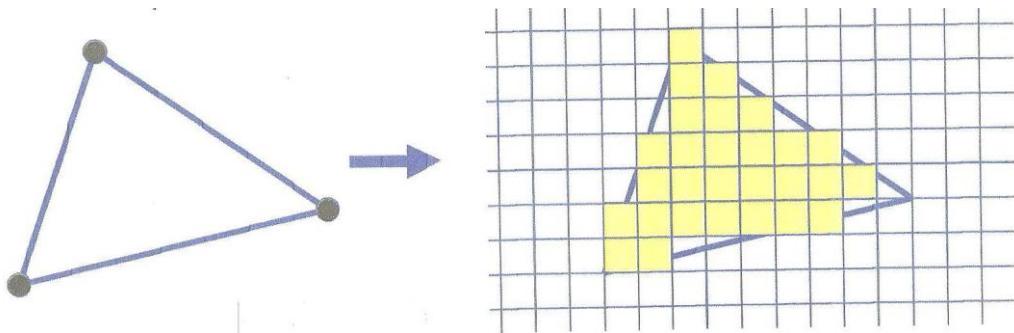
Dacă vertex-urile au culori diferite, aceasta înseamnă că există un gradient cu o variație continuă la trecerea dintre culorile de pe suprafața triunghiului. Mai realistic, pentru informația de îmbinare a culorilor se pot considera surse de lumină, în consecință rezultă o iluminare combinată a un vertex de către aceste surse.



Programul **vertex shader** calculează pentru modelul triunghiularizat al obiectului: pozițiile vertex-urilor, orientarea, culoarea și transparența generând un sir de date vertex sub forma unui sir de numere în virgulă flotantă.

Apoi un program **geometric shader** (*procesare primitive*) operează asupra sirului de date vertex (produs de vertex shader, care conține informația despre vertex-uri și conectivitatea acestora) generând, pentru formatul 2D (plan) al screen-ului, primitive geometrice (puncte, linii, triunghiuri).

Imaginea obiectului pe ecran se obține din proiecția acestor primitive pe ecran, imagine ce se compune din pixeli, deci toate primitive trebuie să fie convertite în pixeli (*rasterizare*). De fapt se realizează fragmente de pixeli ca în figura următoare.



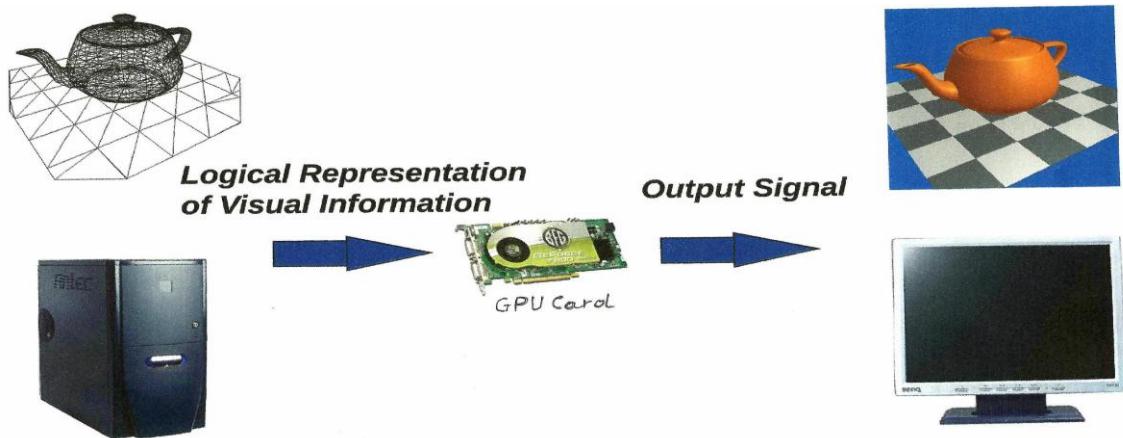
De ce se lucrează cu fragmente de pixel? pentru că în culoarea unui pixel se combină culorile din mai multe triunghiuri vecine, a se vedea nuanțele și reflexiile pentru ceainicul reprezentat în figura următoare. Apoi urmează *procesarea fragmentelor* cu programul **pixel shader**. Operația de determinare a culorii și transparenței unui pixel este referită ca **fragment shader**. Pentru fiecare pixel sunt calculate componente de culoare (R; G, B) și componenta de transparență alpha, adică opacitatea: valoarea 1 corespunde unui pixel complet opac, iar valoarea 0 pentru un pixel complet transparent. Mai nou, cele patru componente (R, G, B, alpha) sunt reprezentate ca numere flotante în simplă precizie, pe cuvinte de 32 biți, conform standardului IEEE 754. Anterior, cele patru componente erau reprezentate ca numere întregi pe cuvinte de 8 biți



Se poate introduce apoi, tot printr-o operație de shader, și o componentă artistică, adică maparea unei texturi pentru fiecare fragment de pixel sau pentru un pixel, operație referită ca texturare (texture), ca în figura următoare.



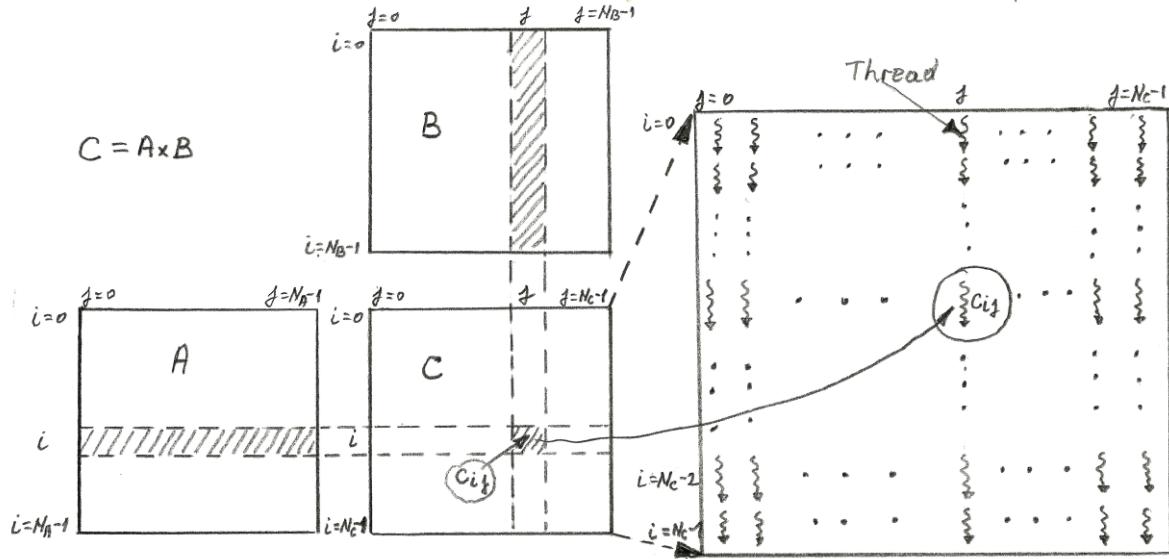
Toate operațiile necesare procesării grafice 3-D, descrise anterior, sunt efectuate pipelinizat (rendering pipeline) cu implementare în soft sau cu implementare în hard. Schematic, această procesare pe un GPU este reprezentată în imaginea următoare



2. Cuda

Foarte mult din procesările efectuate în etapele pipe-lui graphic, prezentat anterior, se reduc la calcule/transformări de matrice de dimensiuni ridicate, care prezintă un grad masiv de paralelism. Se va exemplifica gradul înalt de paralelism care apare la înmulțirea a două matrice pătrate A și B de dimensiune N_A și N_B , rezultând matricea produs C de dimensiune N_C ; se consideră că dimensiunile (widths) sunt $N_A = N_B = N_C$. Un termen produs, C_{ij} al matricei rezultat C se calculează ca sumă a tuturor produselor dintre elementele corespunzătoare din linia i a matricei A cu elementele corespunzătoare din coloana j a matricei B, conform modului reprezentat în figura următoare, cu relația

$$C_{ij} = A_{i0} \cdot B_{0j} + A_{i1} \cdot B_{1j} + \cdots + A_{i(N_A-2)} \cdot B_{(N_A-2)j} + A_{i(N_A-1)} \cdot B_{(N_B-1)j} \Rightarrow \text{constituie un thread}$$



Fiecare termen produs C_{ij} se obține prin N operații de înmulțire și N operații de adunare (acumulări succesive). Pentru matrice pătrate cu dimensiunea $N_A = N_B = 1000$ se obțin 1 000 000 termeni C_{ij} , fiecare necesitând 1000 de multiplicări și 1000 de acumulări succesive efectuate asupra termenilor/coeficientilor matricelor, (de exemplu cu o instrucțiune de tipul multiply and add aplicată asupra a 1000 de perechi de date), fiecare termen rezultat poate fi calculat independent de ceilalți, în program constituind un **thread**, deci posibilitatea de efectuare a 1 000 000 de segmente de program (grid de thread-uri) procesate în paralel. Fiecare thread execută același program (aceleași instrucțiuni) dar cu date diferite, deci masivul paralelism de thread-uri este de fapt un SPMD (Single-Program Multiple-Data) ce poate fi efectuat pe o organizare multiprocesor de tip SIMD.

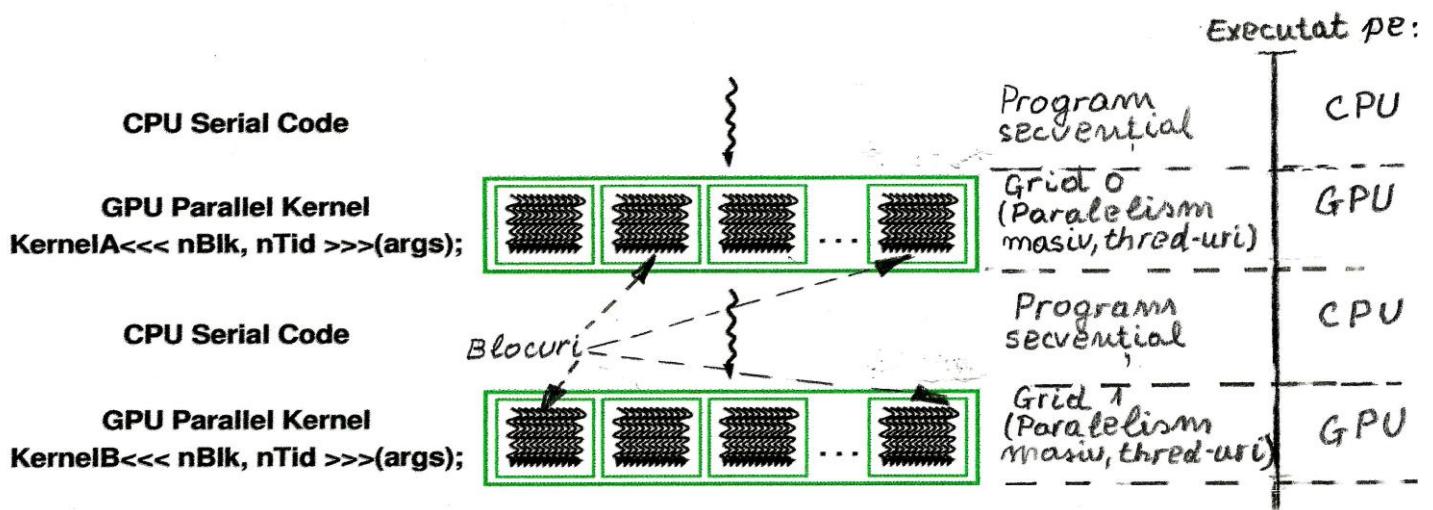
- Structura programului de aplicație

Pentru elaborarea de aplicații pe un sistem eterogen, format din CPU (**host**) + GPU (coprocesor, **device**), firma Nvidia a elaborat platforma **CUDA** (Compute Unified Device Architecture) care este o platformă hardware și software pentru aplicații de calcul paralel accesibile prin limbi standard (extinse) C, C++, Fortran, OpenCL (Open Computing Language), DirectComput și altele. (Recomandăm pentru CUDA cursul <http://courses.engr.illinois.edu/ece498/al/textbook/>).

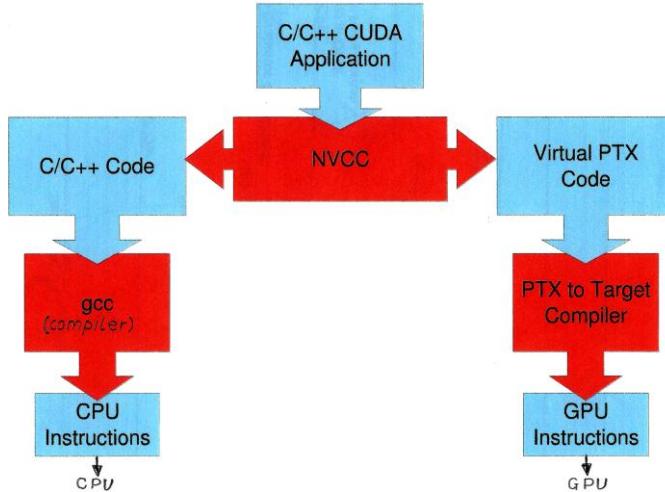
CUDA este o soluție elegantă pentru problema exprimării paralelismului în majoritatea algoritmilor. În acest sens CUDA unifică toate formele de paralelism (multithreading, MIMD, SIMD și chiar ILP) într-un paralelism elementar sub forma unei primitive de programare - **CUDA Thread**; compilatorul și hardware generând sute de astfel de CUDA Thread. În consecință, modelul de programare CUDA se poate fi privit ca Single Instruction, Multiple Thread (SIMT). Deși materialul din această prezentare se bazează pe CUDA și pe procesorul GPU - NVIDIA, aceleași idei sunt baza și pentru procesoarele GPU produse de alte firme precum și în limbajul de programare OpenCL.

Un program dezvoltat pe această platformă conține una sau mai multe părți a căror execuție se repartizează între CPU (host) și GPU (device); partea care conține paralelism redus sau deloc se implementează în cod C/C++ pentru host, iar partea care este bogată în paralelism masiv în cod C/C++ dialect pentru device.

În figura următoare este prezentată o structură de program care conține două segmente de program de procesare serială (CPU Serial Code) alternând cu două segmente de program de procesare paralelă (GPU Parallel Kernel, respectiv grid 0 și grid 1). Funcția **kernel** (KernelA <<< nBlk, Ntid >>> arg) când este întâlnită în program prin execuția sa generează un grid de thread-uri, adică se repetă aceeași succesiune de instrucțiuni (thread) de un număr foarte mare de ori (în exemplul dat anterior 1 000 de thread-uri), care sunt executate în paralel pe un număr oarecare de procesoare (PE, Processing Element) disponibile. Fiecare thread execută aceeași funcție kernel



Programul de aplicație supus compilatorului NVIDIA C Compiler (NVCC), vezi figura următoare, este separat pe cele două tipuri de componente/segmente de program. Segmentele de cod serial sunt compilate cu compilatoare standard (gcc) și rulate pe procesorul host, CPU. Segmentele de paralelism masiv, gridurile (de threaduri) generate de funcțiile kernel sunt transalatate de NVCC în codul ISA al GPU în două etape: întâi din C/C++ într-un cod intermediar al unei mașini virtuale PTX (Parallel Thread eXecution), care are un pseudo-limbaj (intermediar) de asamblare, apoi din acest cod intremediar PTX în codul propriu GPU, instrucțiunile ISA.



Pentru dezvoltarea programelor în CUDA, s-a realizat sistemul de dezvoltare denumit NVIDIA NEXUS, orientat pentru aplicații de sisteme eterogene, CPU+GPU; Mai mult, pentru a se ușura etapa de elaborare a aplicațiilor pentru sisteme eterogene, sistemul Nexus a fost integrat în platforma Microsoft Visual Studio, ceea ce oferă procesului de dezvoltare al aplicațiilor pentru CPU+GPU aceeași ușurință ca și cea în dezvoltare de aplicații pentru sisteme CPU

- Hierarhizarea thread-urilor

Thread-urile generate de execuția unei funcții kernel, ușual în număr de mii sau chiar milioane, sunt organizate pe două niveluri: **la nivel de grid și la nivel de bloc**.

Un grid este compus dintr-un număr de blocuri (dimGrid), blocurile grid-ului respectând o structurare matriceală, dimensiunile după cele trei direcții fiind specificate prin gridDim.x, gridDim.y și gridDim.z. În general, dimensiunile gridDim.x și gridDim.y pot fi între 0 și 65 536, iar gridDim.z = 1, adică o matrice numai după două dimensiuni. Dimensiunile grid-ului se specifică la lansarea în execuție a funcției kernel în felul următor

dim3 dimGrid(gridDim.x, gridDim.y, 1)

În cadrul matricei unui grid, coordonatele unui bloc se exprimă: block(blockId.x, blockId.y) în care indicii blockId.x, blockId.y pot avea valori între 0 și gridDim.x - 1 respectiv între 0 și gridDim.y - 1.

La nivel de bloc, fiecare bloc conține același număr de threaduri structurate matriceal tridimensional, fiecare thread având valorile de coordonate : trThreadId.x după x, trThreadId.y după y și trThreadId.z după z. Dimensiunile după cele trei direcții se determină în funcție de numărul de threaduri din bloc, dimBlock, (dimensiunea blocului) care se specifică la lansarea în execuție a funcției kernel în felul următor

dim3 dimBlock(trThreadId.x, trThreadId.y, trThreadId.z)

De exemplu, dacă parametrul dimBlock, specificat la lansarea funcției kernel, este de 512 threaduri, structurări după cele trei direcții pot fi de forma : (512, 1, 1); (8, 16, 2); (16, 16, 2), dar nu (32, 32, 1) deoarece $32 \times 32 \times 1 = 1024 > 512$ threduri!

Crearea unei organizări de threaduri la lansarea unei funcții kernel se exprimă astfel:

**dim3 dimGrid(gridDim.x, gridDim.y, 1)
dim3 dimBlock(threadId.x, threadId.y, threadId.z)
KernelFunction <<< dimGrid, dimBlock>>>(...);**

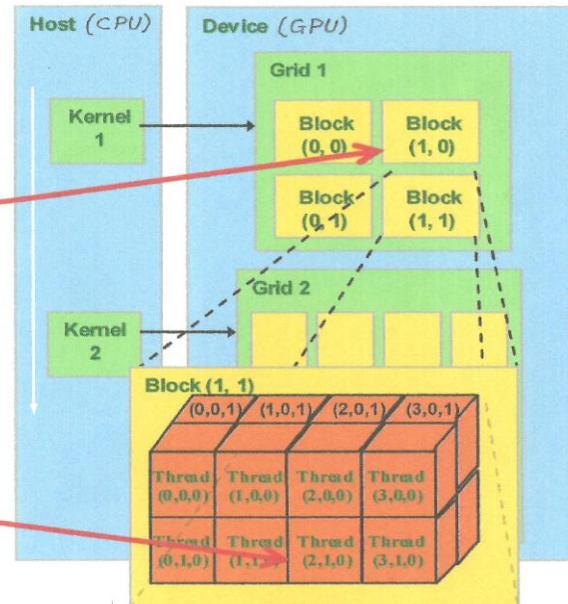
<<< dimGrid, dimBlock>>> specifică numărul total de threaduri

(=[nr blocuri (**dimGrid**) x nr. threaduri într-un bloc (**dimBlock**)]) dintr-un grid care se execută la apelarea funcției kernel.

```
Dim3 dimGrid(2, 2, 1)
Dim3 dimBlock(4, 2, 2);
KernelFunction<<<dimGrid,
dimBlock>>>(...);
```

Block(1,0) has
~~blockIdx.x=1 & blockIdx.y=0~~

Thread(2,1,0) has
~~threadIdx.x=2 ; threadIdx.y=1;
threadIdx.z=0~~



În exemplul de program, din figura anterioară, sunt două apelări de funcție kernel care generează grid1 și grid2. Grid 1 este creat prin lansarea funcție kernel de forma:

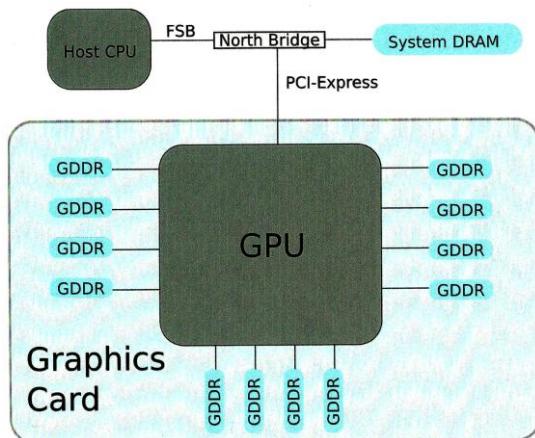
**dim3 dimGrid(2, 2, 1)
dim3 dimBlock(4, 2, 2)
KernelFunction <<< dimGrid, dimBlock>>>(...);**

Grid 1 este organizat matriceal 2x2x1 blocuri, iar block(1,0) este expandat în partea de jos a figurii având o organizare de 4x2x2, adică 16 threaduri/bloc; în total grid 1 are dimensiunea 4 blocuri x 16 threaduri/bloc = 64 threaduri. (Figura nu detelișă și pentru grid 2).

3. Arhitectura procesorul grafic

Arhitectura unui sistem multiprocesor eterogen, CPU+GPU, este prezentată în figura următoare. Placa grafică este conectată la sistem printr-o conexiune PCIe x16 (8+8)GB/s. GPU necesită un schimb masiv de date cu memoria locală de pe placa grafică, deci magistrala spre memoria sa trebuie să asigure un Bandwidth foarte ridicat (> 100GB/s) în consecință se realizează cu o lățime, w, de cel puțin șase canale fiecare de 64 biți (w=6x64= 384 biți). În figura următoare este figurată pe placa grafică posibilitatea de conectare prin 12 canale la

memoria DRAM, aceste circuite de memorie sunt de tipul DDR3 (Double Data Rate 3) sau GDDR5 (Graphics DDR5).



Arhitectura și organizarea unui GPU se va prezenta printr-o exemplificare pe un produs comercial foarte cunoscut, procesorul grafic GTX 480 al firmei Nvidia [1]. Firma Nvidia a introdus:

1. 2006 – arhitectura G80
2. 2008 – GT200(a două revizie pentru arhitectura G80)
3. 2011 – arhitectura Fermi (GF100) este implementată pe un chip cu 3 miliarde ($3 \cdot 10^9$) de tranzistoare

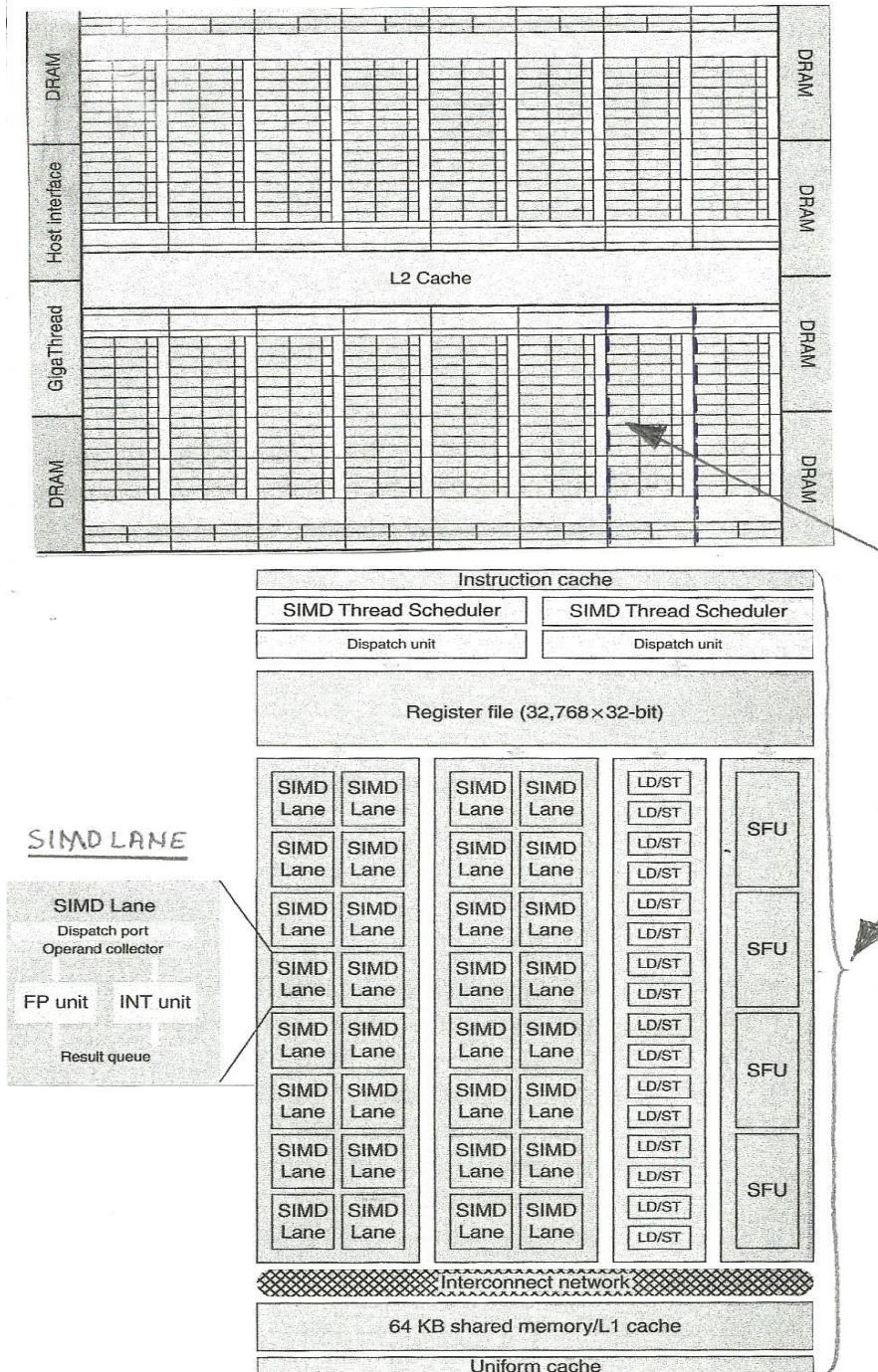
Layoutul acestui procesor este prezentat în figura următoare a). De fapt, în general, *un GPU este un multiprocesor compus din mai multe multithreaded SIMD procesoare, organizare scalabilă în funcție de performanțele cerute*. Acest procesor grafic conține 16 multithreaded SIMD procesoare, sunt cele 16 dreptunghiuri verticale poziționate sub și deasupra memoriei cache L2. În partea dreaptă este poziționat planificatorul/programatorul/schedulerul blocurilor de threaduri (Thread Block Scheduler), Giga Thread. Rolul acestui scheduler în cadrul GPU este de a selecta, dintr-un grid, blocurile de threaduri și a le repartiza la un anumit multithreaded SIMD procesor (la o analogie cu o procesarea clasica un bloc de threaduri ar corespunde unei iterații dintr-o buclă, iar gridul ar corespunde buclei compuse din toate aceste iterații). Pe părțile laterale ale layoutului sunt 6 porturi pentru memorii GDDR5, fiecare cu o latime de 64 biți (în total $6 \times 64 = 384$ biți, capacitate de memorie fiind de 6GB). Host (CPU) interface este portul spre magistrala PCIe x16 căi, care asigură un bandwidth de 8 GB/s spre host (CPU).

La rândul său multithreaded SIMD procesor are o organizare de tipul SIMD, Figura b), cuprinzând în structura sa multe elemente de procesare, PE (processing element, vezi 5.3.1), numite în acest context **thread procesor/SIMD lane**. În acest exemplu sunt 32 de SIMD lane formând două grupuri de cîte 16, deci GTX 480 conține 512 SIMD lane (16 multithreaded SIMD procesor x 32 SIMD lane).

GPU-FERMI GTX 480

CUPRINDE:

- 16 Multithreaded SIMD processors
- $16 \times 32 = 512$ SIMD lane
- $6 \times 64\text{bit GDDR5 DRAM core}$
interfata (prim $6 \times 64 = 384$ pini)
o capacitate de 6GB memorie



Multithreaded SIMD processor

Contine:

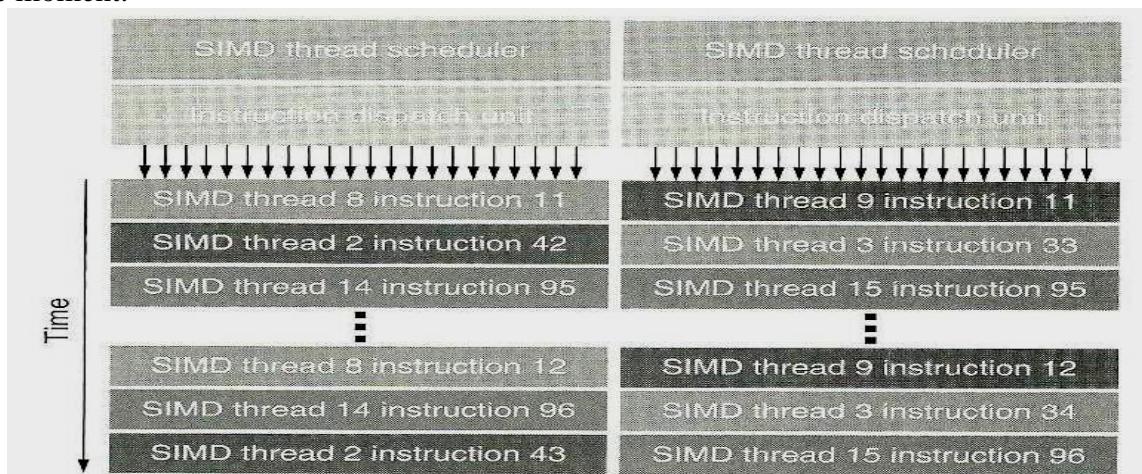
- 32 SIMD lane
- 16 unități LOAD/STORE
- 4 unități SFU(Special Function Unit)
- $32,768 \times 32$ biti registre

Un multithreaded SIMD procesor are organizarea sa următoarele componente:

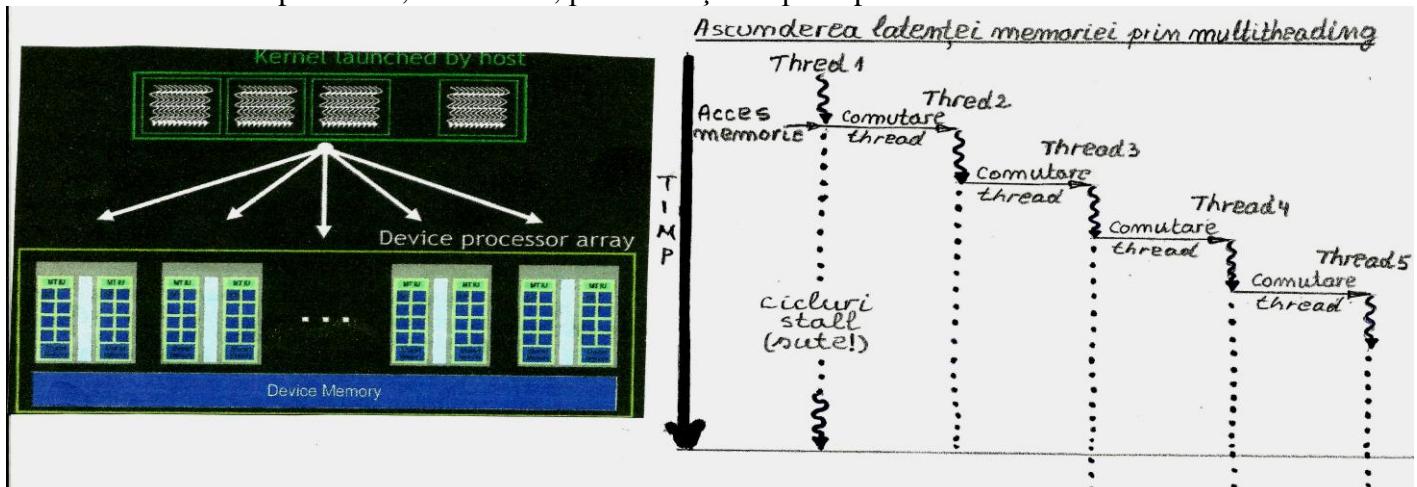
- 32 de procesoare SIMD lane, grupate în două coloane de câte 16
- 16 unități de procesare de tip load/store (pentru lucrul cu memoria)
- 4 unități pentru procesarea funcțiilor speciale ($\sqrt{ }$, $1/\sqrt{ }$, \cos , \sin , $\log(a)/\log 2$ etc)
- două blocuri care selectează (SIMD thread scheduler) și dipecerează instrucțiunile spre unitățile de procesare (sunt selectate două threaduri de instrucțiuni SIMD și trimise simultan, câte unul la 16 SIMD unități).
- 32 728 de registre de 32 biți (se poate privi că fiecare SIMD lane își poate asigna 64 registre cu 32 celule fiecare de 32 biți, pentru simplă precizie; sau 32 registre cu 32 elemente fiecare de 64 biți, pentru dublă precizie). La GTX 480 capacitatea registrelor depășește capacitatea memoriei cache (L1+L2)!

Elementul de procesare, procesorul SIMD lane, este un procesor cu două unități funcționale pipelinizate, una pentru floating-point, celalătă pentru întregi, plus un bloc care dipecerizează instrucțiuni la cele două unități pipeline.

Managementul și lansarea paralelă a threadurilor nu este efectuată de către aplicație sau de către sistemul de operare ci este realizată prin hard de către GPU, pe care există două niveluri de schedulere: cel de blocuri din cadrul unui grid și cel de threaduri din cadrul unui bloc. Pe primul nivel, există un scheduler pentru blocuri de threaduri (thread block scheduler, GigaThread) care selectează și asignează dintr-un grid blocurile de threaduri spre multithreaded SIMD procesor. Pe al doilea nivel, fiecare multithreaded SIMD procesor are unul sau două schedulere pentru threaduri (SIMD Thread Scheduler) care, din blocurile deja asignate aceluia multithreaded SIMD procesor, selectează și dispecerizează threaduri de instrucțiuni spre elementele de procesare (SIMD lane). În scopul ușurării sarcinii schedulerului pentru threaduri, CUDA impune ca blocurile de thread-uri să fie executate independent și în orice ordine. În figura următoare se exemplifică această modalitate de selectare, din cadrul unui bloc de threaduri care a fost asignat acestui multithreaded SIMD procesor, a threadului de instrucțiuni SIMD și lansarea asincronă spre execuție a instrucțiunilor la toate SIMD lane ale acestui multithreaded SIMD procesor. Deoarece threadurile de instrucțiuni SIMD sunt independente, thread scheduler poate selecta un alt SIMD thread în fiecare moment.

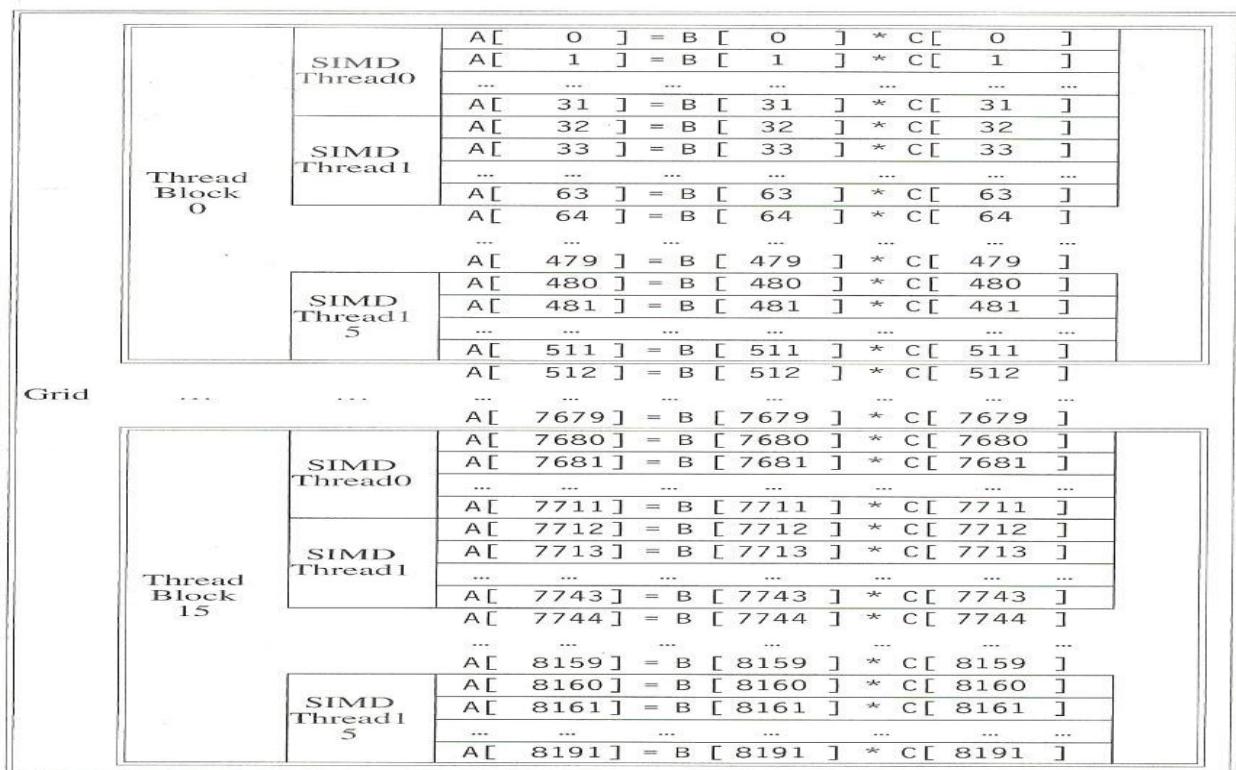


Ideeă în abordarea aplicațiilor specifice pentru GPU se bazează pe presupunerea că aplicațile produc foarte multe threaduri de instrucțiuni independente, astfel că SIMD thread scheduler poate oricând alege oricare thread de instrucțiuni care este gata, fără a fi legat de executarea instrucțiunii umătoare din prezentul thread. În consecință, există o abundență de threaduri care pot fi selectate pentru execuție încât se poate ascunde atât latența memoriei cât și asigurarea creșterii eficienței utilizării permanente a multithreaded SIMD processor. În intervalul de timp când un thread este în aşteptare, datorită latenței memoriei, pot fi rulate multe alte threaduri disponibile până când threadul respectiv devine disponibil, cum este schițat cu thread 1 în figura următoare, astfel că pipelinele elementelor de procesare, SIMD line, pot fi menținute pline prin alimentare cu threaduri.



Pentru GTX 480 în scopul creșterii eficienței utilizării hardului unui multithread SIMD procesor, cele 32 de SIMD line formează două grupuri de câte 16, pentru fiecare grup există câte un scheduler de threaduri și câte o unitate de dispecerizare. Schedulerul selectează dintr-un thread de instrucțiuni SIMD o instrucțiune și o lansează spre toate cele 16 SIMD lane ale unui grup, sau dacă este cazul spre grupul de 16 load/store sau spre gupul de 4 unități de funcții speciale, deci se lansează simultan două instrucțiuni spre execuție (pentru că sunt două SIMD thread scheduler) spre cele 16 SIMD lane. Deoarece o instrucțiune SIMD are o latime de 32 (de date asupra căror operează) aceasta este executată întâi pentru primele 16 date pe grupul de 16 SIMD lane, apoi restul de 16 date tot pe același gup, deci sunt necesare două tacte pentru execuția completă a instrucțiunii. Astfel un thread de instrucțiuni SIMD este selectat la fiecare două tacte de ceas (se procesează complet în paralel două instrucțiuni pe două tacte). Deoarece threadurile sunt independente nu este necesar să se verifice dependența de date în streamul de instrucțiuni.

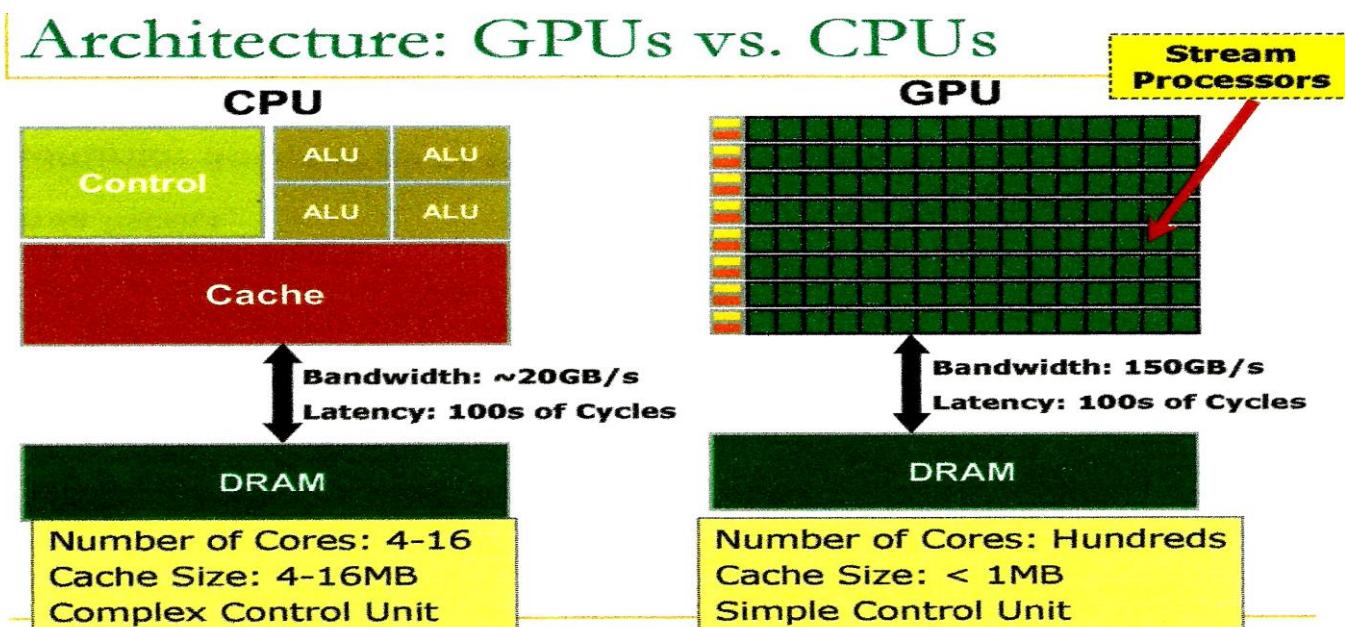
Pentru că acum se cunoaște structurarea grid–bloc de threaduri (thread block) – thread precum și organizarea GPU, exemplificăm: calculul înmulțirii a două matrice coloană $B[i]$ și $C[i]$; $i = 0, 1, 2, \dots, 8191$ prin care se obține matricea coloană $A[i] = B[i] * A[i]$ tot de 8192 elemente. Gândit în maniera clasică această înmulțire ar necesita o buclă pentru care corpul buclei se iterează, pentru înmulțirea $B[i] * A[i]$, de 8192 ori. Structurarea în CUDA este: bucla pentru cele 8192 înmulțiri formează un grid din care se structurează în 16 thread block (corpul buclei) fiecare conținând 512 elemente, iar fiecare din cele thread blockuri compus din 16 SIMD threaduri de instrucțiuni, un thread conține o singură instrucțiune SIMD care execută/inmulțește simultan 32 de elemente ($16 \times 32 = 512$). Această structurare Grid–Thread Block–(SIMD)Thread(instrucțiuni) este reprezentat în figura următoare.



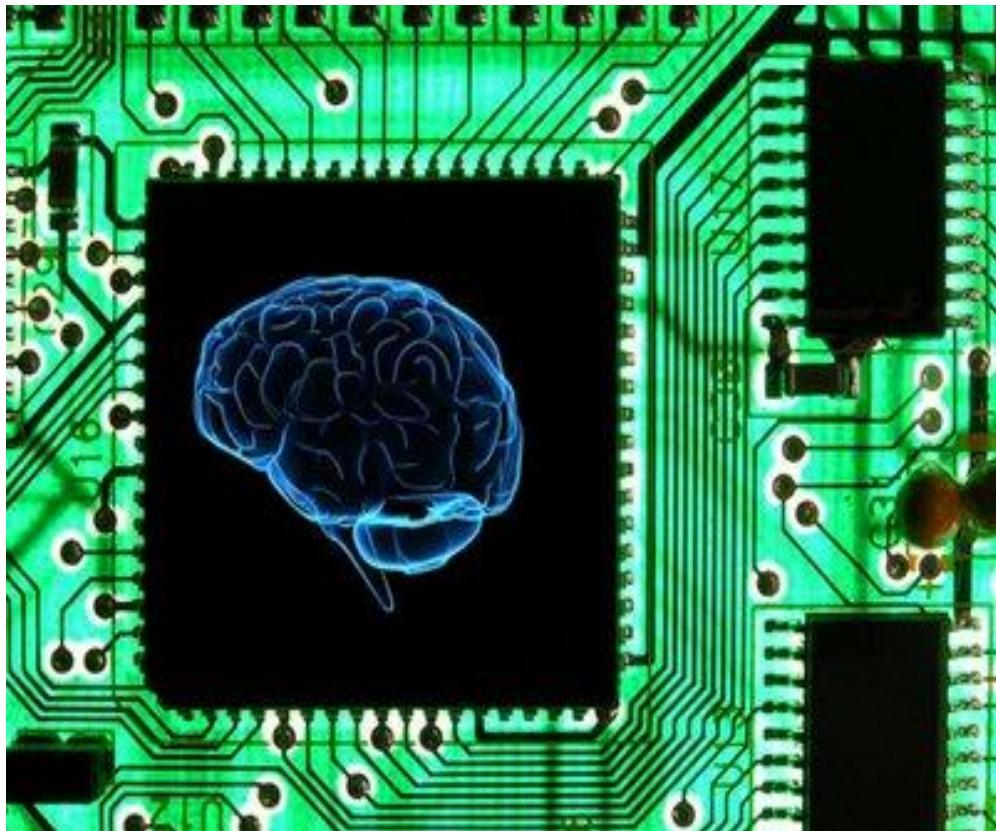
Particularizând pentru GTX 480 care are 16 multithreaded SIMD procesor, la fiecare se asignează câte unul dintre cele 16 Thread Block. Pe unul din cele două grupuri de 16 SIMD lane, ale unui multithread SIMD procesor, se lansează o instrucție (pentru 32 de date) dintr-un SIMD Thread de instrucții (în acest caz SIMD Thread este format dintr-o singură instrucție, cea de înmulțire) care se execută pe două tacte, deci 2 instrucții executate pe două tacte (fiind două grupuri de 16 SIMD lane). Fiind 16 SIMD Thread de instrucții într-un Thread Block rezultă că blocul de instrucții necesită 16 tacte pe un multithread SIMD procesor, și pentru că în GTX 480 sunt 16 multithreaded SIMD procesor, produsul a două matrice de 8192 de elemente este calculat în 16 tacte.

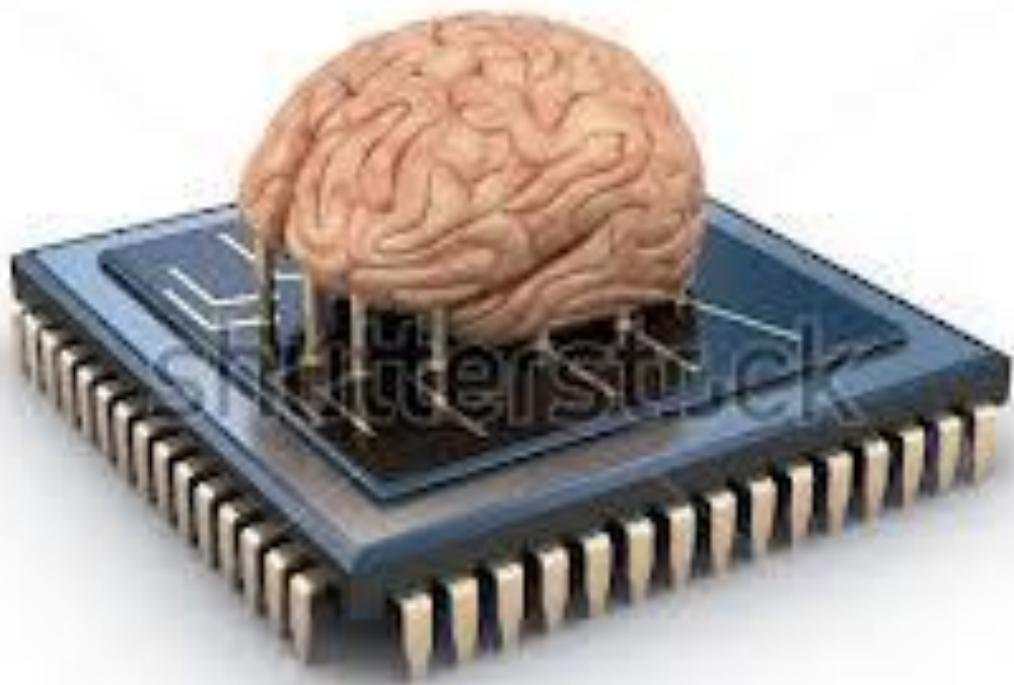
Diferența între arhitectura CPU și GPU, vezi figura următoare, și evident între performanțele de viteză de calcul rezidă în filozofia care stă la baza proiectării lor. Proiectarea CPU este optimizată pentru obținerea de performanță în procesarea secvențială. Aceasta se bazează pe un sistem de control sofisticat de execuție încât să fie exploarat paralelismul la nivel de instrucțiune, ILP. De asemenea, pentru reducerea latenței memoriei s-a devolat un sistem complicat pe două sau chiar trei niveluri de memorie cache. Dar nici ILP nici memoria cache nu a dus la performanțe spectaculoase.

În schimb proiectarea GPU este de a optimiza execuția masivă de threaduri. Astfel în timp ce unele threaduri sunt în aşteptare datorită latenței memoriei sunt suficiente alte threaduri care pot fi procesate asigurând un pipeline fără a fi nevoie de un sistem sofisticat pentru control execuției. De asemenea, memoria cache de dimensiune redusă este necesară doar pentru a ajuta bandwidth, pentru ca nu toate threadurile să acceseze memoria DRAM. În consecință, deoarece spațiul ocupat de cache este foarte redus, zestrea disponibilă de suprafață pe siliciu, pentru a implementa sute de PE (elemente de procesare) cât și pentru a dezvolta calculul în virgulă flotantă, este mult mai mare!



OARE CÂND VA FI POSIBIL?





www.shutterstock.com · 115199059