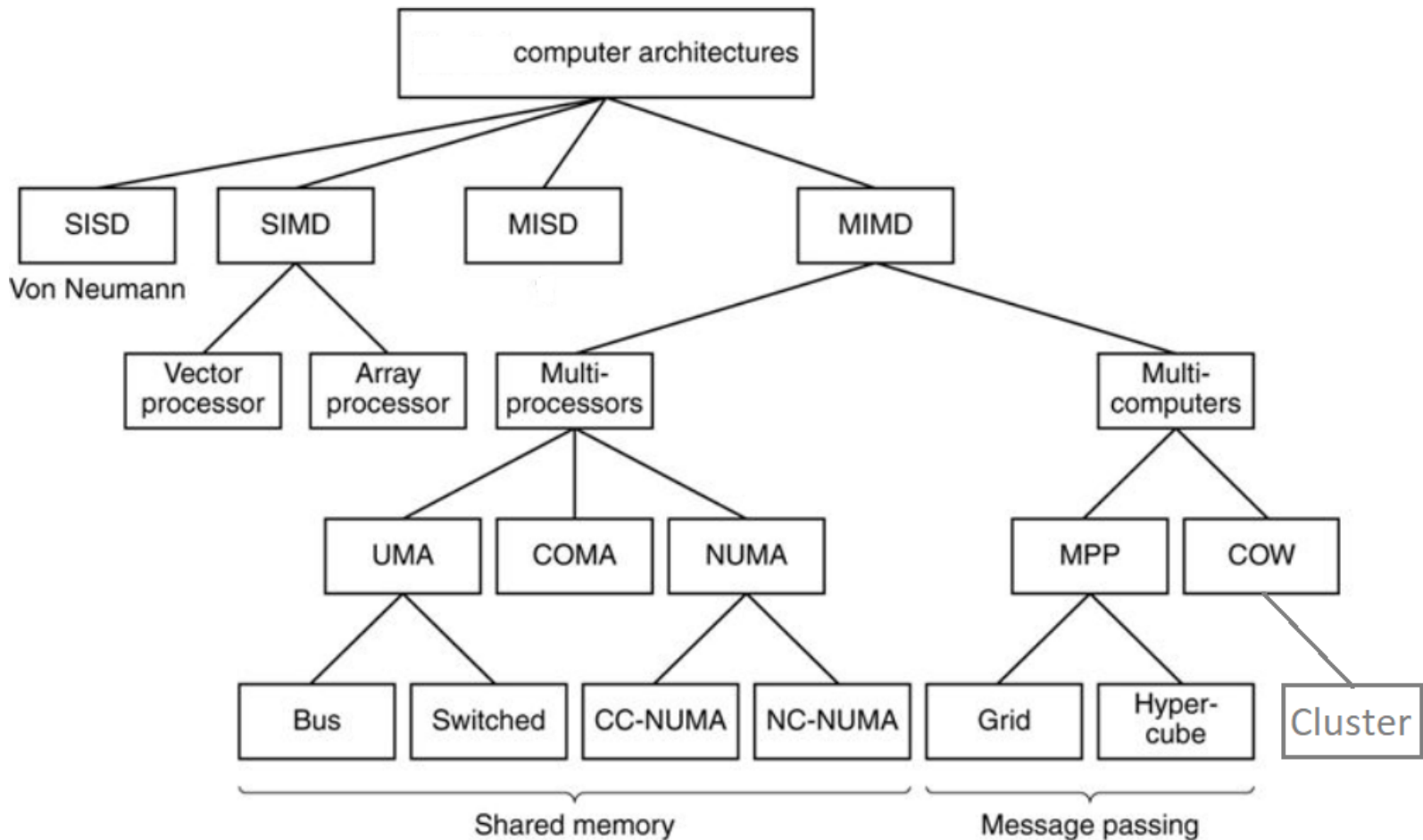

Taxonomii

Modele de calcul Paralel

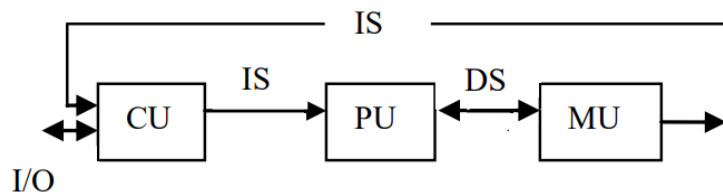
Arhitecturi de calculatoare



Taxonomii

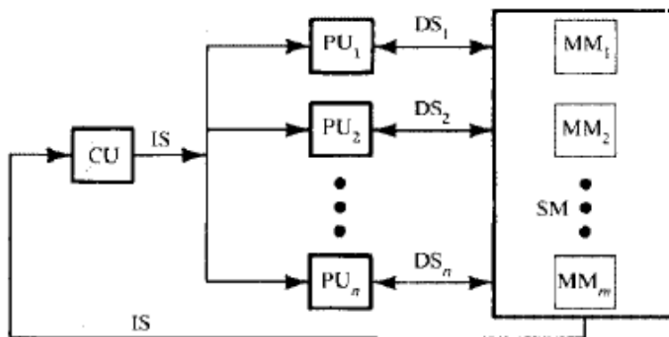
1. *Taxonomia lui Flynn (bazata pe relatia dintre fluxul de instructiuni si fluxul de date:*

- SISD (single instruction single data flow)
- SIMD (single instruction multiple data flow)
- MISD (multiple instruction single data flow)
- MIMD (multiple instruction multiple data flow)

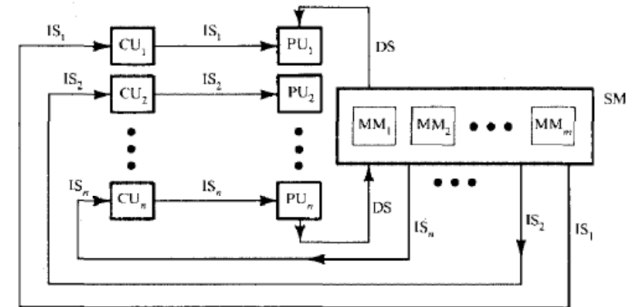


(a) SISD Uniprocessor Architecture

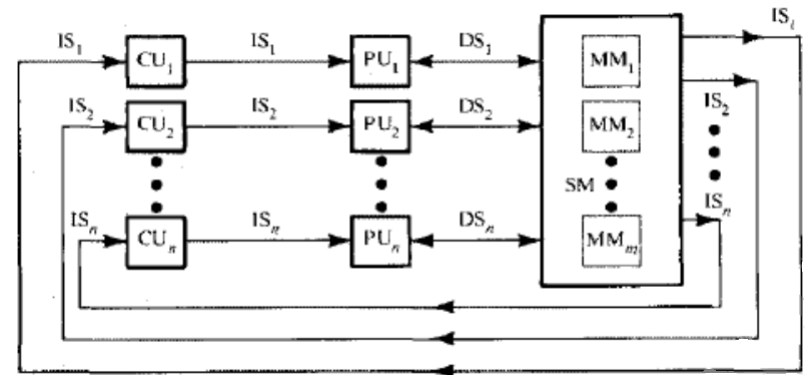
(b) SIMD Architecture



(c) MISD Architecture (the Systolic Array)



(d) MIMD architecture



Taxonomii

Taxonomia lui Shore:

- masina seriala pe cuvant paralela pe biti
- masina paralela pe cuvant seriala pe biti
- masina ortogonală
- masina masiv neconectat
- masina masiv conectat



Structurala

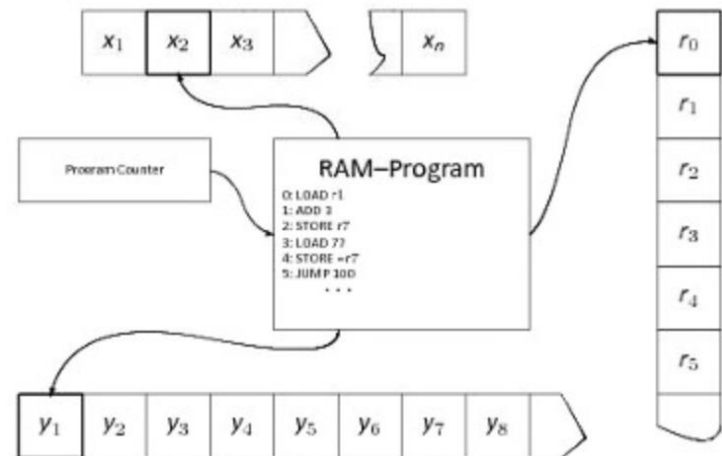
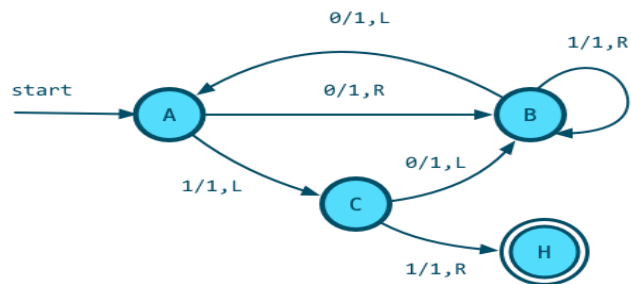
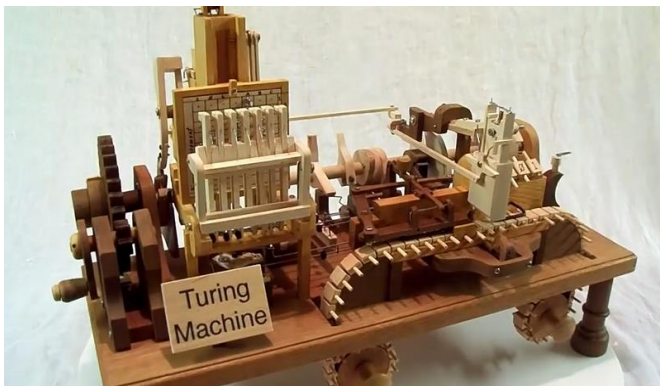
- uniprocsoare seriale
 - unicalculatoare paralele bazata pe paralelism functional si pipeline
 - masive de procsoare
-

Modele ale calculului paralel

- 1. Model RAM (Random Acces Machine)
 2. PIPELINE
 3. PROCESOARE DE VECTORI
 4. MULTIPROCESOARE cu memorie partajata
 5. MULTIPROCESOARE cu transfer prin mesaje
 6. PRAM (Parallel Random Acces Machine)
 7. PROCESARE SISTOLICA
 8. PROCESARE DATA FLOW

O mașină cu acces aleatoriu (RAM):

- o bandă de intrare infinită X cu poziția capului citit $i \in \mathbb{N}$,
- o bandă de ieșire infinită Y cu poziția capului de scriere $o \in \mathbb{N}$,
- un registru contor de program PC
- un program format dintr-un număr finit de instrucțiuni $P[1], \dots, P[m]$,
- o memorie constând dintr-o succesiune infinită de registre r_0, r_1, r_2



$RAM = (X, Y, S, s_0, f, Z)$

S multimea finită a stărilor automatului;
 s_0 starea inițială;

X alfabetul finit al benzii de intrare;

Y alfabetul finit al benzii de ieșire;

f funcția de tranziție, eventual parțial definită

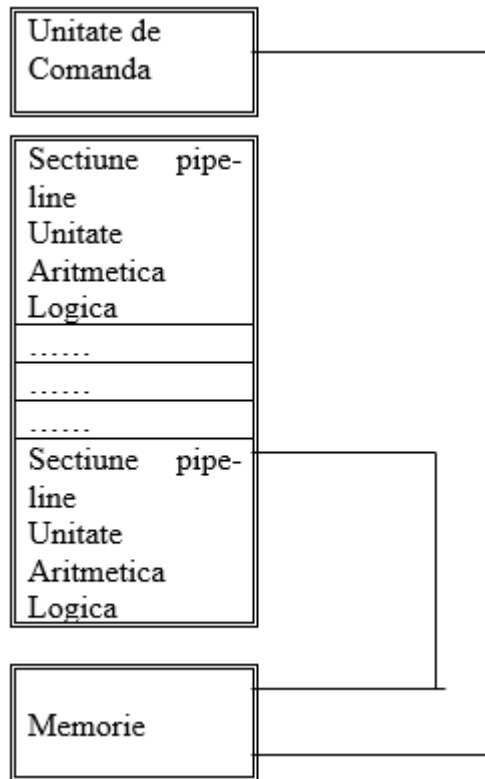
$f : S \times X \dashrightarrow S \times Y$

Z inclus în S - mulțimea stărilor finale

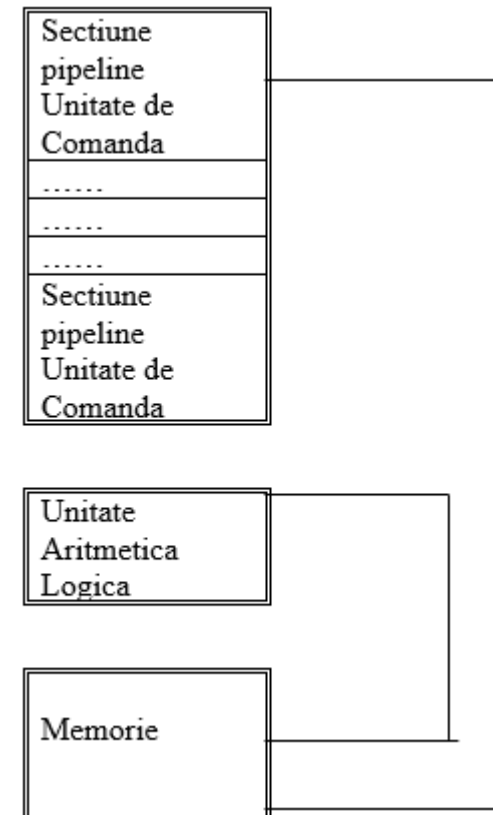
Structura pipeline.

Conceptul de pipeline consta in a imparti un task T in subtask-uri T_1, T_2, \dots, T_k si de a le atribui unor elemente de procesare.
Paralelismul se poate realiza fie la nivel de:

* prelucrare aritmetica;



citire, interpretare, executie instructiuni

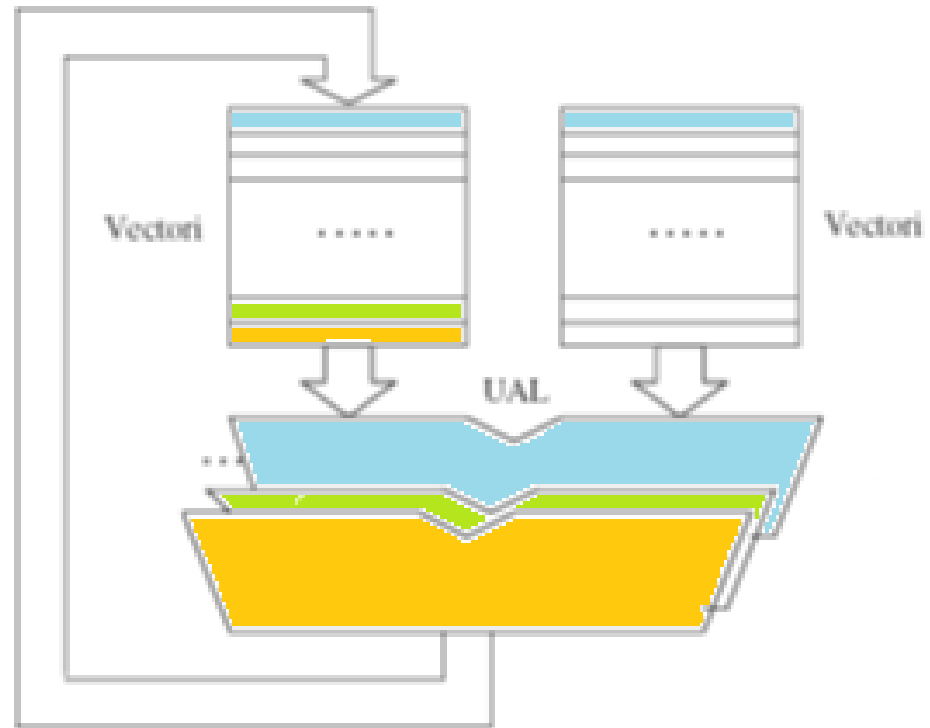


Procesoare de vectori.

In cadrul acestui model, exista un set de instructiuni care trateaza vectorul ca operand simplu. (SIMD) sau elemente de procesare vectoriala. (UAL de tip pipeline si registre de vectori).

O astfel de structura o gasim in sisteme monoprocesor care au ca extensie procesare de vectori.

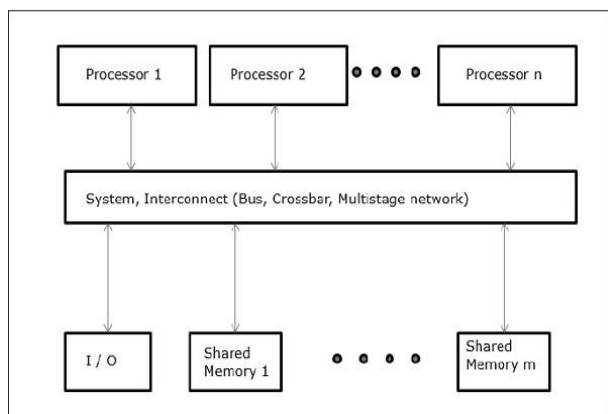
Acopera o gama redusa de probleme cu caracter vectorial.



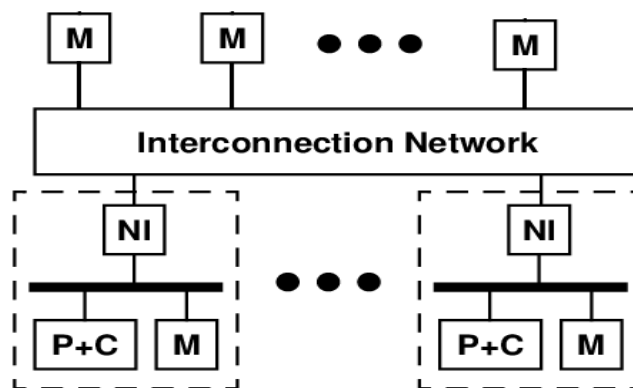
Multiprocesoare cu memorie partajata.

- Acest model descrie structurile in care fiecare procesor contine propria unitate de prelucrare, propria memorie si propria unitate de comanda si comunica prin intermediul unei retele de comutare cu module de memorie partajate.
- Fiecare procesor executa propriul set de instructiuni din memoria locala sau memoria partajata.
- Reteaua de comutare permite schimbul de informatii intre procesoare si intre procesoare si memoria partajata.

UMA



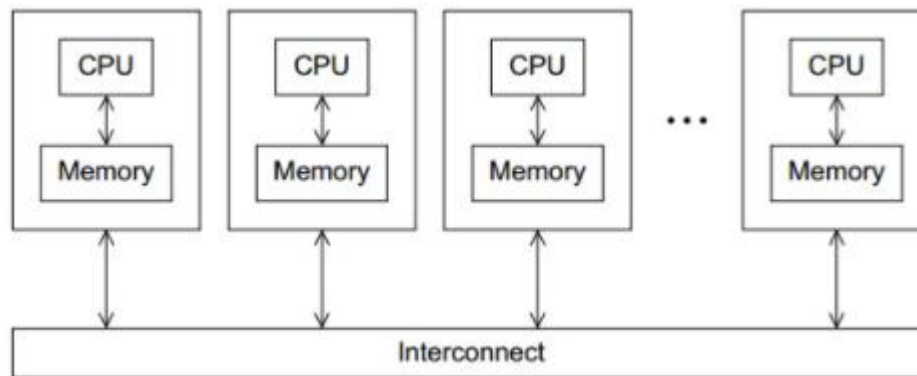
NUMA



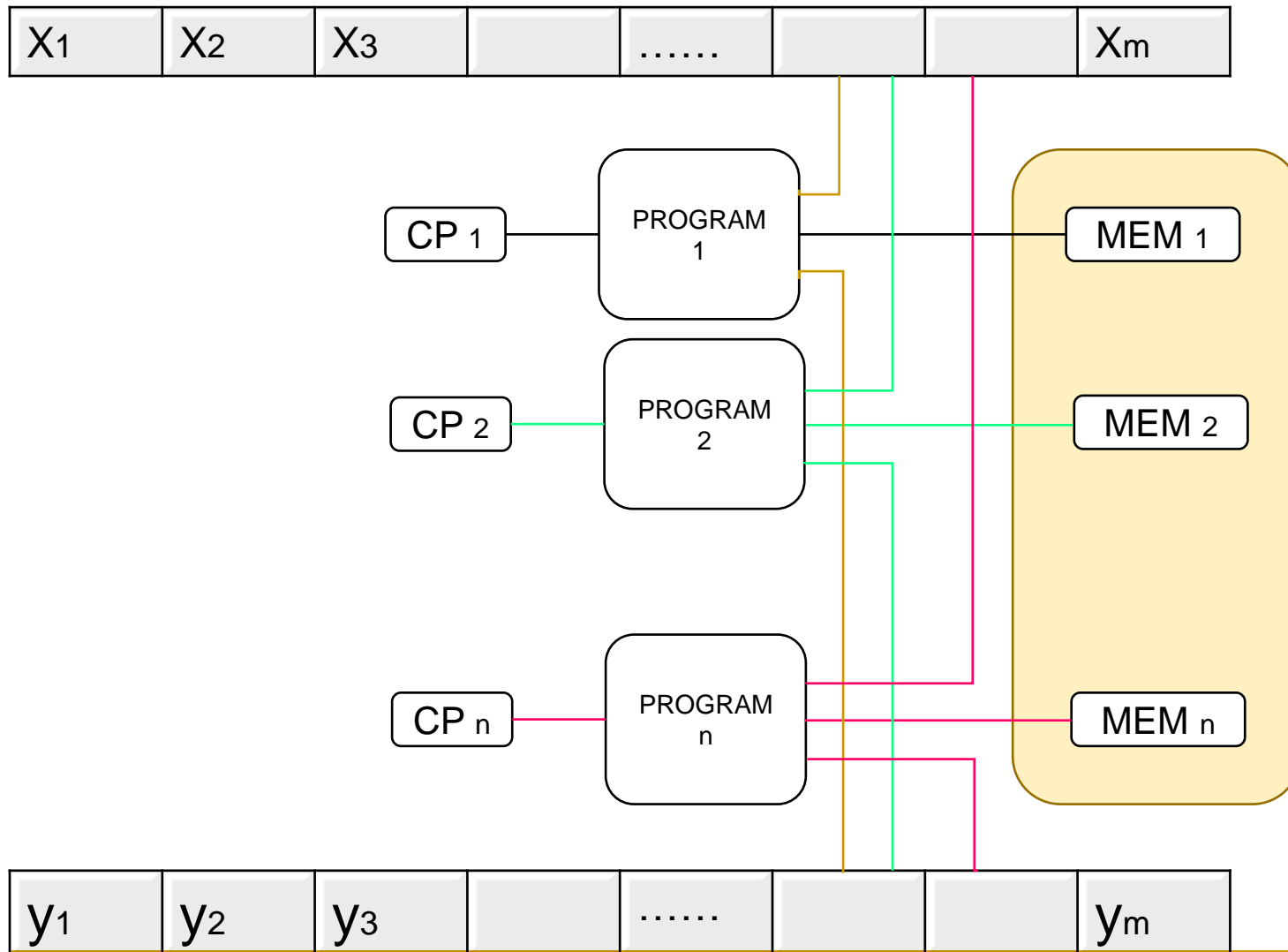
Multiprocesoare cu transfer prin mesaje

Multiprocesoare cu transfer prin mesaje.

- In acest model memoria este distribuita fiecarui procesor astfel incit fiecare procesor dispune de propriul program si propria memorie de date.
- Comunicarea datelor partajate este realizata prin schimburi de mesaje prin intermediul unei retele de comutare mesaje.
- Acest model este diferit de modelul cu memorie partajata deoarece interconexiunea se realizeaza prin mesaje si nu prin acces direct (comutare de circuite).



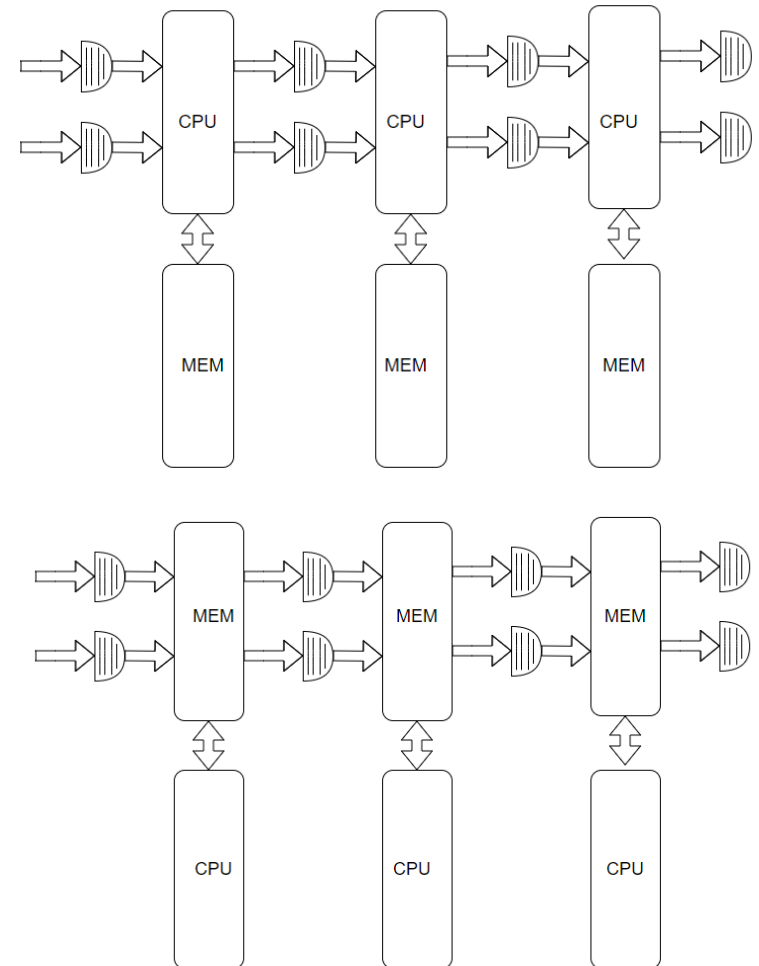
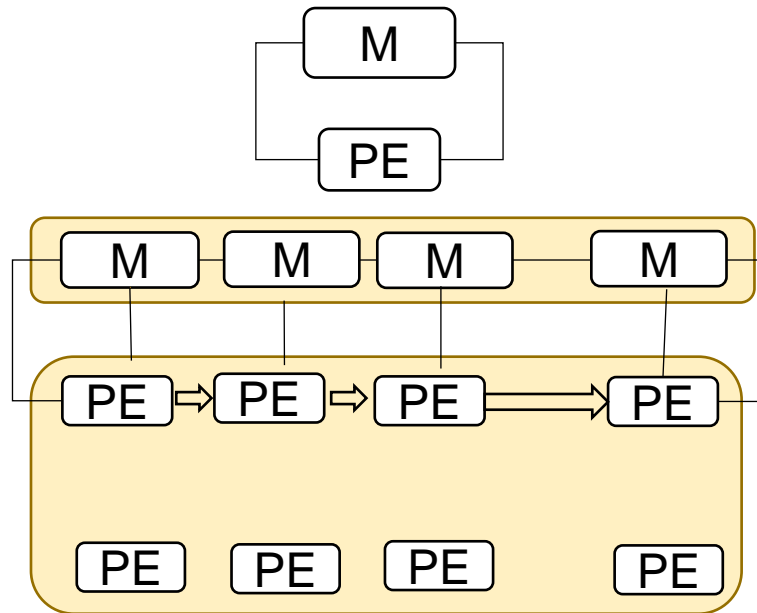
Modelul PRAM -parallel RAM



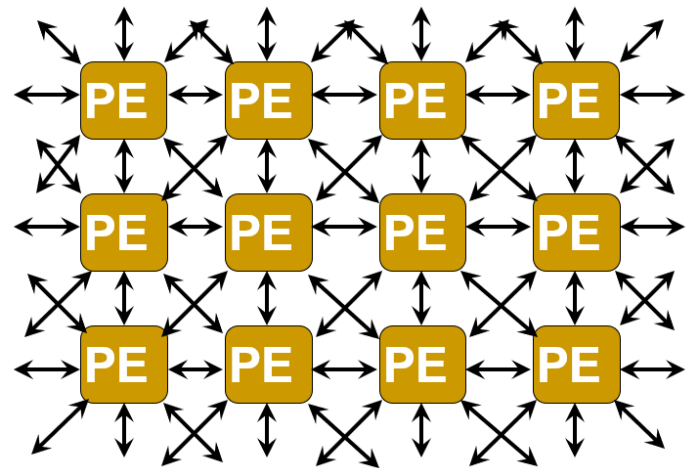
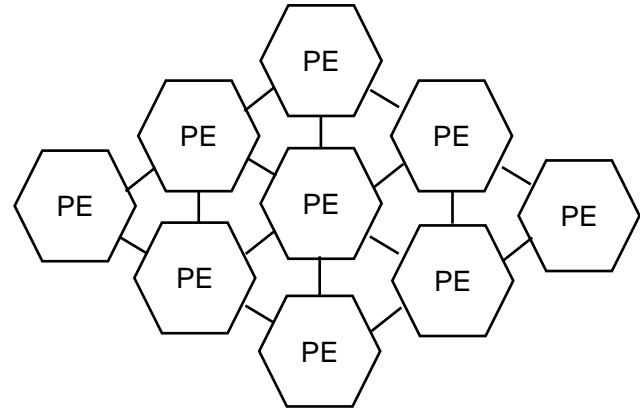
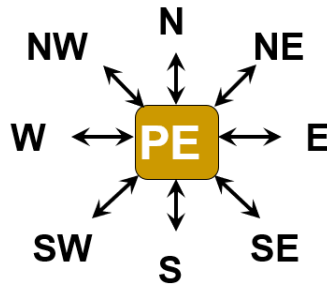
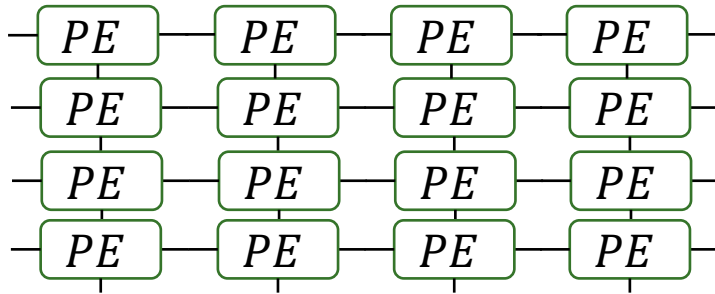
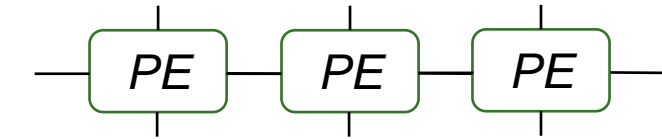
Procesare sistolica

Modelul procesarii sistolice consta din elemente de procesare identice aranjate intr-o structura **pipeline**.

Procesarea sistolica este caracterizata prin **interconexiuni simple** si **regulate** ceea ce permite integrarea **VLSI**.



Modalitati de conexiune Sistolica



Exemplu

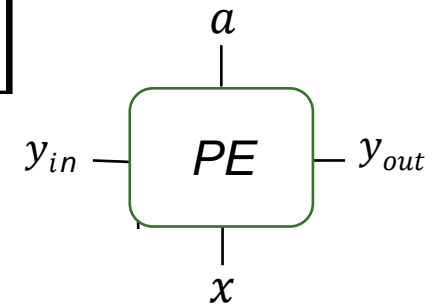
■ Fie

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \quad Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad Y_0 = \begin{bmatrix} y_{01} \\ y_{02} \\ \vdots \\ y_{0n} \end{bmatrix}$$

■ Dorim sa calculam

$$Y = A * X + Y_0$$

■ Presupunem ca avem un element de procesare sistolica

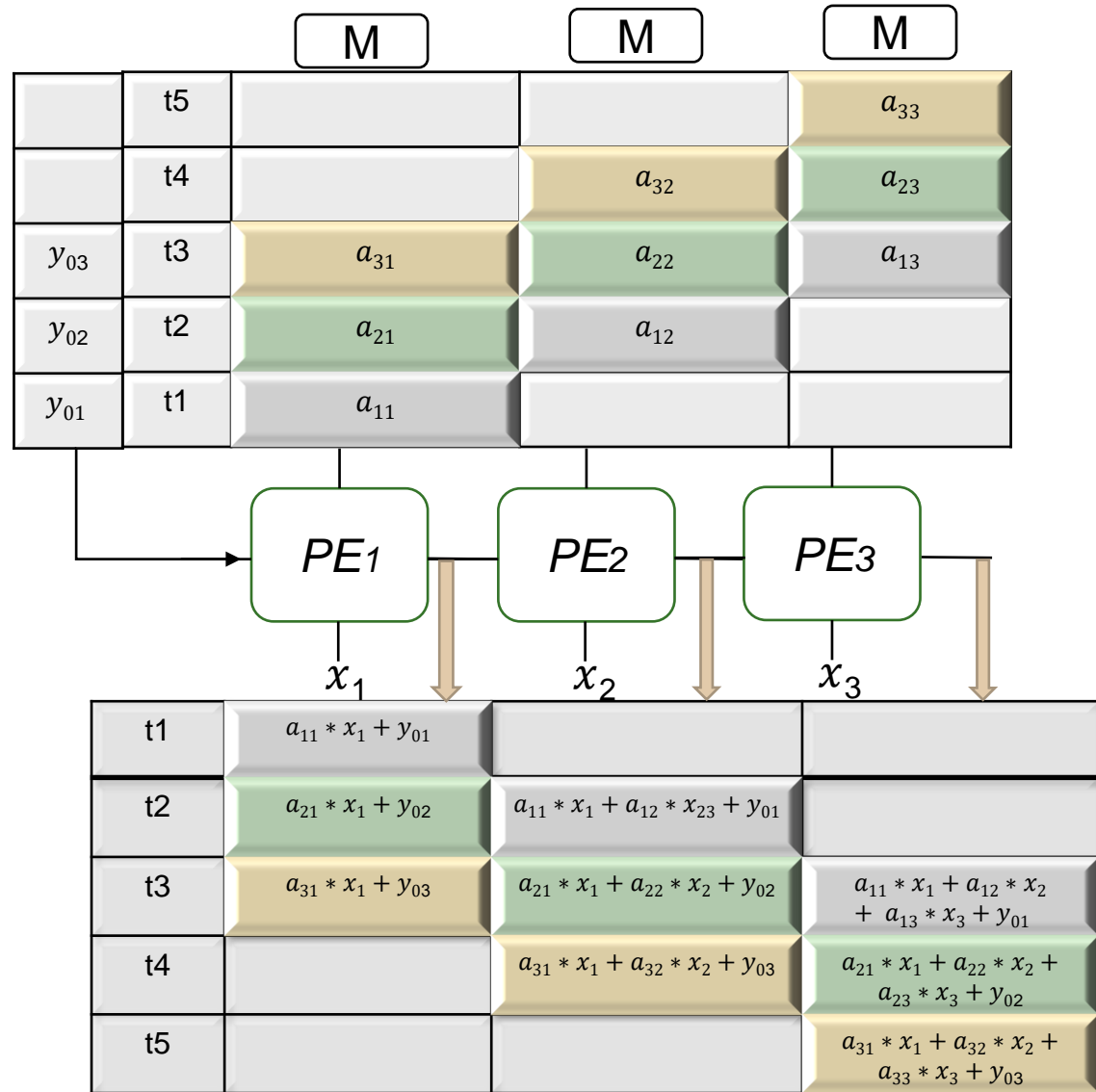
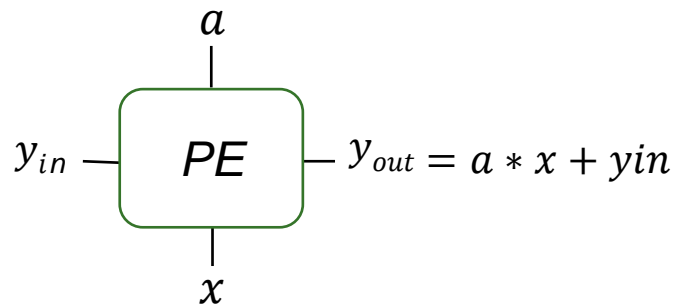


$$y_{out} = a * x + y_{in}$$

Y		A					X		Y ₀	
y ₁		a ₁₁	a ₁₂	a ₁₃		a _{1n}	x ₁		y ₀₁	
y ₂		a ₂₁	a ₂₂	a ₂₃		a _{2n}	x ₂		y ₀₂	
y ₃		a ₃₁	a ₃₂	a ₃₃		a _{3n}	x ₃		y ₀₃	
y _n		a _{n1}	a _{n2}	a _{n3}		a _{nn}	x _n		y _{0n}	

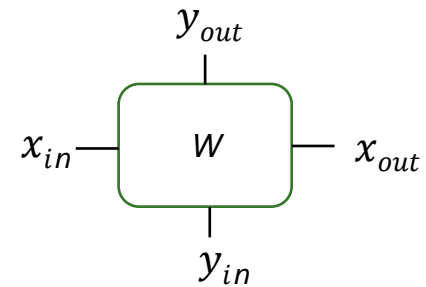
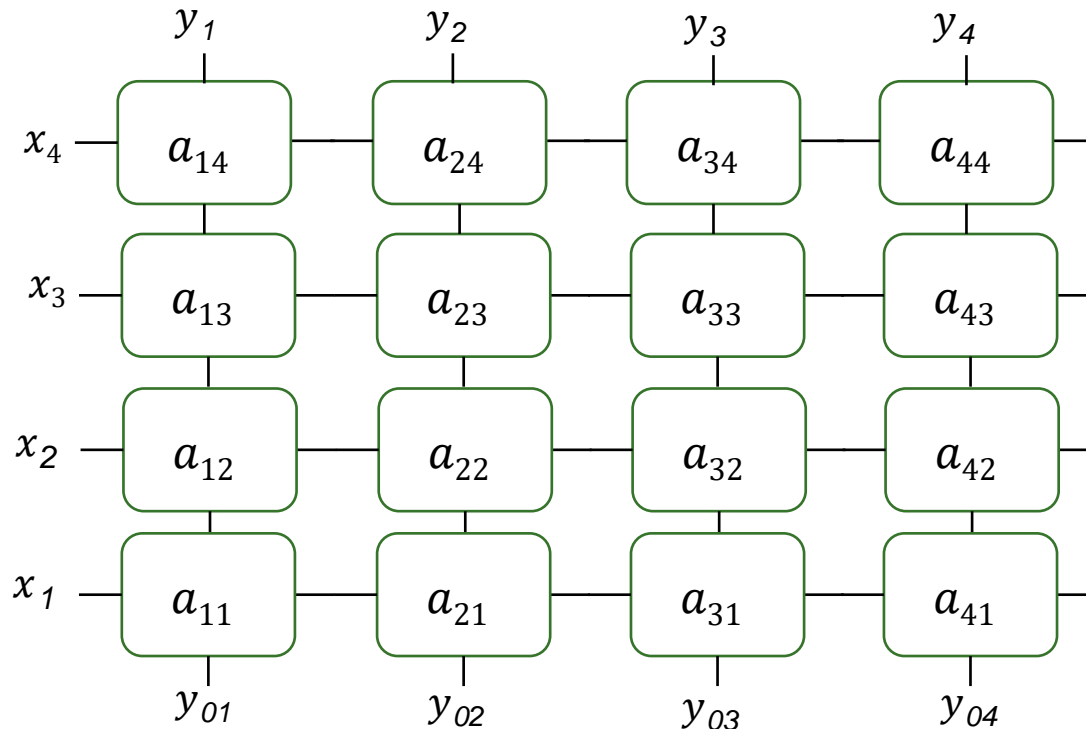
Exemplu pentru $n=3$, structura unidimensională

- $y_i = \sum_{j=1}^n a_{ij} * x_j + y_{0i}$
- $y_1 = a_{11} * x_1 + a_{12} * x_2 + a_{13} * x_3 + y_{01}$
- $y_2 = a_{21} * x_1 + a_{22} * x_2 + a_{23} * x_3 + y_{02}$
- $y_3 = a_{31} * x_1 + a_{32} * x_2 + a_{33} * x_3 + y_{03}$



Exemplu pentru $n=4$, structura bidimensionala

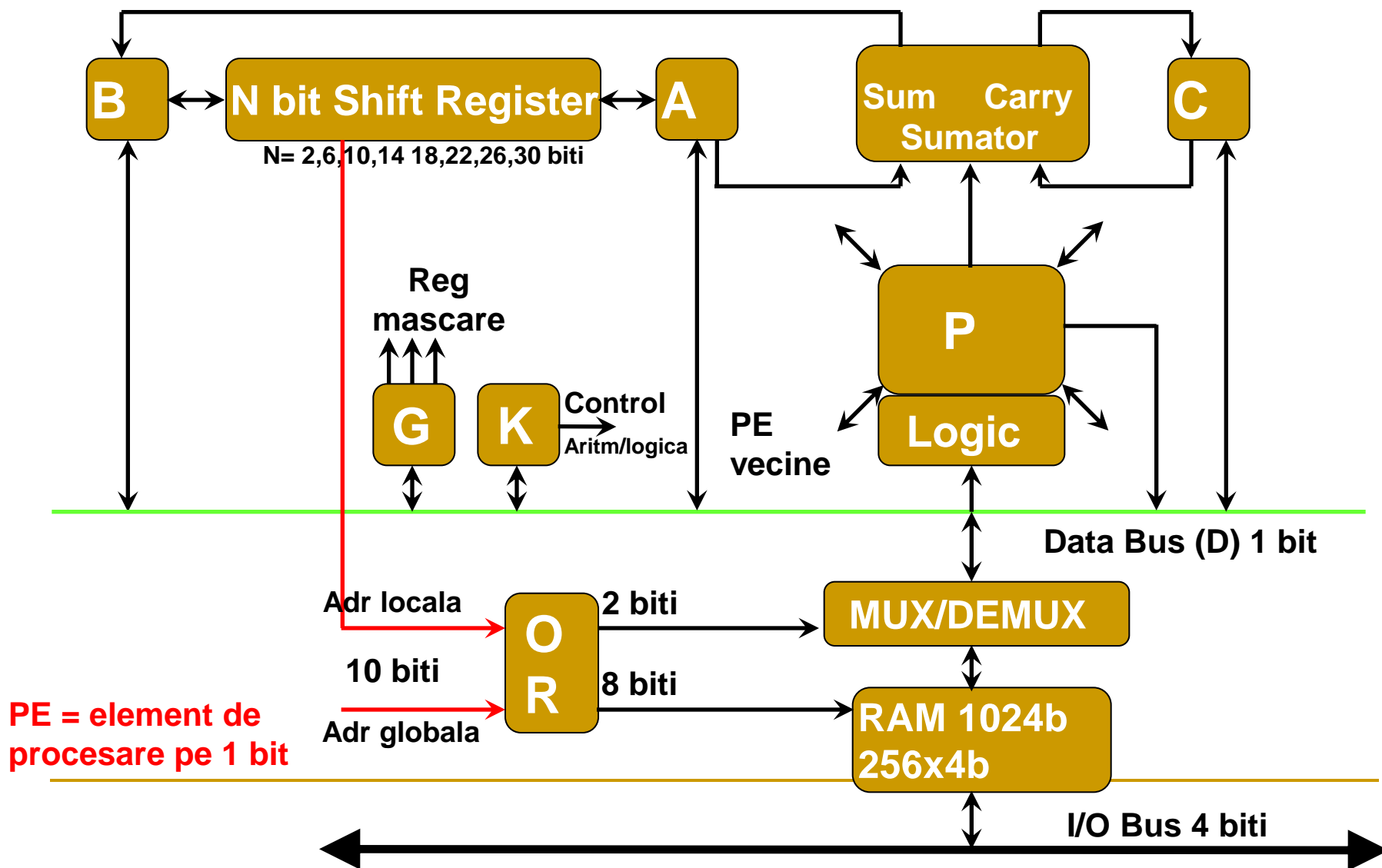
- $y_i = \sum_{j=1}^n a_{ij} * x_j + y_{0i}$
- $y_1 = a_{11} * x_1 + a_{12} * x_2 + a_{13} * x_3 + a_{14} * x_4 + y_{01}$
- $y_2 = a_{21} * x_1 + a_{22} * x_2 + a_{23} * x_3 + a_{24} * x_4 + y_{02}$
- $y_3 = a_{31} * x_1 + a_{32} * x_2 + a_{33} * x_3 + a_{34} * x_4 + y_{03}$
- $y_4 = a_{41} * x_1 + a_{42} * x_2 + a_{43} * x_3 + a_{44} * x_4 + y_{04}$



$$y_{out} = w * x_{in} + y_{in}$$

$$x_{out} = x_{in}$$

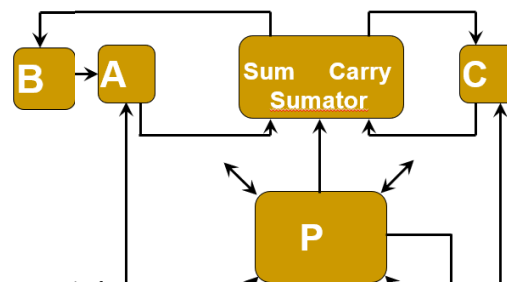
Exemplu de masina sistolica, Arhitectura Masinii Blitzen – PE



Elementele PE

- P – registru pe 1 bit
 - Operanzi
 - Operatori in functiile aritmetice
 - Utilizat la rutare: primeste ops de la PE-uri adiacente
- K – registru de control aritmetico-logic
 - Faciliteaza aparitia ops aritmetice inverse (+/-)
 - Algoritmi de impartiri (non-returning-division)
- G – registru de mascare
 - Se indica daca operatia se executa sau nu
- A & B – registru pe 1 bit
 - Preiau rezultatele de la sumator
 - Contribuie la suma

Elementele PE

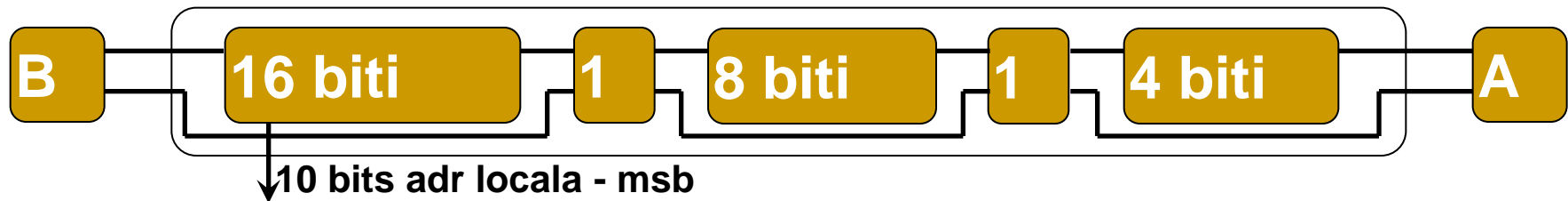


A	P	C	B	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

■ Sumator

- ADD: $A + P + C \rightarrow B, C$
- HADD: $A + C \rightarrow B, C$

■ Shift Register (SR) – Registru de deplasare



- N stagii = 2, 6, 10, 14, 18, 22, 26, 30
- 30 biti + A & B = 32 biti
- Bitul obtinut prin deplasare nu e neaparat salvat
- La fiecare pas SR se deplaseaza cu o pozitie R/L
- SR-ul poate fi sters

Memoria & I/O Bus

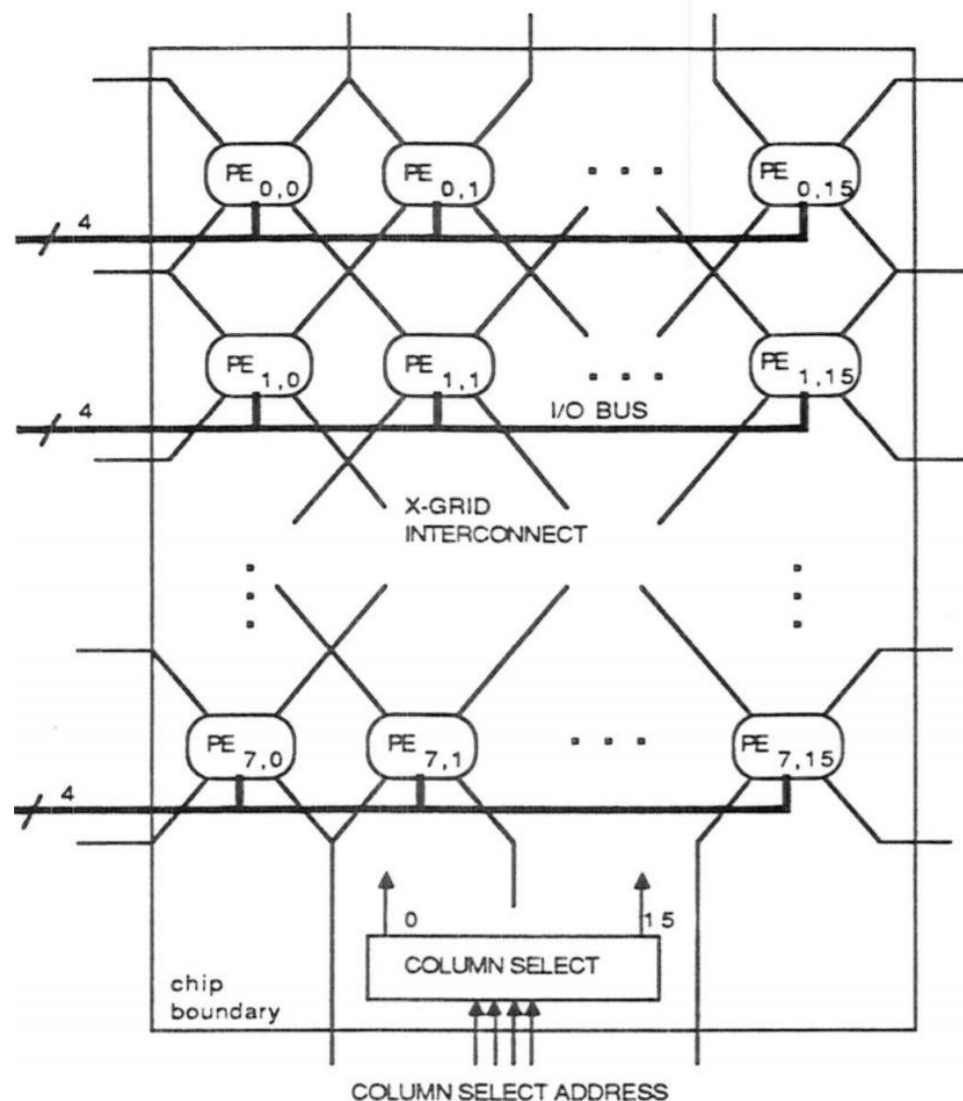
- Accesul la memoria RAM
 - Cu o adresa globala de forma 2^p (p e multiplu de 2)
 - Cu o adresa locala → specific masinii Blitzen (10 biti)
- I/O Bus
 - Formata din 4 biti si utilizata pentru conectarea PE-urilor la resurse externe de stocare a datelor
 - O magistrala I/O este folosita in comun de catre 16 PE-uri



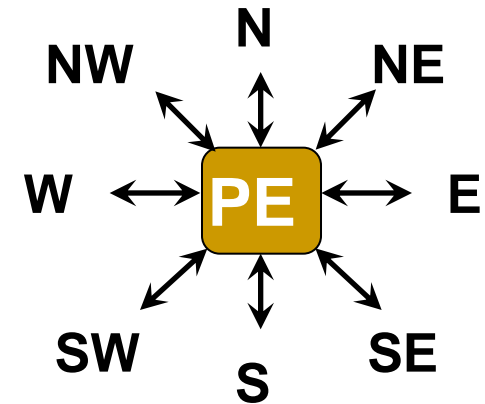
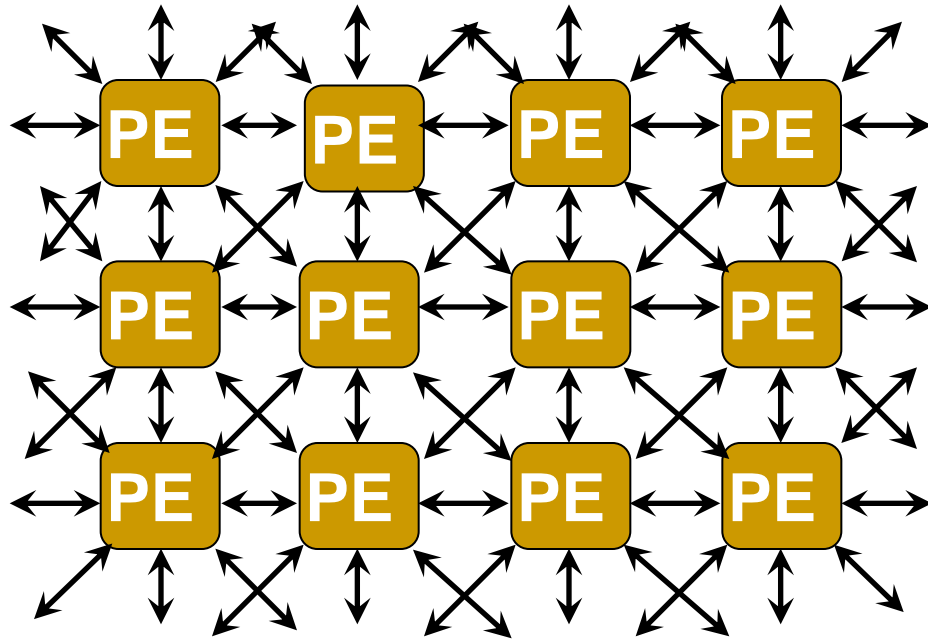
Chip-ul Blitzzen

Un Chip Biltzen este format din

- 8 magistrale
- 8 linii a cate 16 procesoare fiecare → 128 PE/chip, fiecare cu cate 1K de RAM
- Fiecare PE functioneaza la 20MHz cu $8 \times 4 = 32$ biti/ciclu



Un Sistem Blitzen

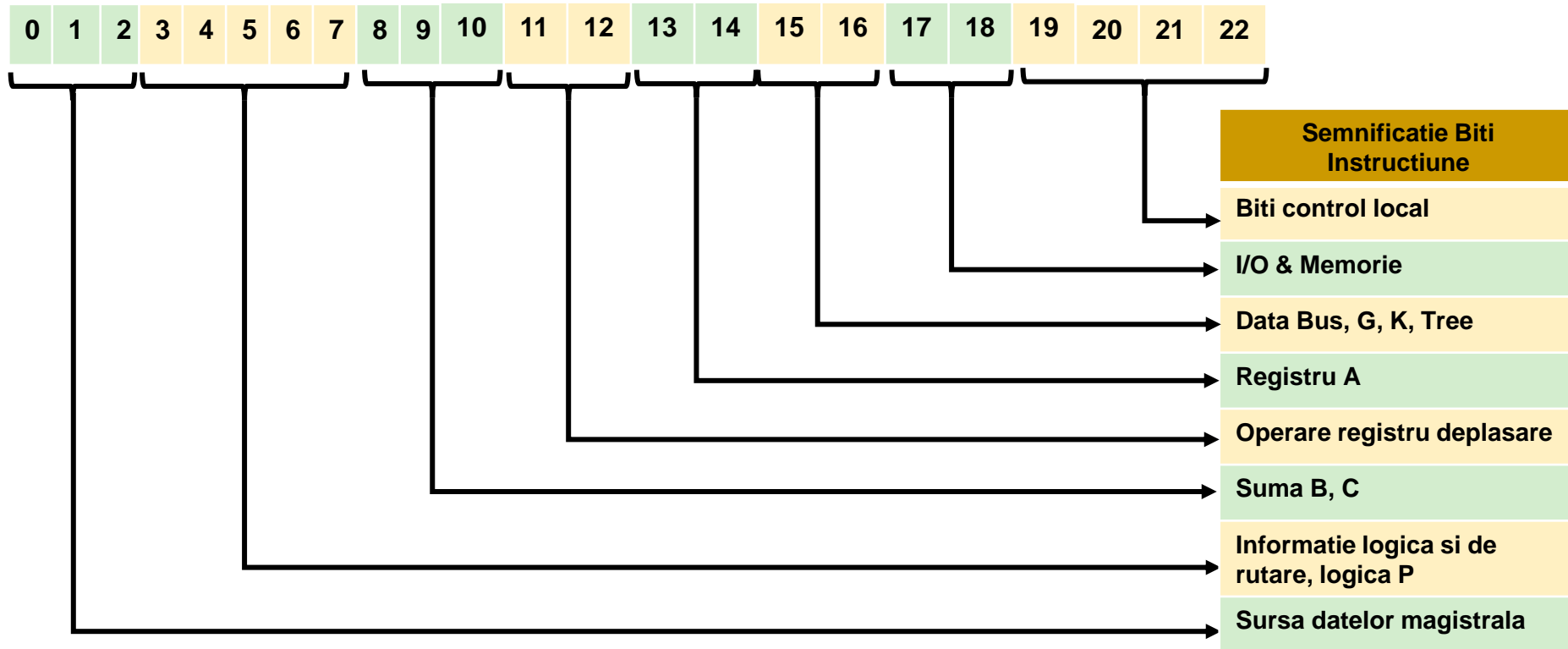


Structura X-Grid

- Sunt 8 directii de rutare
 - N, S, E, W, NE, SE, SW, NW
- Chip-urile Blitzen se grupeaza pe linii de 128 chip-uri → 128 x 128 PE = 16384PE (sistem Blitzen)

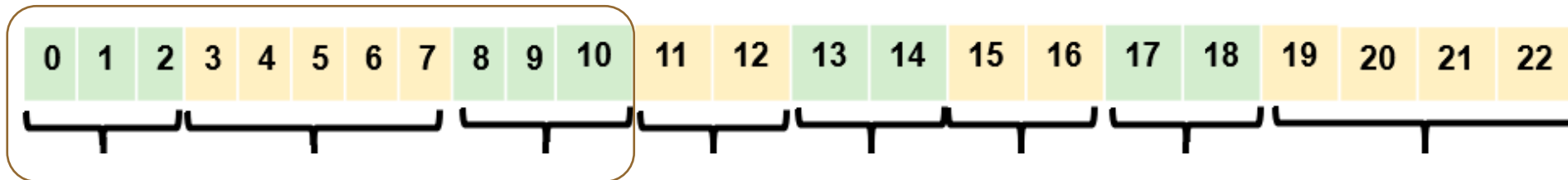
Formatul Instructiunilor

- La fiecare ciclu se executa o instructiune intr-un pipeline cu 3 stagii: Decodare, Broadcast, Executie
- Instructiunile sunt pe 23 de biti organizati astfel:



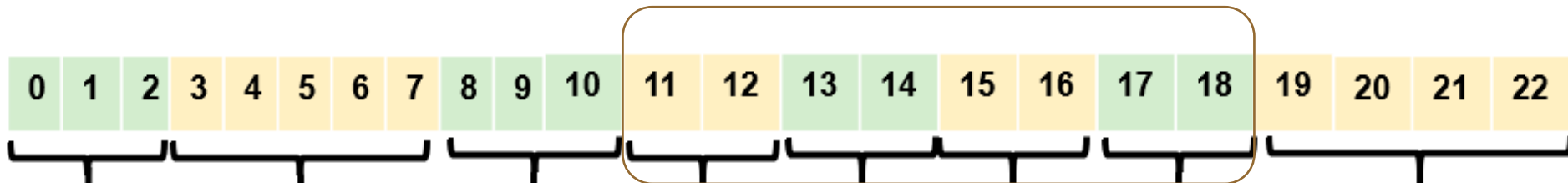
Semnificatia Bitilor Instructiune

- Bitii 0, 1, 2: sursa datelor de pe magistrala
 - ❑ Registrele A, B, C
 - ❑ Registrele G sau K
 - ❑ Registrul P
 - ❑ Memorie (cand este adresata memoria se executa un read 1b)
- Bitii 3, 4, 5, 6, 7:
 - ❑ Operatia logica ce trebuie operata in P
 - ❑ Operatia de rutare catre unul din cei 8 vecini
 - ❑ Configurarea registrului de deplasare (no assigns to P)
- Bitii 8, 9, 10: operatii executate de sumator
 - ❑ Suma & Carry din sumator catre registrele B si C
 - ❑ Pot fi utilizate si pentru incarcarea lui B & C



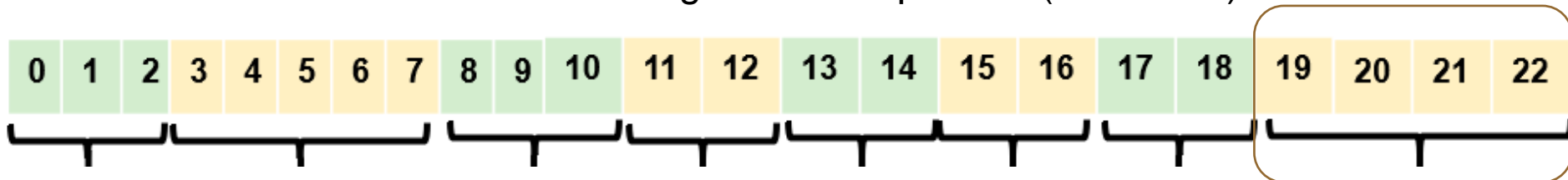
Semnificatia Bitilor Instructiune

- Bitii 11,12: operatii de deplasare/shiftare
 - Stanga (L)/Dreapta (R)/Stergere (D)
- Bitii 13,14: operatii cu registrul A
 - Setare directa
 - Setare cu continului registrului de deplasare/shiftare
- Bitii 15,16: rezultatul e transferat din magistrala de date
 - Catre registrele G, K
 - Catre OR-ul ierarhic (Tree)
- Bitii 17,18: prelucrarea informatiilor din
 - Memoria locala (R/W operations)
 - I/O Bus (I/O operations)



Semnificatia Bitilor Instructiune

- Bitii 19,20,21,22: control local
- Bit 19: operatii mascate & nemascate ale registrului P
 - Bit 19 = 0: operatiile P nemascate
 - Bit 19 = 1: operatiile P mascate;
- Bit 20 : identifica operatii mascate efectuate de memorie, sumator, SR, & registrii A, B sau C
 - Bit 20 = 0: operatiile nemascate
 - Bit 20 = 1: operatiile mascate;
- Bit 21 : poate identifica daca operatiile sunt transmise sau daca a avut loc un broadcast. Este folosit pt:
 - Bit 21 = 0: operatiile curente
 - Bit 21 = 1: op complementate
- Bit 22 : operatii cu
 - Bit 22 = 0: adresa globala
 - Bit 22 = 1: cu cei 10 biti din registrul de deplasare (adr locala)



Particularitati Blitzen

- PE-urile lucreaza la nivel de bit
 - Registrul de Shiftare/Deplasare este bidirectional
 - Blitzen-ul are memorie on-chip de 1k (1024 b)
 - Este posibila mascarea separata a diverselor tipuri de operatii cu memoria sau cu SR (de deplasare)
 - Exista un control local al adresei de memorie (Local Address Modifier)
 - Bitul K de la Blitzen permite o executie a datelor de tip MIMD
 - Masina Blitzen in anumite aplicatii este mult mai performanta ca MPP
-

Exemple

- Mai multe instructiuni elementare distincte se pot executa in acelasi timp:
 - ADD + preluare date din memorie + preluare de pe magistrala de date
- Exemple de microinstructiuni:
 - SET_C // $C \leftarrow 1$
 - ADD // $A + P + C \rightarrow B, C$
 - MOV_BD // pune B pe magistrala de date D
 - MOV_MD(ADDR) // pune de la ADDR pe magistrala D
 - MOV_DP // pune in P continutul de pe D
 - ROUTE_E // trimite P catre E si incarca in P din W

Adunarea a doua Numere (8b)

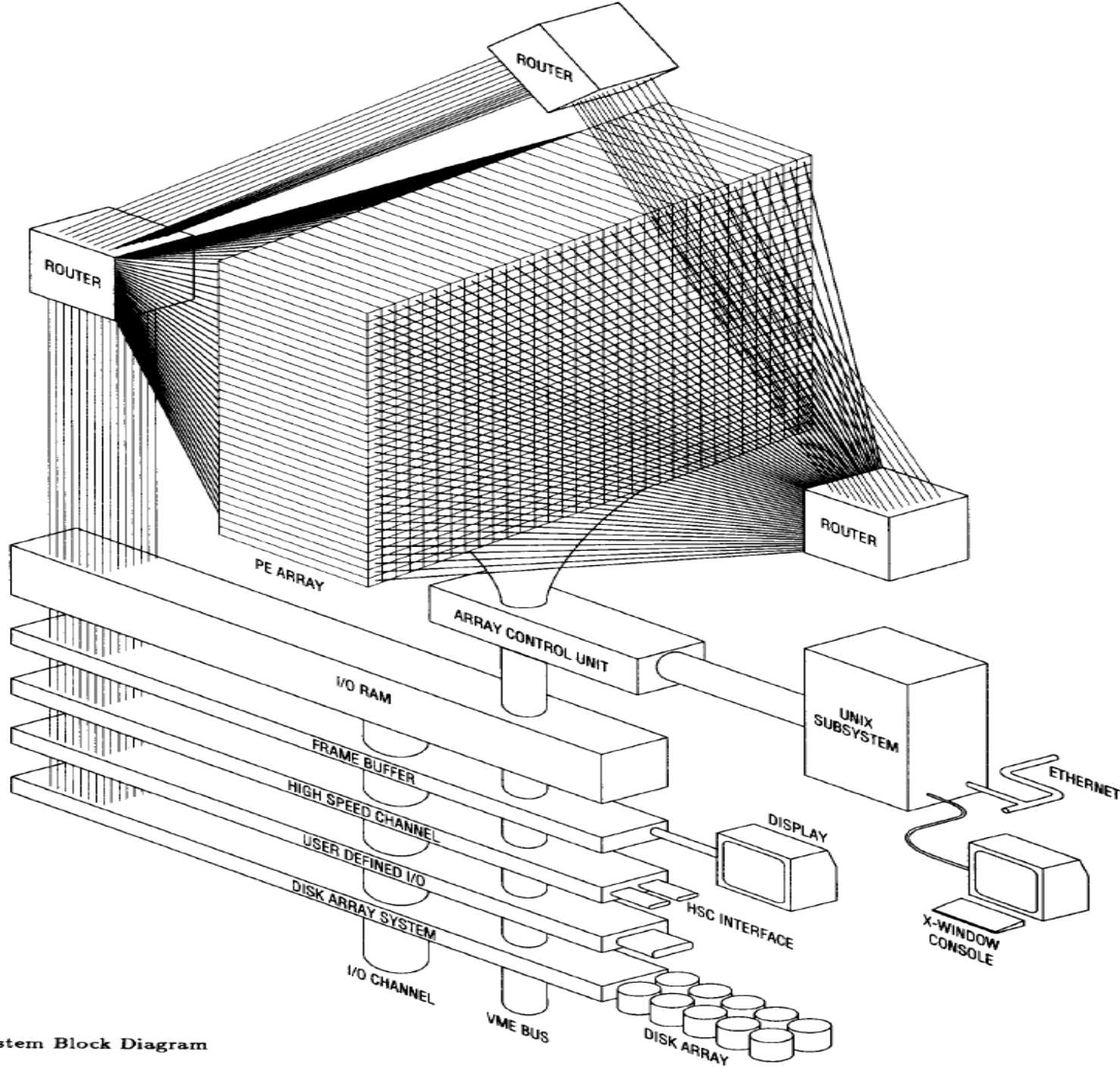
```
#include "blitzen.h"          /*contine setul de instructiuni */
#define XADDR    100
#define YADDR    200
#define WADDR    300
#define NUMBITS  8
main () {
    // suma W = X + Y;
    // valori cum ar fi "X0" fac referinta la bitul 0 din X
    set_route(GRID);
    load_file("in1.ism",XADDR,XADDR,8);    //citesc X si Y
    load_file("in2.ism",YADDR,YADDR,8);
    CLR_C;                                // clear C,
    for (int i=0; i< NUMBITS; i++){
        MOV_MD(XADDR+i);  /* A <- X(i), data de la adresa XADDR pe mag da date D
        MOV_DA;            // pune in A continutul de pe D
    END;
        MOV_MD(YADDR+i);  /* P <- Y(i)
        MOV_DP;
    END;
        ADD;                /* adunare, rezultat in B
    END;
        MOV_BD;              /* W(i) <- B
        MOV_DM(WADDR+i);
    END;

    }

    save_file("sum.osm",WADDR,WADDR,8);    //salvare rezultat
    zyg_end();
}
```

Categorii de Algoritmi

- Algoritmii ce se preteaza la utilizarea masinii Blitzen fac parte din urmatoarele categorii
- 1. Embarrassingly Parallel Algorithms (algoritmi trivial de paraleli)
 - Fiecare PE actioneaza independent (paralelism maxim);
 - Exemplu: Valoarea de prag (apartenenta la domeniu)
- 2. Near Embarrassingly Parallel Algorithms
 - Un PE are nevoie de informatii pe care le va primi de la alte PE
 - Exemplu: tratarea imaginilor (filtre etc); detectia contururilor



MP-1 System Block Diagram

Modelul Data Flow

- In Model bazat pe flux de date (data flow):
 - execuția unei instrucțiuni de prelucrare este efectuată doar de disponibilitatea operanzilor!
 - Nu dispune de PC
 - lipsesc cele două caracteristici ale modelului von Neumann care devin blocaje în exploatarea paralelismului
-

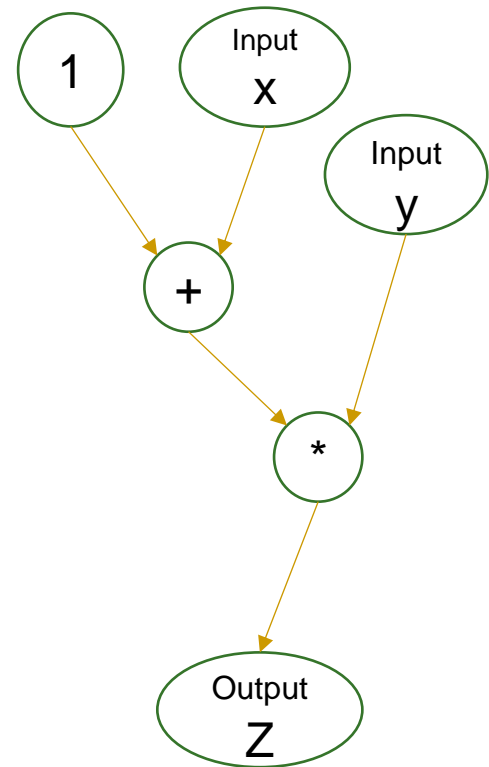
Regula de activare:

- O instrucțiune este activată (adică executabilă) dacă sunt disponibili toți operanzii.
- Observați că în modelul von Neumann, o instrucțiune este activată dacă este indicată de PC.
- Regula de calcul sau regula de declanșare, specifică când se execută efectiv o instrucțiune activată.
- O instrucțiune este declanșată (adică executată) când devine activată.
- Efectul declanșării unei instrucțiuni este prelucrarea datelor sale de intrare (operanzi) și generarea datelor de ieșire (rezultate).

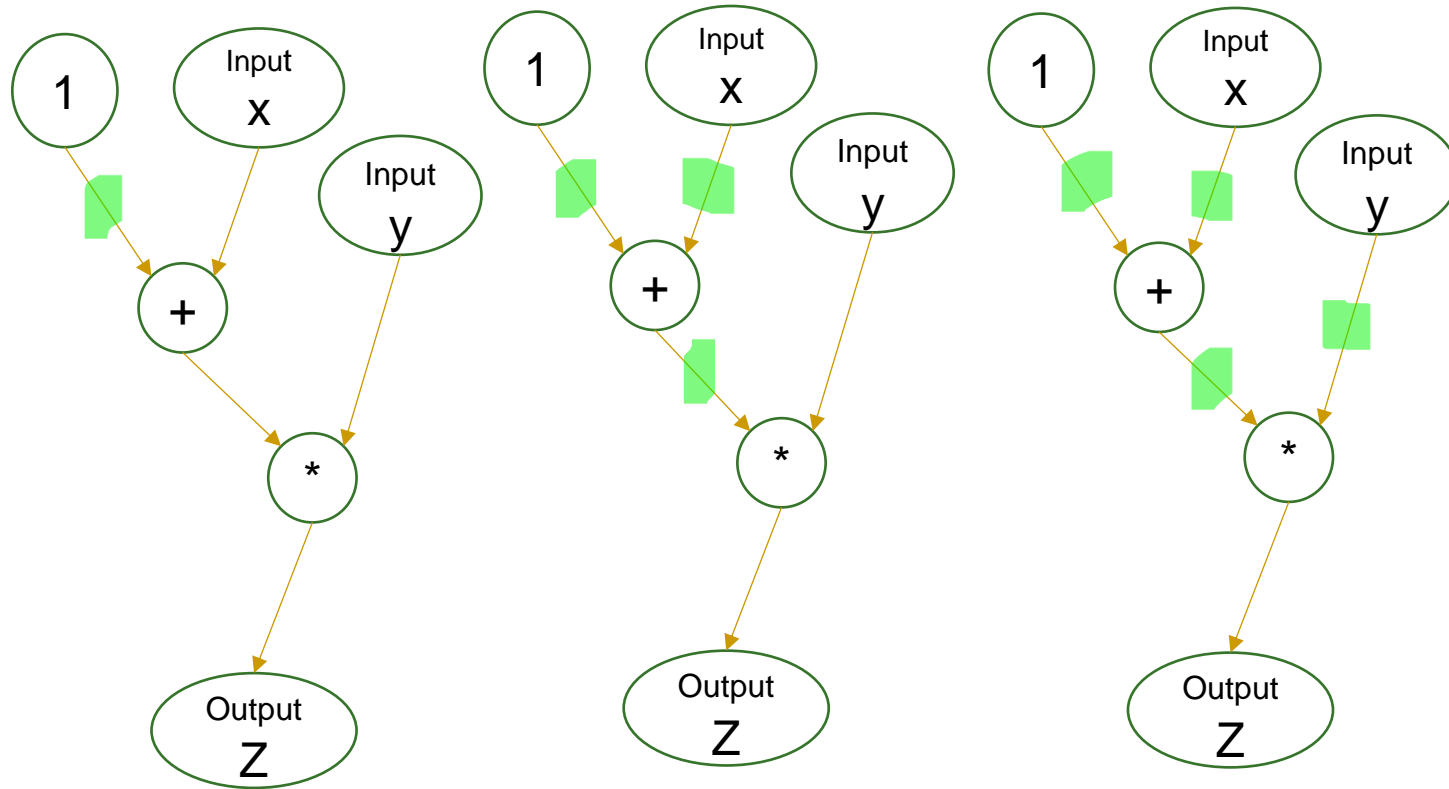
Exemplu

- Un program de flux de date este compilat într-un graf de flux de date care este un graf direcționat format din noduri, care reprezintă instrucțiuni și arce, care reprezintă dependențe de date între instrucțiuni.
- Grafiul fluxului de date este similar cu un graf de dependență utilizat în reprezentările intermediare ale compilatoarelor.
- În timpul executării programului, datele se propagă de-a lungul arcurilor în pachete de date, numite jetoane (token-uri).
- Acest flux de jetoane permite unele dintre noduri (instrucțiuni) să înceapă execuția și să le declanșează.

$$Z = (x+1) * y$$



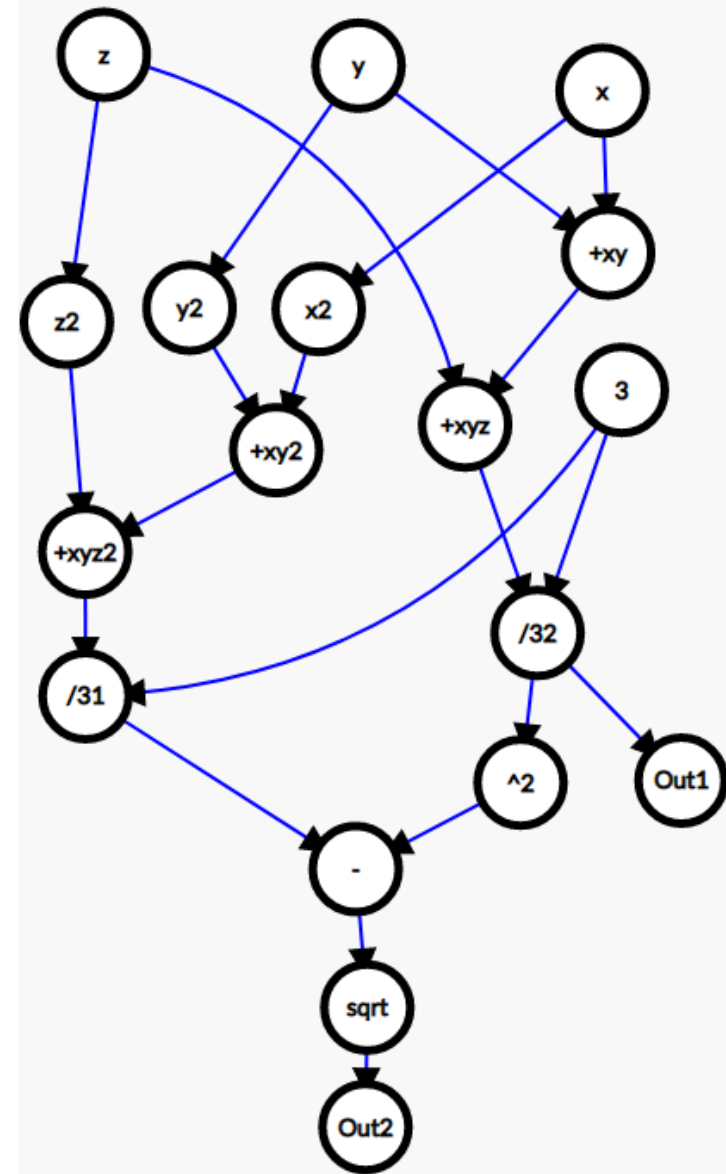
Executie



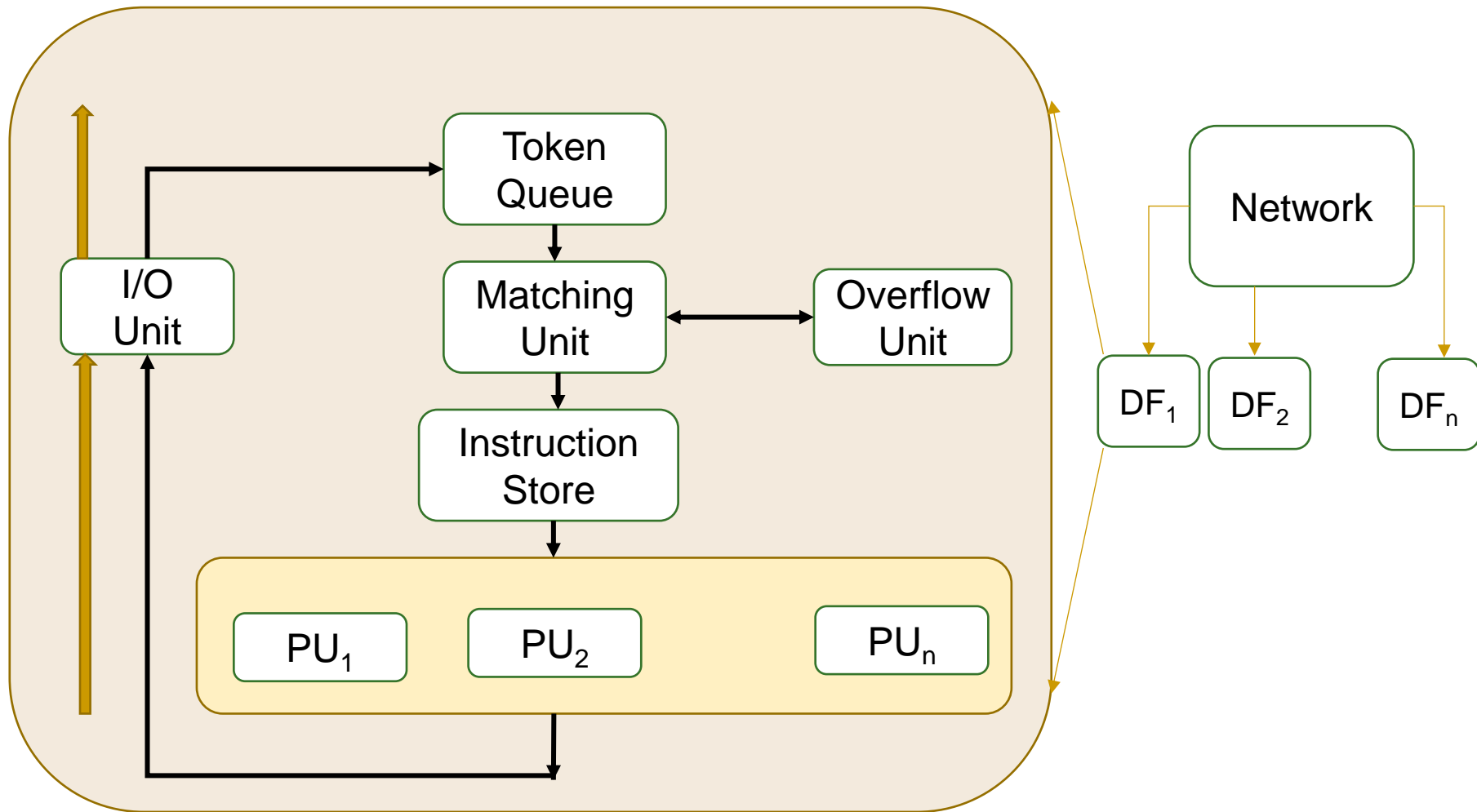
Exemplu

- function Med_Dev:
- Calculeaza media si deviatia standard a trei variabile

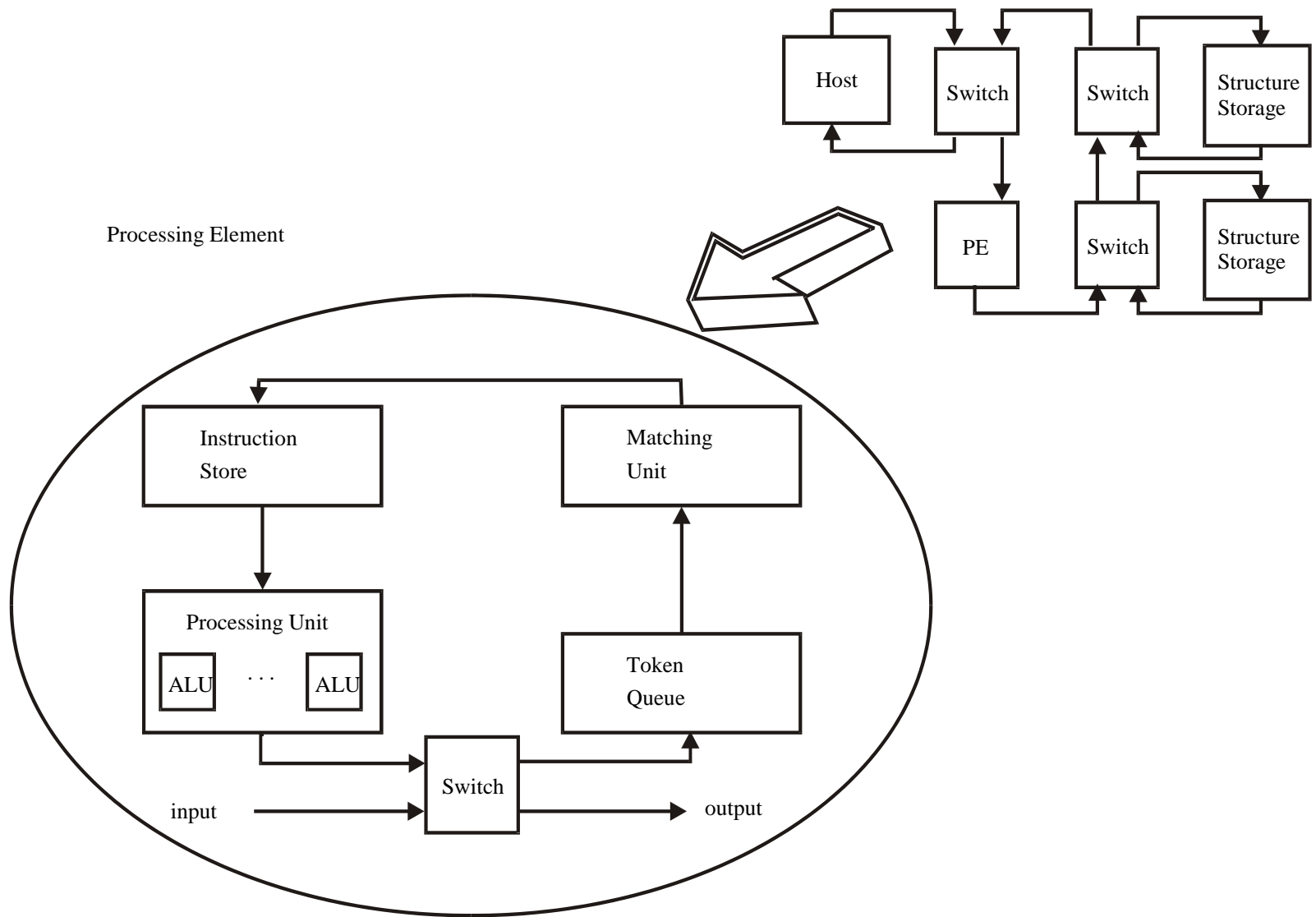
```
function Med_Dev(x,y,z: real returns  
    real,real);  
let  
    Out1 := (x + y + z)/3;  
    Out2 := SQRT((x**2 + y**2 + z**2)/3 - Out1**2);  
in  
    Out1, Out2  
endlet  
endfun
```



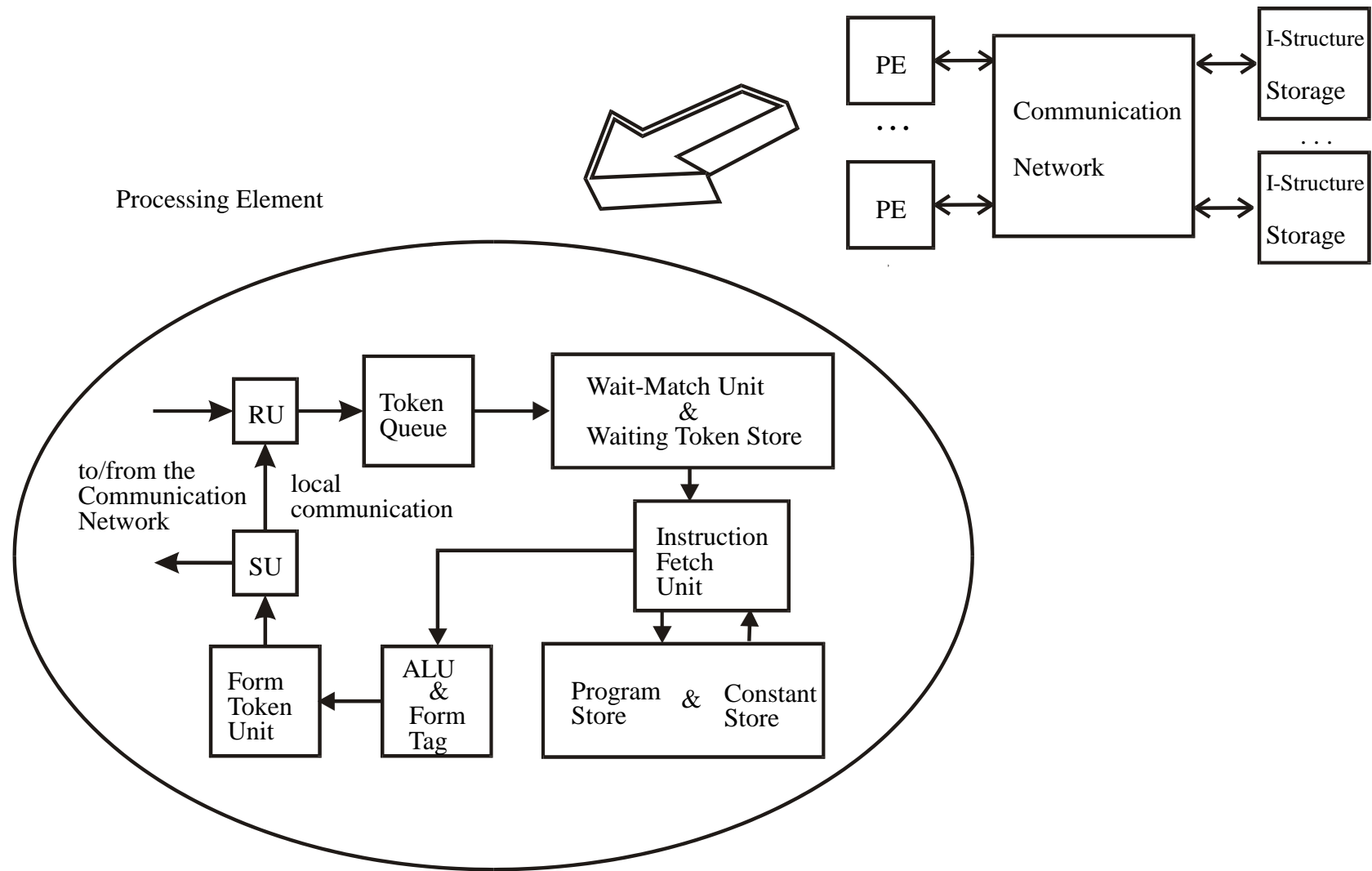
Manchester Dataflow Computer



Manchester Dataflow Machine



MIT Tagged-Token Dataflow Architecture



Caracteristici importante ale grafului asociat fluxului de date

■ Funcționalitate:

- *Evaluarea unui graf de flux de date este echivalentă cu evaluarea funcției matematice corespunzătoare.*


■ Compozibilitate:

- *Grafurile fluxului de date pot fi combinate pentru a forma grafuri noi.*

Dataflow Static

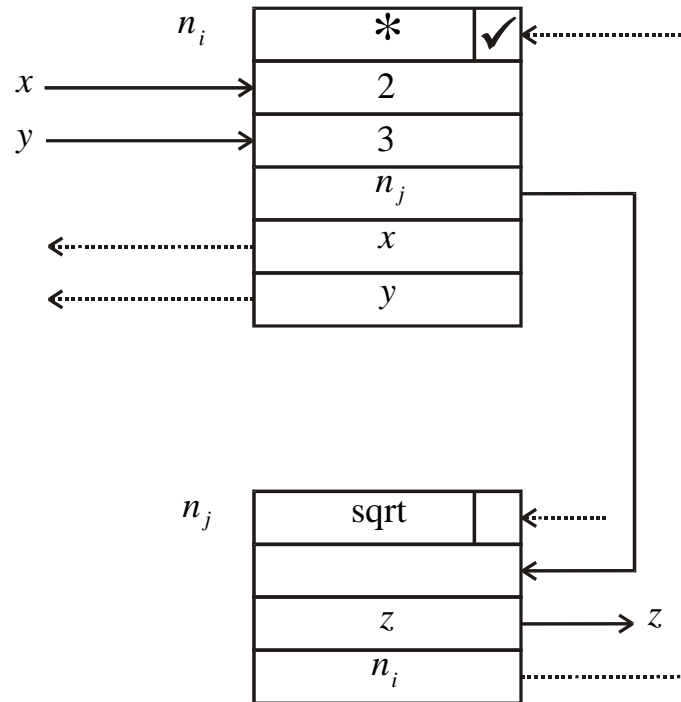
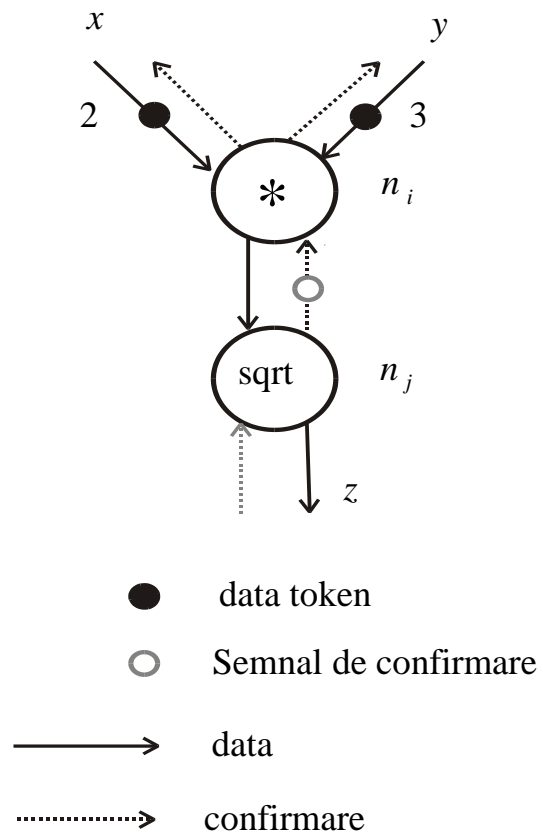
- Un graf de flux de date este reprezentat ca o colecție de șabloane de activitate, fiecare conținând:
 - codul instrucțiunii reprezentate,
 - sloturi de operand pentru păstrarea valorilor de operand,
 - câmpurile de adresă de destinație, referindu-se la sloturile de operand din șabloanele de activitate din secvențe care trebuie să primească valoarea rezultatului.

Deficiente dataflow static:

- Iterațiile consecutive ale unei bucle pot fi doar executate sub forma de pipeline.
- Datorită token-urilor (jetoanelor) de confirmare, traficul de token-uri este dublat.
- Lipsa suportului de programare care este esențial pentru limbajul de programare modern 
 - fără apeluri de procedură,
 - fără recursivitate.

Avantaj: model simplu de implementat

Dataflow : Template-ul de activitate



Semnale de confirmare

- Abordarea **statică** a fluxului de date permite **cel mult un simbol pe orice arc**.
- Nu pot fi token-uri (jetoane) diferite destinate aceleiași destinații.
- Extinderea regulii de bază pentru declansare se face astfel:
 - Un nod activat este declanșat dacă nu există nici un indicativ pe oricare dintre arcurile sale de ieșire.
- Implementarea restricției prin semnale de confirmare (jetoane suplimentare), transmise de-a lungul arcurilor suplimentare de la nodurile consumatoare către cele producătoare.
- Regula de declansare poate fi modificată la forma sa originală:
 - Un nod este declanșat în momentul în care devine activat.
- *Atentie: restricțiile structurale sunt ignorate presupunând resurse nelimitate!*

Dataflow Dinamic

- Fiecare iterație de buclă sau invocare de subprogram ar trebui să poată executa în paralel ca o instanță separată a unui subgraf reentrant.
- Replicarea este doar conceptuală.
- Fiecare *token* (jeton) are un *tag* (o *etichetă*):
 - adresa instrucțiunii pentru care este destinată valoarea particulară a datelor
 - informații de context
- Fiecare arc poate fi văzut ca o *multime* care poate conține un număr arbitrar de Token-uri cu tag-uri diferite.
- Regula de activare și declanșare este:
 - Un nod este activat și declanșat de îndată ce jetoanele cu etichete identice sunt prezente pe toate arcele de intrare.
- Restricțiile structurale sunt ignorate!

Interpretare de tip U

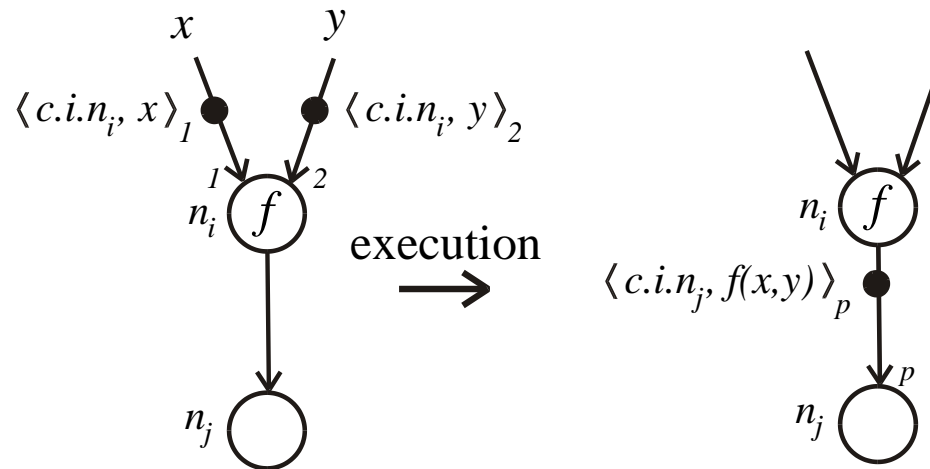
- Fiecare *token* constă dintr-un nume de activitate și date
 - numele activității cuprinde tag-ul.
- Tag (eticheta) are:
 - o adresă de instrucțiune **n**
 - câmpul contextual **c** care identifică în mod unic contextul în care urmează să fie invocată instrucțiunea,
 - numărul de inițiere **i** care identifică iterația de buclă în care are loc această activitate.
- De menționat faptul ca **c** este el însuși un nume de activitate.
- Deoarece instrucțiunea de destinație poate necesita mai multe intrări, fiecare token poartă și numărul portului său de *destinație* **p**.
- Token-ul se poate reprezenta prin $\langle c.i.n, data \rangle_p$
 $< c\text{-context} . i\text{-iteratia} . n\text{-adresa} , data >$

Interpretare de tip U

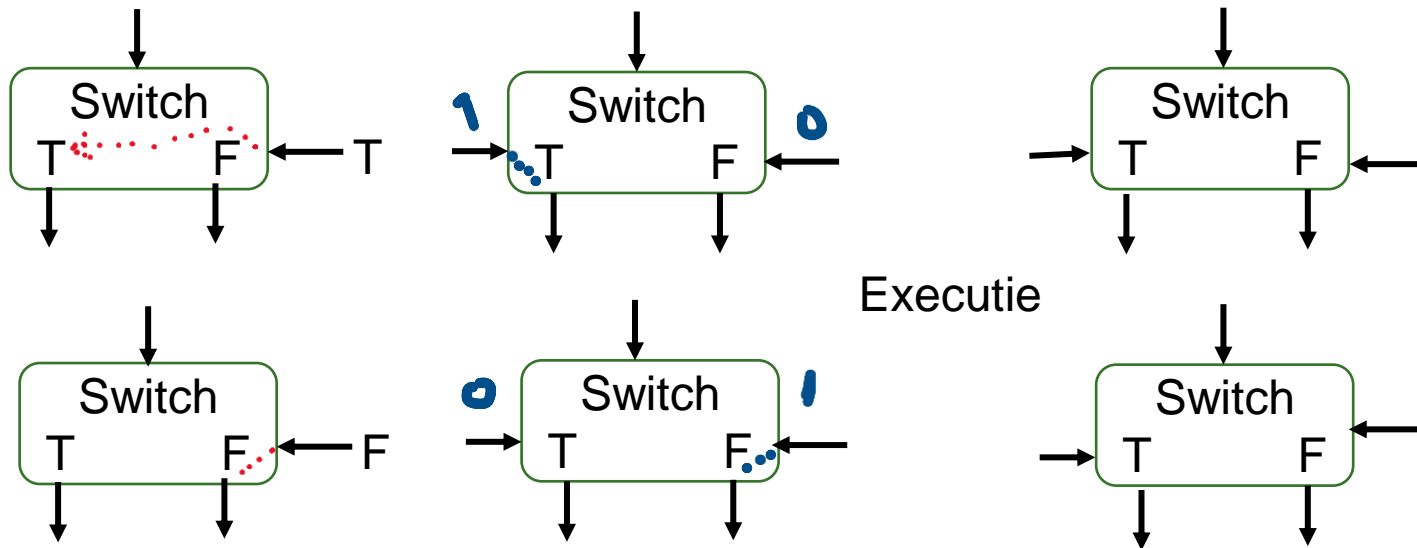
$$\langle c.i.n, data \rangle_p \quad < c\text{-context} . i\text{-iteratia} . n\text{-adresa} , data >$$

- dacă nodul n_i îndeplinește o funcție f și
- dacă portul p al lui n_j este destinația lui n_i ,
- atunci avem

$$in : \{ \langle c.i.n_i, x \rangle_1, \langle c.i.n_i, y \rangle_2 \} \quad out : \{ \langle c.i.n_j, f(x, y) \rangle_p \}$$



Implementare decizie

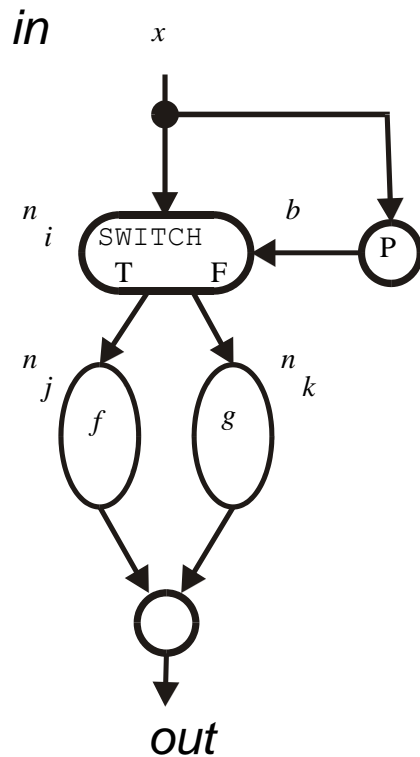


T/F sunt complementare
0/1 sau 1/0

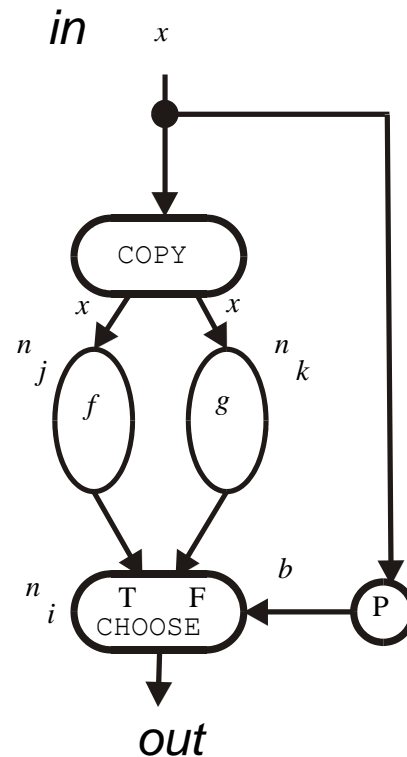
Implementare ramificatie (decizie)

$\langle c.i.n, data \rangle_p < c\text{-context} . i\text{-iteratia} . n\text{-adresa} , data >$

$in : \left\{ \left\langle c.i.n_i, x \right\rangle_{data} ; \left\langle c.i.n_i; b \right\rangle_{control} \right\}$ $out : \begin{cases} \{ \{ c.i.n_j, x \} \} & \text{if } b = T \\ \{ \{ c.i.n_k, x \} \} & \text{if } b = F \end{cases}$



Ramificatie
clasica



Evaluare
speculativa a
ramificatiei

L, L⁻¹, D, and D⁻¹ Operatori pentru implementarea Buclelor

$\langle c.i.n, data \rangle_p < c\text{-context} . i\text{-iteratia} . n\text{-adresa} , data >$

$L: in: \{\langle c.i.n_i, x \rangle\} \quad out: \{\langle c'.1.n_k, x \rangle\}, \quad unde \quad c' = \langle c.i.n_i \rangle$

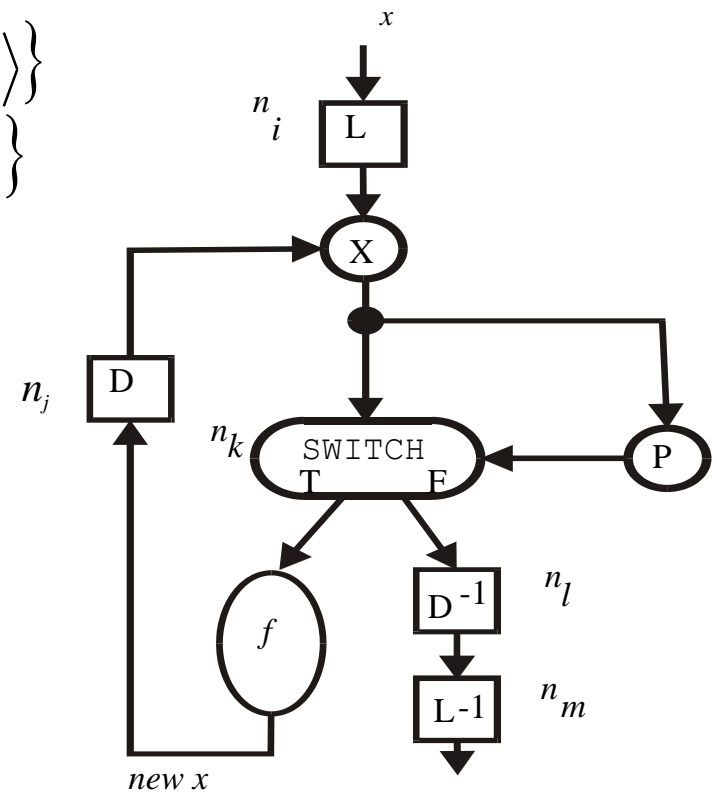
$D: in: \{\langle c'.j.n_j, x \rangle\} \quad out: \{\langle c'.j+1.n_k, x \rangle\}$

$D^{-1}: in: \{\langle c'.k.n_l, x \rangle\} \quad out: \{\langle c'.1.n_m, x \rangle\}$

$L^{-1}: in: \{\langle c'.1.n_m, x \rangle\} \quad out: \{\langle c'.i.n_n, x \rangle\}$

- L: Initalizare, context bucla
- D: incrementeaza contor bucla
- D⁻¹: initializeaza contor la 1
- L⁻¹: reface contextul original

Nota: **c** este el însuși un nume de activitate



A, A⁻¹, BEGIN, and END Operatori pentru functii

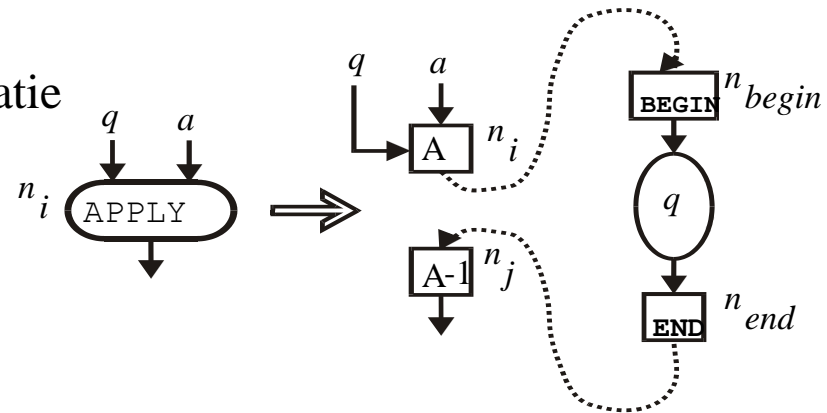
$\langle c.i.n, data \rangle_p < c-context . i-iteratia . n-adresa , data >$

■ **A:** $in: \{\langle c.i.n_i, q \rangle_{func}, \langle c.i.n_i, a \rangle_{arg}\}$ $out: \{\langle c'.i.n_{begin}, a \rangle\}$

unde $c' = \langle c.i.n_j \rangle$ si n_j este adresa operatorului A⁻¹

■ **END:** $in: \{\langle c'.i.n_{end}, q(a) \rangle\}$ $out: \{\langle c.i.n_j, q(a) \rangle\}$

- A: creaza un nou context
- BEGIN: replica token-ul pentru fiecare ramificatie
- END: returneaza rezultatul
- A⁻¹: replica iesirea pentru succesori

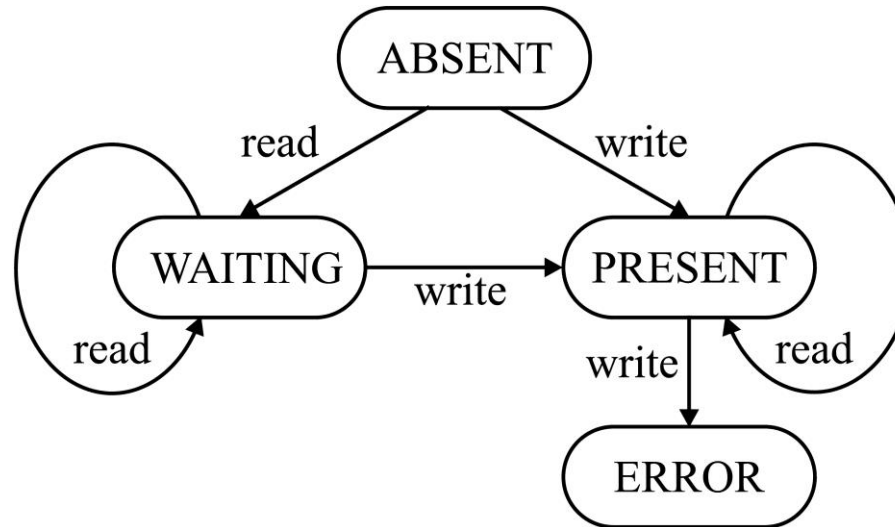


I-structures (I = incremental)

- Problemă: regulă de atribuire unică și structuri complexe de date
 - fiecare actualizare a unei structuri de date consumă structura și valoarea produce o nouă structură de date.
 - incomod sau chiar imposibil de implementat.
- Soluție: conceptul de structură I:
 - un depozit de date care respectă regula de atribuire unică
 - fiecare element al structurii I poate fi scris o singură dată, dar poate fi citit de orice număr de ori
- Ideea de bază este să asociați cu fiecare element biți de stare și o coadă de citiri amânate.

Structura I

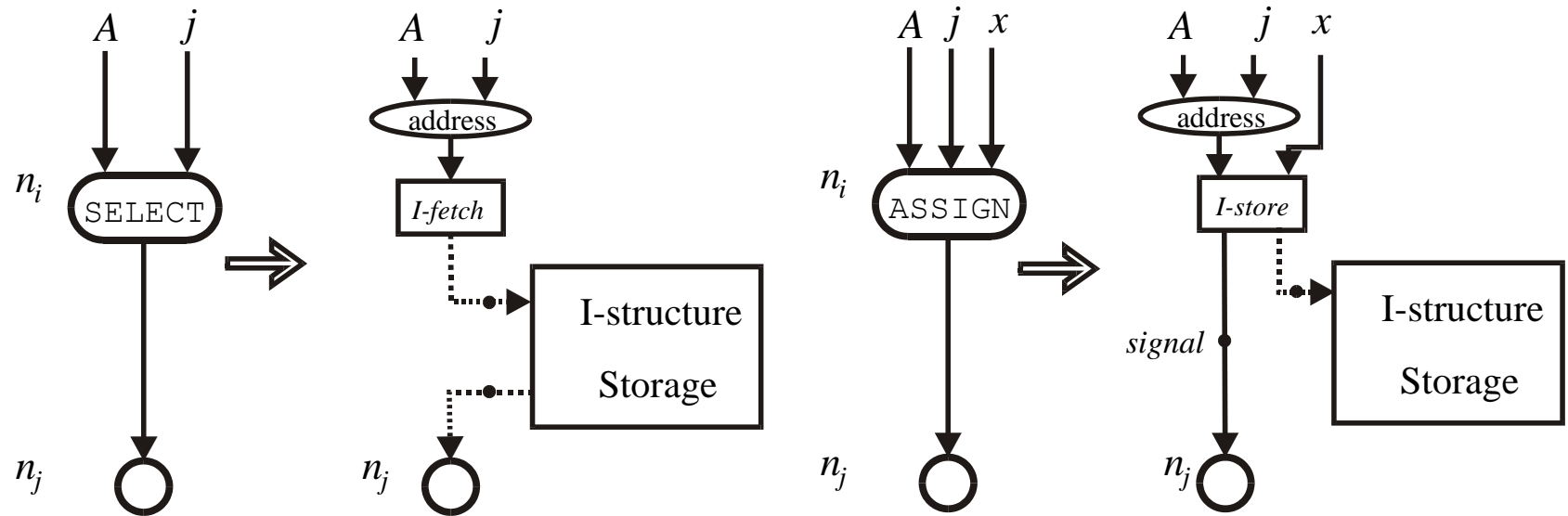
- Statutul fiecărui element al structurii I poate fi:
 - ❑ **present**: elementul poate fi citit, dar nu scris,
 - ❑ **absent**: o cerere de citire trebuie amânată, dar este permisă o operație de scriere în acest element,
 - ❑ **așteptare**: cel puțin o cerere de citire a elementului a fost amânată.



structura I

- Sunt definite trei operații elementare pe structurile I:
 - **alocare:** rezervă un număr specificat de elemente pentru o nouă structură I,
 - **I-fetch:** recuperează conținutul elementului structura - I specificat (dacă elementul nu a fost încă scris, atunci această operațiune este amânată automat),
 - **I-store:** scrie o valoare în elementul structura - I specificat (dacă acel element nu este gol, se raportează o condiție de eroare).
- Aceste operații elementare sunt utilizate pentru a construi noduri SELECT și ASSIGN.
- **I-fetch** este implementată ca operație de memorie în fază divizată:
 - o cerere de citire emisă unei structuri I este independentă în timp de răspunsul primit și, prin urmare, nu provoacă o așteptare de către PE emitent.

I-structure select and assign



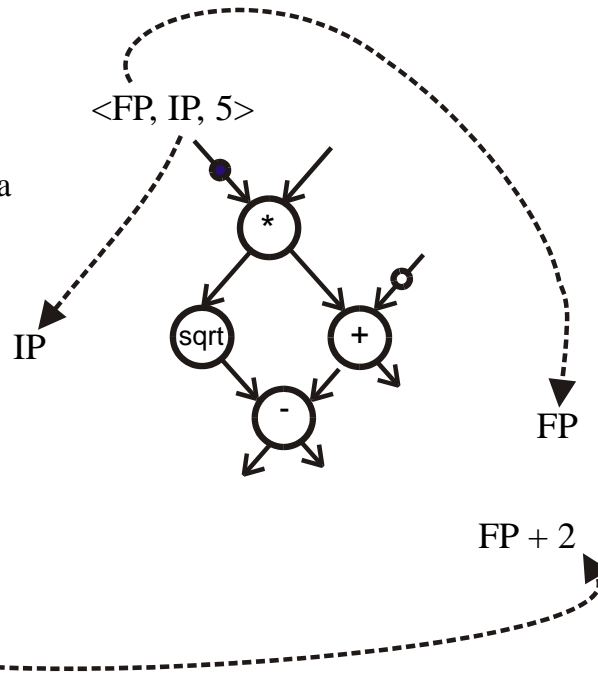
Implementare cu token explicit

- **Scop:** implementarea eficientă a potrivirii tokenurilor.
- **Ideea de bază:** se alocă un frame separat în memoria de frame-uri pentru fiecare iterație de buclă activă sau invocare de subprogram.
- Un ***frame*** este format din sloturi în care fiecare slot deține un operand care este utilizat în activitatea corespunzătoare.
- Deoarece accesul la sloturi este direct nu este necesară nici o căutare asociativă.

Explicit Token

Memoria de Instructiuni

Cod operatie	Offset de frame	destinatie	
		stanga	dreapta
*	2	+1	+2
sqrt			+2
+	3	+1	+5
-	5	+3	+2



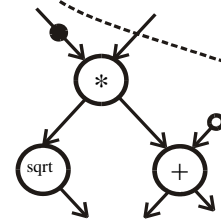
Memoria de frame-uri

bit de prezenta	valoare
✓	7.25

Explicit Token Store Matching Scheme

$t=0$

$\langle \text{FP}, \text{IP}, 3.01 \rangle$

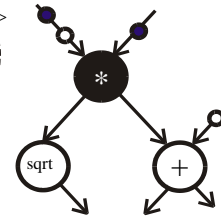


FP

presence bit	value
✓	2.34

$t=1$

$\langle \text{FP}, \text{IP}, 7.4 \rangle$

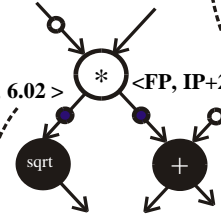


FP

presence bit	value
✓	3.01
✓	2.34

$t=2$

$\langle \text{FP}, \text{IP}+1, 6.02 \rangle$



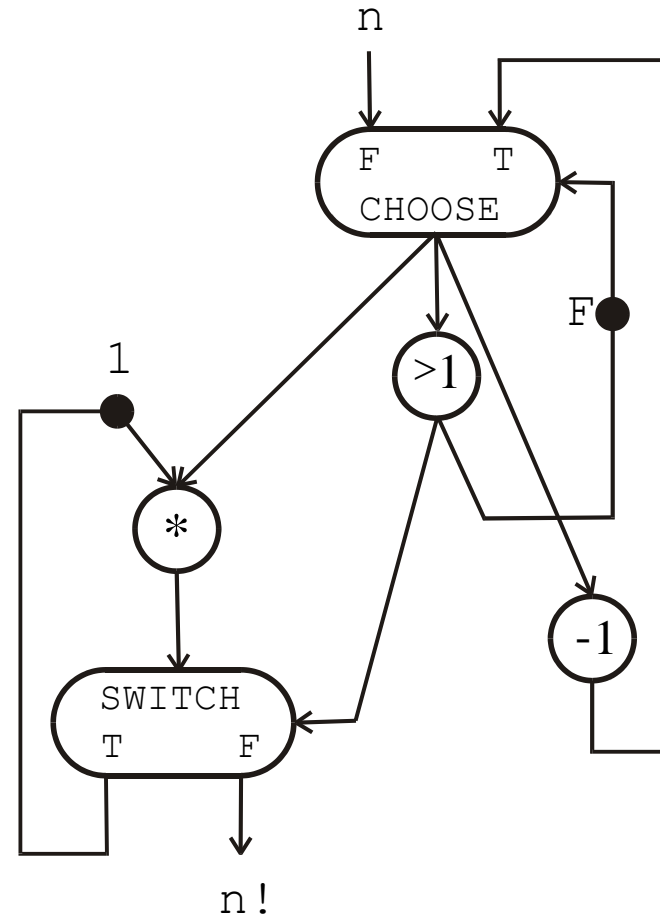
FP

presence bit	value
✓	2.34
✓	7.4

FP'

Exemplu calcul factorial n!

```
initial j = n; k = 1
while j > 1 do
  j = j - 1;
  k = k * j;
return k
```



Exemplu de prelucrare vectori

Input d,e,f

//a,b,c,d,e,f

//vectori de date

$C_0=0$

For i=1 to n

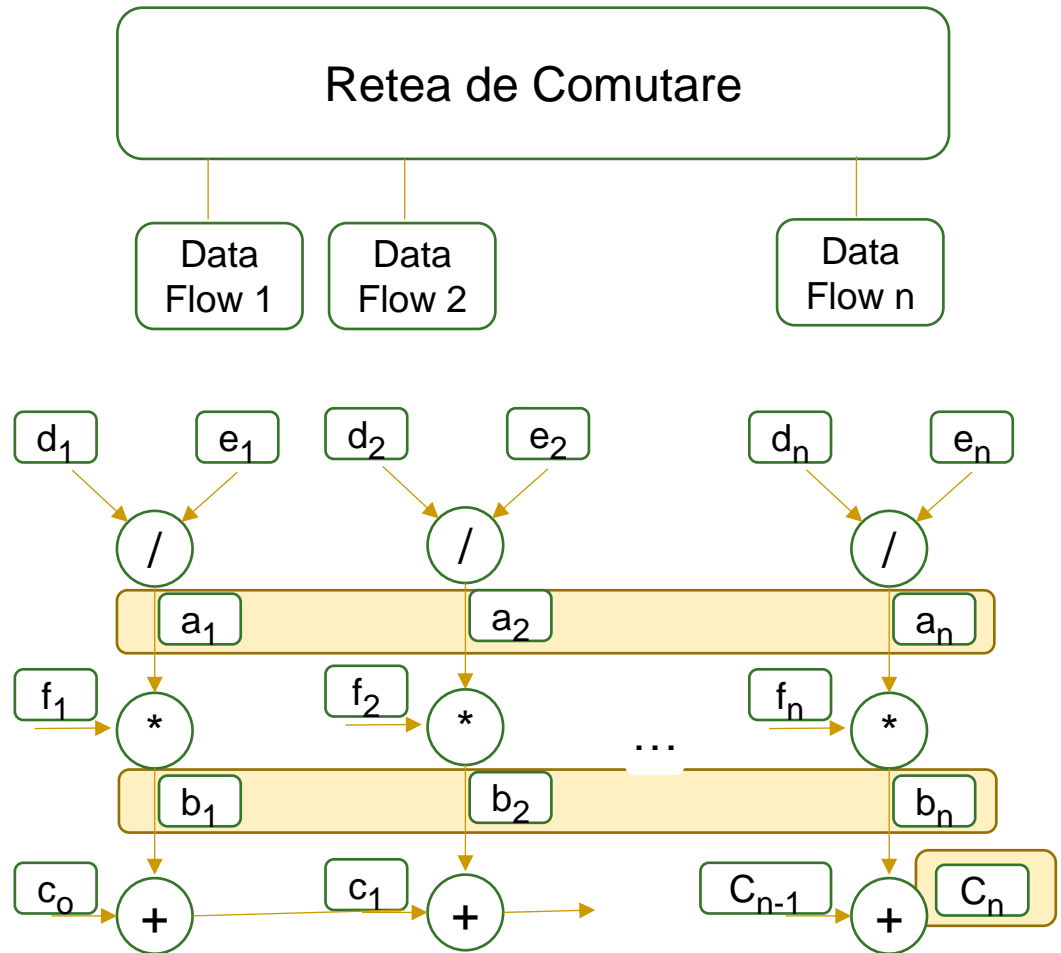
$a_i = d_i / e_i$

$b_i = a_i * f_i$

$c_i = b_i + c_{i-1}$

End for

Output a,b,c_n



Ordinea instructiunilor irelevanta

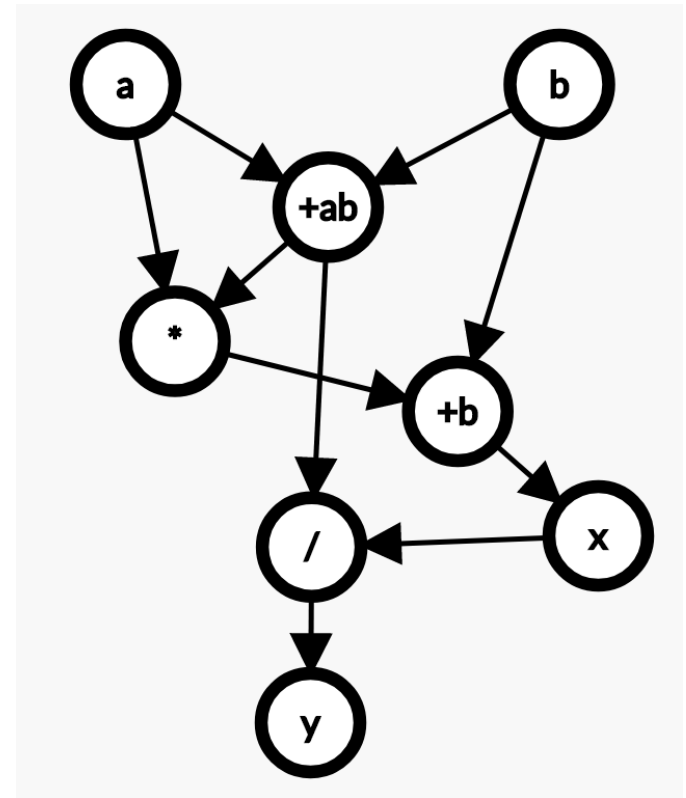
Exemplu:

input a, b

$y = (a+b) / x$

$x = (a * (a+b)) + b$

output y, x



Exemplu de bucla

```
input y, x
```

```
n = 0
```

```
while y < x do
```

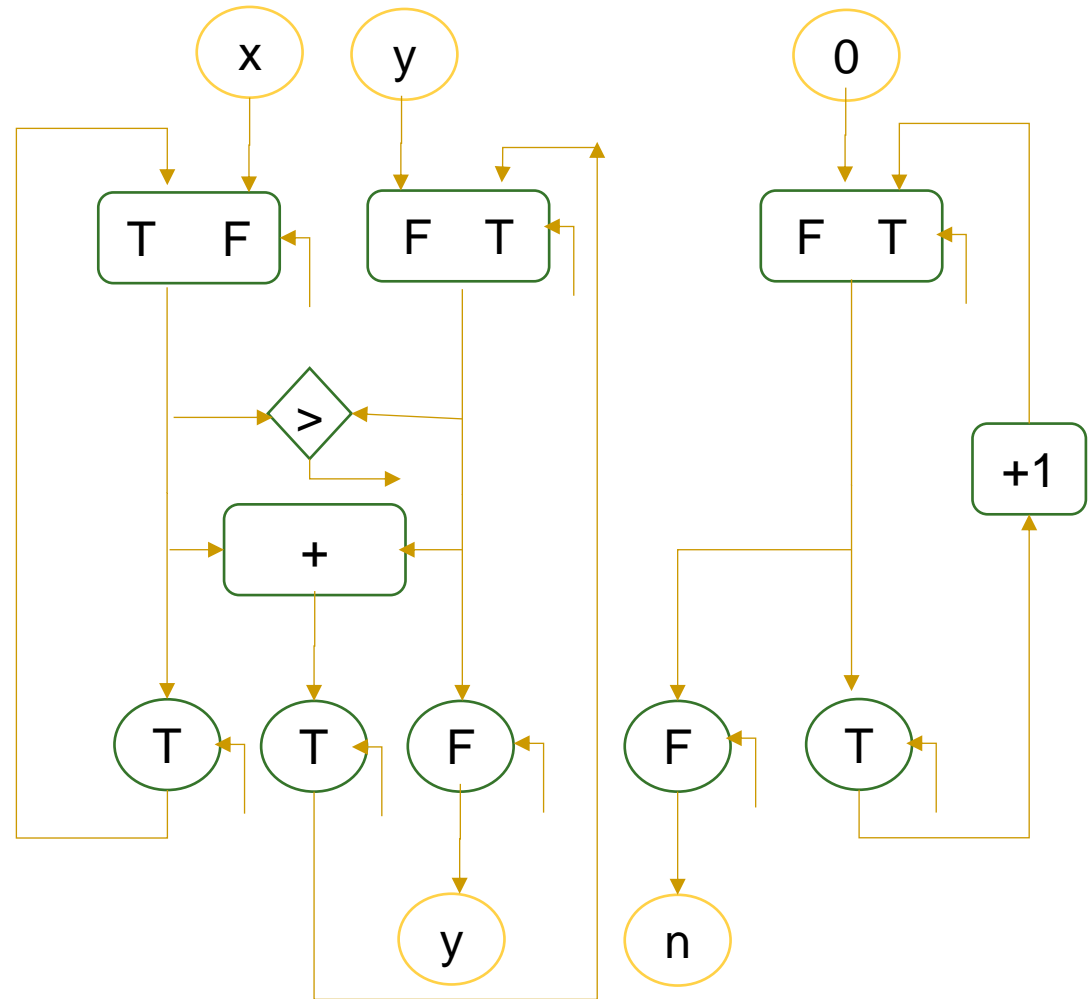
```
    y = y + x
```

```
    n = n + 1
```

```
end
```

```
output y, n
```

Nu s-a folosit array



Caracteristici de baza

- Executia instructiunilor este bazata pe disponibilitatea datelor
 - Datele sunt pastrate in cadrul instructiunii
 - Disponibilitatea datelor este verificata de o unitate specializata
 - Tokenul asociat datelor este verificat de o unitate specializata
 - Executia instructiunilor este asincrona
-

Avantaje/ Dezavantaje

Avantaje

- Potential ridicat de parallelism
- Viteza crescuta de executie
- Usor de integrat comunicatia si sincronizarea

Dezavantaje

- Overhead-ul mare pentru control
 - Manipularea dificila a structurilor de date
-

Unde se foloseste data flow in microprocesoarele actuale

Cea mai recentă generație de microprocesoare superscalare afișează o execuție dinamică în afara ordinii, denumită flux de date local sau flux de date micro.

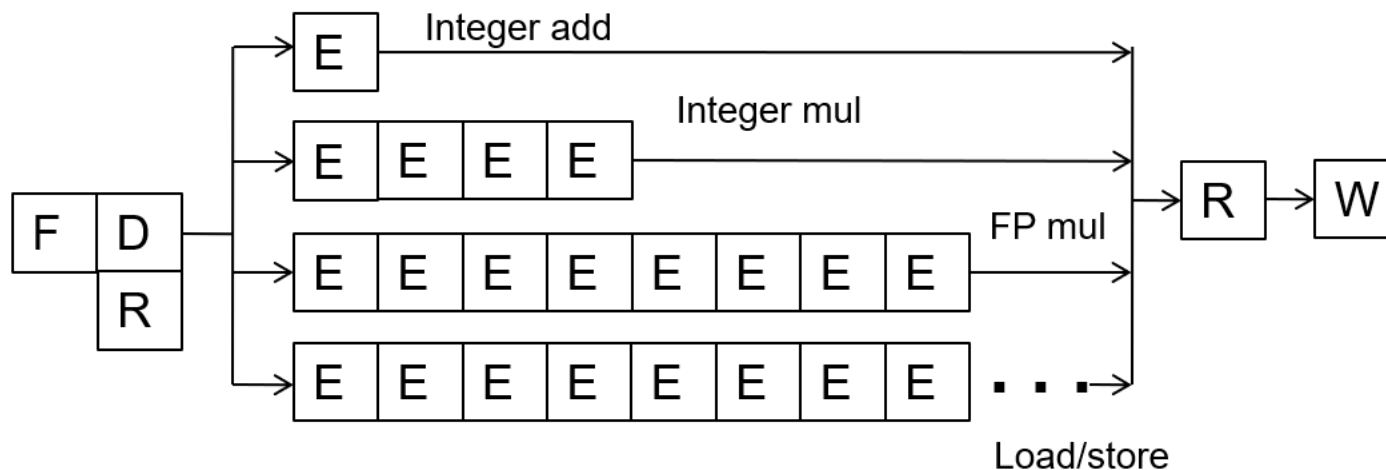
Colwell și Steck 1995, în prima lucrare despre PentiumPro:

Fluxul instrucțiunilor in Arhitectura Intel este prevăzut cu mecanismul prin care instrucțiunile sunt decodificate în micro-operații (μ pops), sau o serie de μ pops, iar aceste μ pops sunt plasate într-un set de microoperatii executabile speculative, care se vor executa în afara ordinii de operații (out of order), Vor fi executate pe principiul data flow (când operanzii sunt gata) și utilizate în ordinea programului sursă.

Fiecare instrucțiune este gata să fie executată imediat ce toți operanzii sunt disponibili.

In-Order Pipeline with Reorder Buffer


- **Decodificare (D):** Accesați fișierul de regulă/ROB, alocați intrarea în ROB, verificați dacă instrucțiunea poate fi executată, dacă da, trimiteți instrucțiunea (trimiteți la unitatea funcțională)
- **Executare (E):** Instrucțiunile se pot finaliza în afara ordinului
- **Finalizare (R):** scrieți rezultatul pentru a reordona bufferul
- **Retragere/Angajare (W):** Verificați dacă există excepții; dacă nu există, scrieți rezultatul în fișierul registrului arhitectural sau în memorie; altfel, resetati pipeline-ul și începeți de la gestionarea excepțiilor
- Expediere/execuție în ordine, finalizare în afara comenzii, în ordine



Tipuri de dependență de date

Dependență de date


$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$



Read-after-Write
(RAW)

Anti dependență

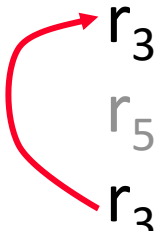
$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_1 \leftarrow r_4 \text{ op } r_5$



Write-after-Read
(WAR)

Dependență de iesiri

$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$
 $r_3 \leftarrow r_6 \text{ op } r_7$



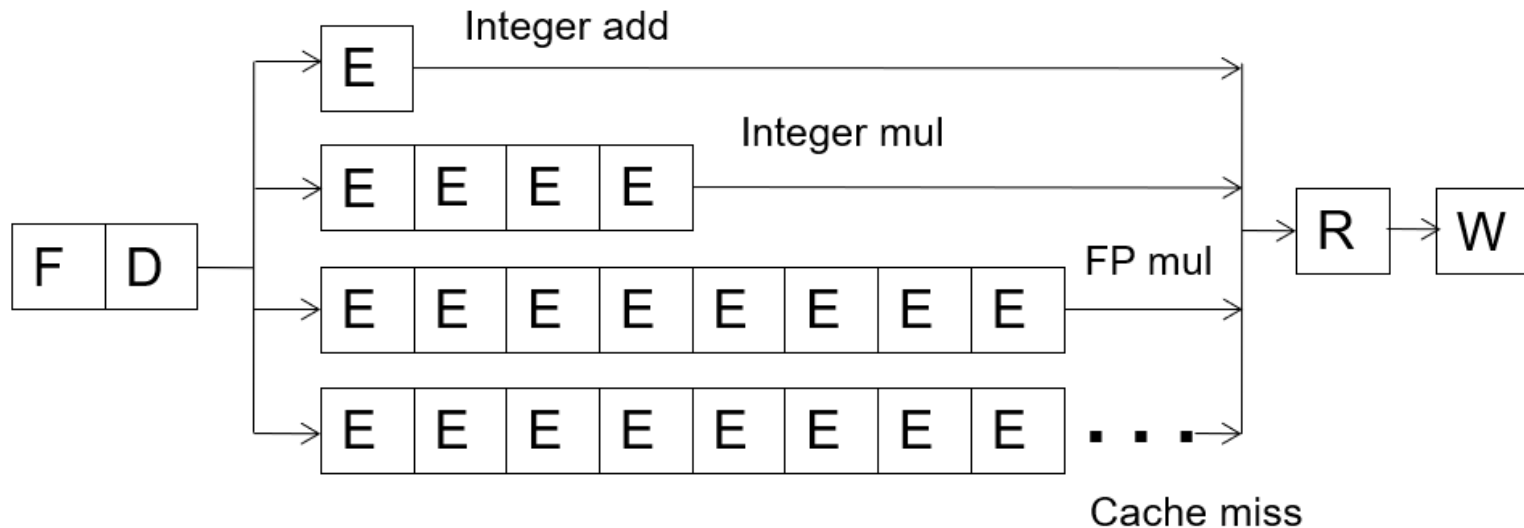
Write-after-Write
(WAW)

Redenumirea registrelor cu un buffer de reordonare

- Dependențe de ieșire și anti nu sunt dependențe adevărate
- DE CE?
 - Același registru se referă la valori care nu au nicio legătură între ele
 - Acestea există din cauza lipsei ID-urilor de registru (adică nume) în ISA
- ID-ul registrului este redenumit intrarea buffer-ului de reordonare care va păstra valoarea registrului
 - ID înregistrare → ID intrare ROB
 - ID registru architectural → ID registru fizic
 - După redenumire, ID-ul de intrare ROB a folosit pentru a se referi la registru
- Acest lucru elimină dependențele anti și de ieșire
- Dă iluzia că există un număr mare de registre

An In-order Pipeline

- Dispatch - Expediere: Act de trimitere a unei instrucțiuni către o unitate funcțională
- Redenumirea cu ROB elimină blocajele din cauza dependențelor false
- Problemă: O adevărată dependență de date blochează trimiterea instrucțiunilor mai noi în unități funcționale (de execuție).



Cum putem face mai bine?

- Ce au în comun următoarele două bucăți de cod (în ceea ce privește execuția din designul anterior)?

```
MUL  R3 ← R1, R2
ADD  R3 ← R3, R1
ADD  R4 ← R6, R7
MUL  R5 ← R6, R8
ADD  R7 ← R9, R9
```

```
LD    R3 ← R1 (0)
ADD  R3 ← R3, R1
ADD  R4 ← R6, R7
MUL  R5 ← R6, R8
ADD  R7 ← R9, R9
```

- Răspuns: În primul rând ADD blochează tot pipeline-ul!
 - ADD nu poate fi expedit deoarece registrul sursă nu este disponibil
 - Instrucțiunile independente ulterioare nu pot fi executate
- Cum diferă porțiunile de cod de mai sus?
 - Răspuns: Latența de încărcare este variabilă (necunoscut până la timpul de execuție)
 - Ce afectează asta? Gândiți-vă la compilator vs. microarhitectură

Prevenirea blocajelor de expediere

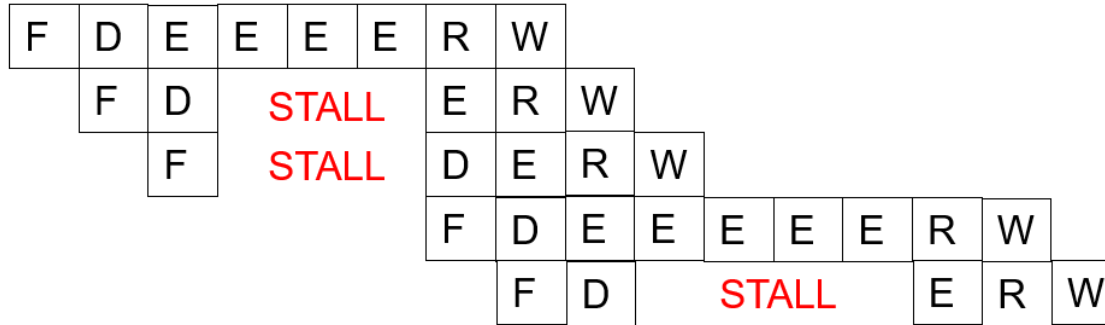
- Problemă: expediere în ordine (planificare sau execuție)
- Soluție: expediere în afara ordinii (programare sau execuție)
- De fapt, am văzut ideea de bază înainte:
 - Flux de date: „declanșează” o instrucțiune numai atunci când intrările sale sunt gata
 - Vom folosi principii similare, dar nu le vom expune în ISA
- Este o altă modalitate de a preveni blocajele de expediere?
 - 1. Programarea/reordonarea instrucțiunilor la compilare
 - 2. Predicția valorii
 - 3. Multithreading cu granulație fină

Execuție în afara ordinii (planificare dinamică)

- Idee: Mutați instrucțiunile dependente din calea celor independente (s.t. cele independente se pot executa)
 - Zone de odihnă pentru instrucțiuni dependente: Stații de rezervare
- Monitorizați „valorile” sursă ale fiecărei instrucțiuni din zona de odihnă (de așteptare).
- Când toate „valorile” sursei unei instrucțiuni sunt disponibile, „declanșează” (adică, expediază) instrucțiunea
 - Instrucțiunile expediate în ordinea fluxului de date (nu a fluxului de control).
- Beneficiu:
 - Toleranță la latență: Permite executarea și finalizarea instrucțiunilor independente în prezența unei operații cu latență lungă

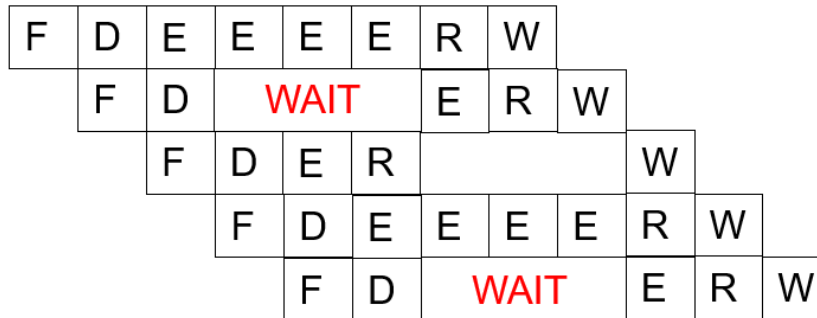
In-order vs. Out-of-order Dispatch

- In order dispatch + precise exceptions:



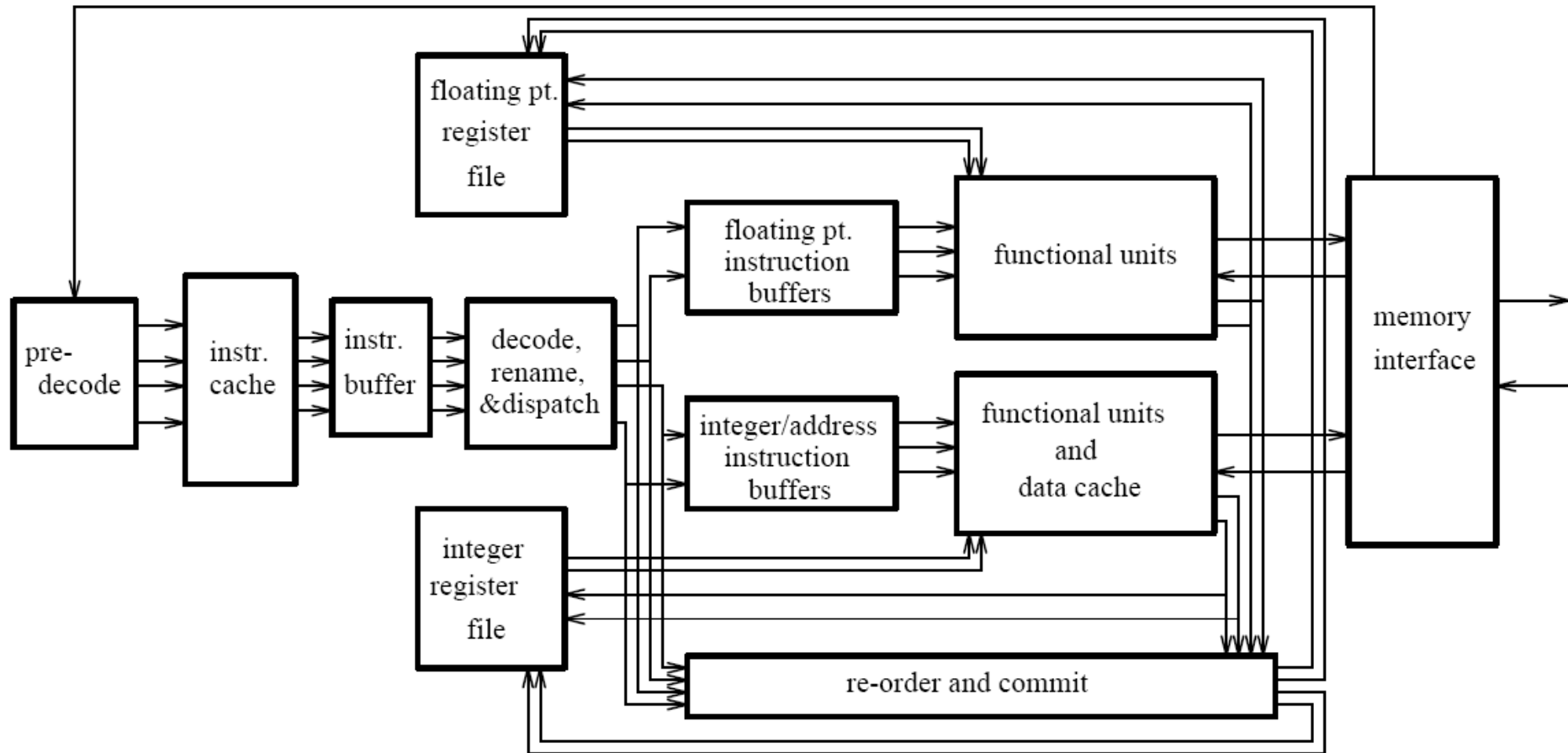
IMUL R3 \leftarrow R1, R2
 ADD R3 \leftarrow R3, R1
 ADD R1 \leftarrow R6, R7
 IMUL R5 \leftarrow R6, R8
 ADD R7 \leftarrow R3, R5

- Out-of-order dispatch + precise exceptions:



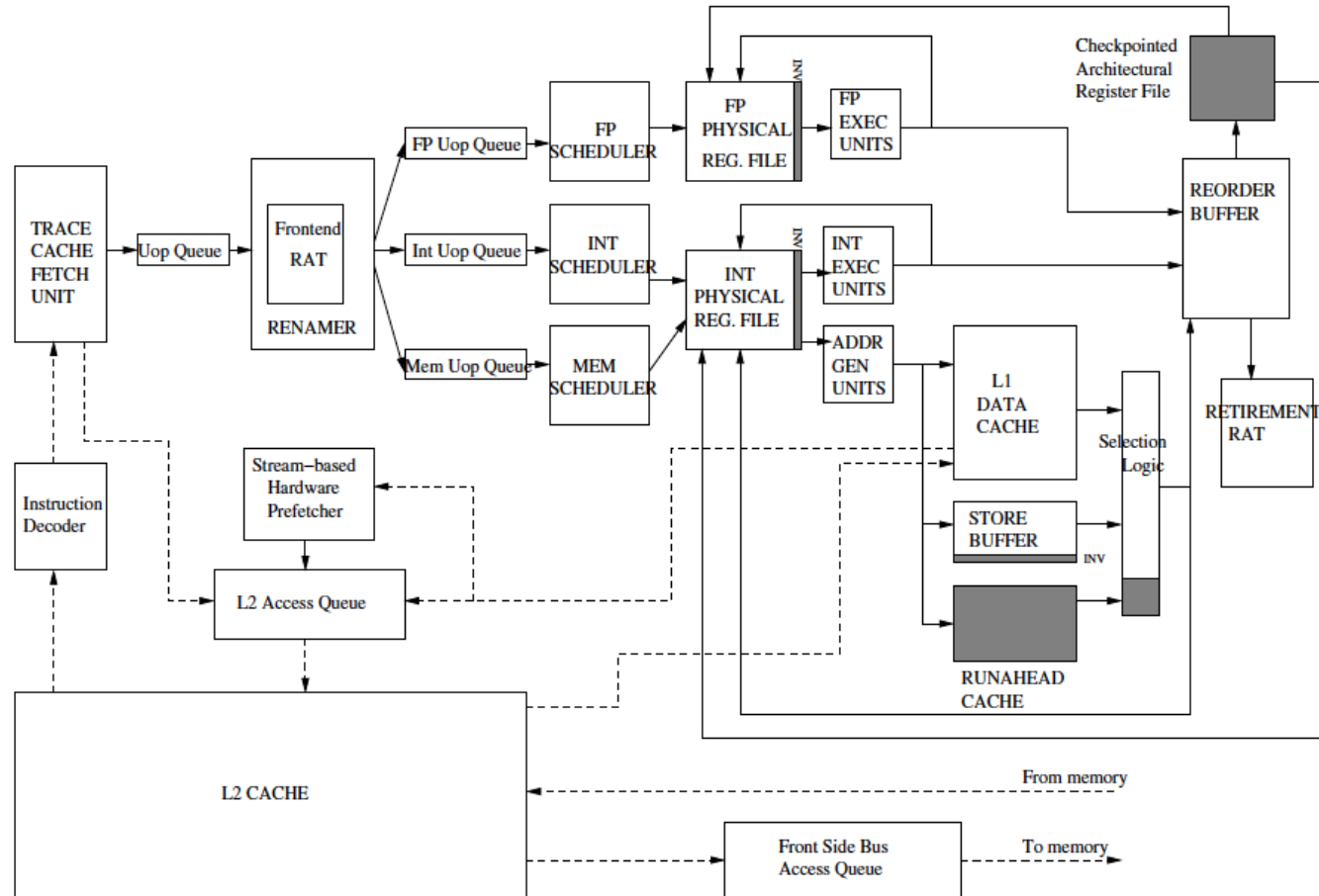
- 16 vs. 12 cycles

Organizarea generala a unui OoO Processor



Variantele OoO sunt folosite în majoritatea procesoarelor de înaltă performanță
Inițial în Intel Pentium Pro, AMD K5Alpha 21264, MIPS R10000, IBM POWER5, IBM z196, Oracle UltraSPARC T4, ARM Cortex A15, Apple M1, ...

Intel Pentium 4 Simplificat



IBM POWER5

Kalla et al., "IBM Power5 Chip: A Dual-Core Multithreaded Processor,"
IEEE Micro 2004.

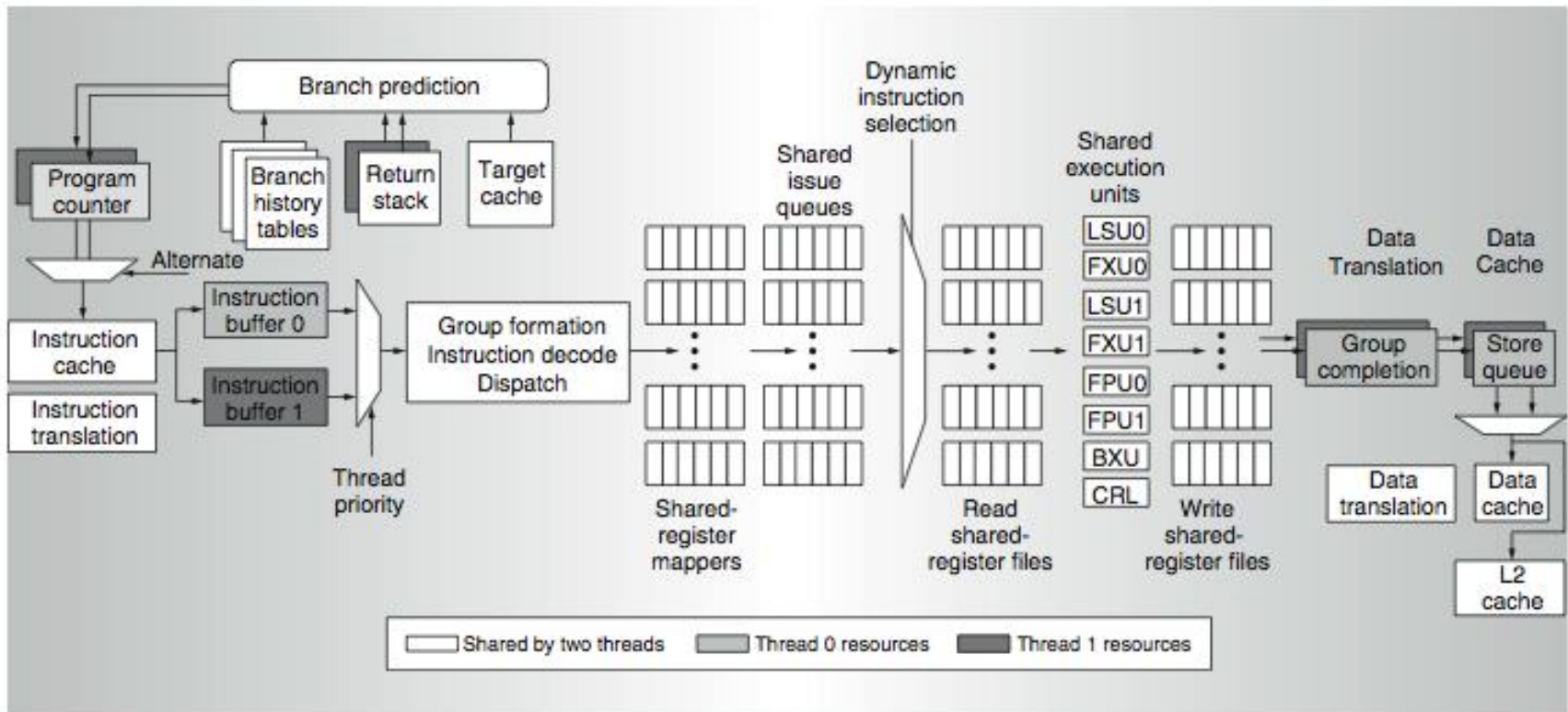
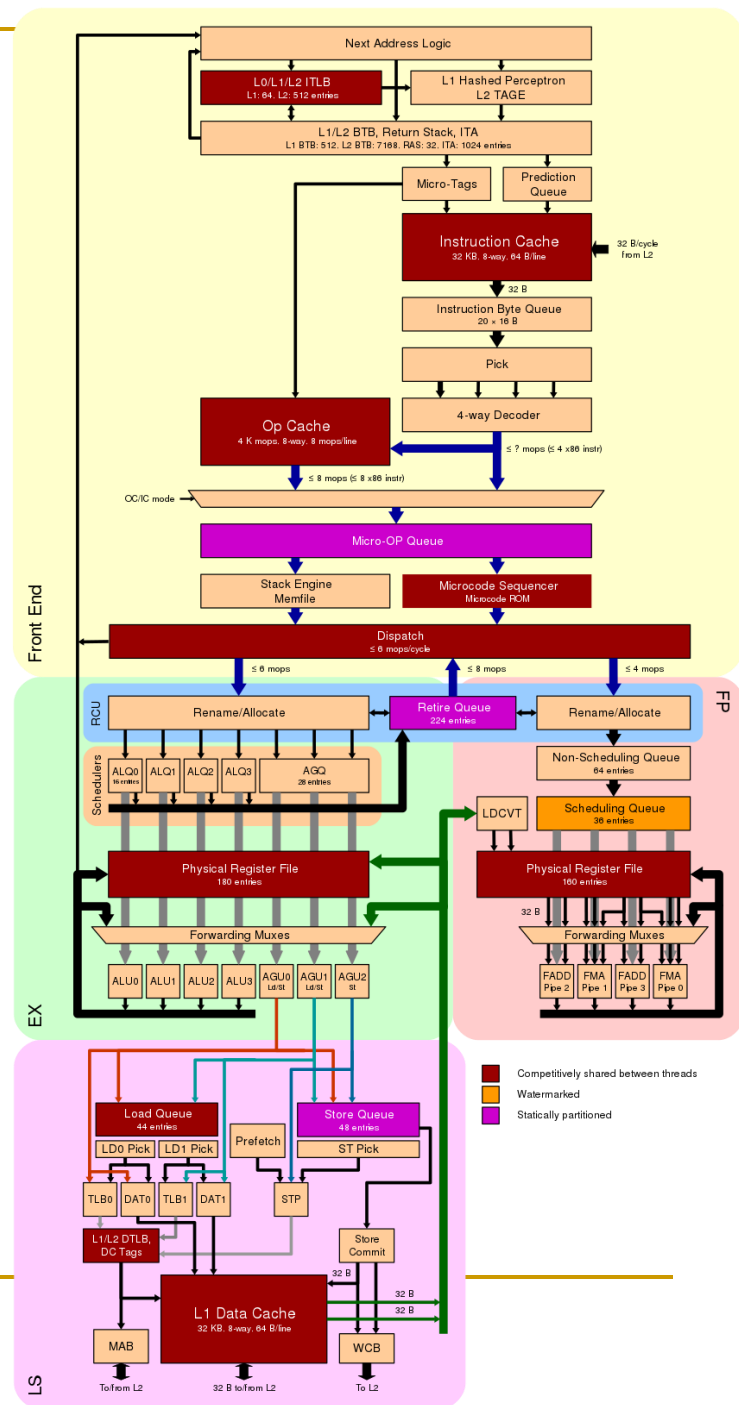
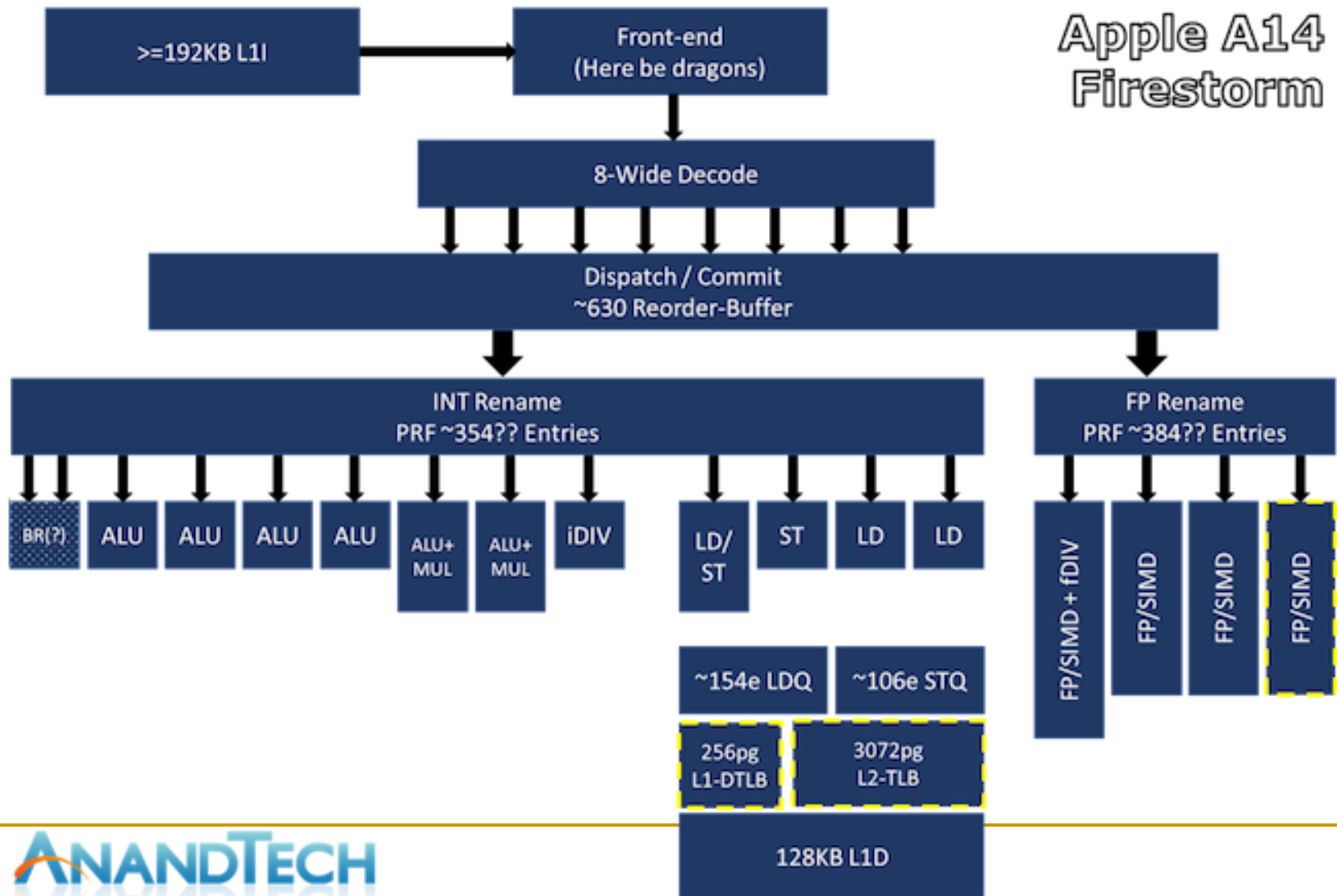


Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).

AMD Zen2? (2019)



Apple M1 FireStorm? (2020)



Tomasulo's Algorithm: Renaming

■ Tabelul de redenumire registre (tabelul de alias)

	Tag	Value	Valid?
R0			1
R1			1
R2			1
R3			1
R4			1
R5			1
R6			1
R7			1

Dacă Valid bit este setat, valoarea din tabel este corectă.
În caz contrar, Tag specifică unde se găsește valoarea corectă.

Eticheta – tag este un nume unic pentru valoarea de produs.

Recall from Precise Exceptions Lecture

Register File (RF)

	Value	Tag (pointer to ROB entry)
R0		
R1		
R2		
R3		
R4		
R5		
R6		
R7		

Reorder Buffer (ROB)

Register Buffer (RDB)

Entry 0					
Entry 1					
Entry 2					
Entry 8					
Entry 13					
Entry 14					
Entry 15					
	Entry Valid?	Dest reg ID	Dest reg value	Dest reg written?	

Cea mai veche instructiune

Cea mai noua instructiune

Register File (RF) or Register Alias Table (RAT)

R0			
R1			
R2			
R3			
R4			
R5			
R6			
R7			

Value Valid?

Value **Tag**
(indicatorul spre intrarea
stației de rezervare care
va produce valoarea)

Vom ignora Reorder Buffer pentru simplitate

Tomasulo's Algorithm

- Dacă stația de rezervare este disponibilă înainte de redenumire
 - Instrucțiune + operanzi redenumiți (valoare sursă/etichetă) introduși în stația de rezervare
 - Redenumiți numai dacă stația de rezervare este disponibilă
- Altfel **stau**
- Atât timp cât este în stația de rezervare, fiecare instrucțiune:
 - Urmărește magistrala de date comună (CDB) pentru eticheta-tag a surselor sale
 - Când eticheta-tag este văzută, luați valoarea pentru sursă și păstrați-o în stația de rezervare
 - Când ambii operanzi sunt disponibili, instrucțiunile sunt gata pentru a fi expediate
- Trimiteți instrucțiuni la unitatea funcțională când instrucțiunile sunt gata
- După terminarea instruirii în Unitatea Funcțională
 - Arbitraj pentru CDB Pune valoarea etichetată pe CDB (difuzare cu etichetă)
 - Fișierul de înregistrare este conectat la CDB
 - Register conține o etichetă care indică cel mai recent scriitor din registru
 - Dacă eticheta din fișierul de registru se potrivește cu eticheta de difuzare, scrieți valoarea de difuzare în registru (și setați bitul valid)
- Reveniți eticheta de redenumire
 - nicio copie validă a etichetei în sistem!

Exemplu

MUL R3 \leftarrow R1, R2

ADD R5 \leftarrow R3, R4

ADD R7 \leftarrow R2, R6

ADD R10 \leftarrow R8, R9

MUL R11 \leftarrow R7, R10

ADD R5 \leftarrow R5, R11



- Presupunem ADD (execuție în 4 cicluri),
- MUL (execuție în 6 cicluri)
- Să presupunem un sumator și un multiplicator
- Câte cicluri ?
 - într-o mașină fără pipeline: 50 de cicluri ($4 \cdot 7 + 2 \cdot 11$)
 - într-o mașină cu pipeline de expediere în comandă, cu excepții imprecise (fără expediere și redirectionare)
 - într-o mașină de expediere ieșită din comandă, excepții imprecise (redirectionare)

out-of-order dispatch pipelined

out-of-order dispatch pipelined machine w/ forwarding: **20 cycles**

			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
MUL	$R3 \leftarrow R1, R2$	F	D	E1	E2	E3	E4	E5	E6	W												
ADD	$R5 \leftarrow R3, R4$		F	D	-	-	-	-	-	E1	E2	E3	E4	W								
ADD	$R7 \leftarrow R2, R6$			F	D	E1	E2	E3	E4	W												
ADD	$R10 \leftarrow R8, R9$				F	D	E1	E2	E3	E4	W											
MUL	$R11 \leftarrow R7, R10$					F	D	-	-	-	E1	E2	E3	E4	E5	E6	W					
ADD	$R5 \leftarrow R5, R11$						F	D	-	-	-	-	-	-	-	-	E1	E2	E3	E4	W	

in-order-dispatch pipelined machine w/o forwarding: **31 cycles**

in-order-dispatch pipelined machine w/ forwarding: **25 cycles**

Our First OoO Machine Simulation

Program We Will Simulate

```
MUL R1, R2 → R3
ADD R3, R4 → R5
ADD R2, R6 → R7
ADD R8, R9 → R10
MUL R7, R10 → R11
ADD R5, R11 → R5
```

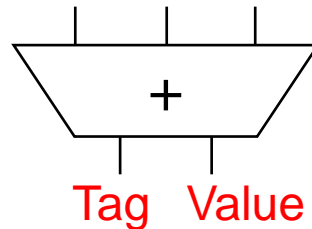
Initially:

1. Reservation Stations (RS's) are all Invalid (Empty)
2. All Registers are Valid

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		3
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

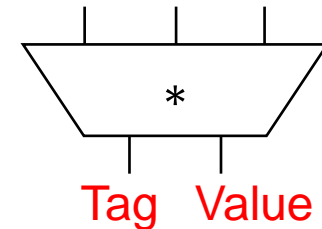
RS for ADD Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a						
b						
c						
d						



RS for MUL Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x						
y						
z						
t						



Register Alias Table

ADD and MUL Execution Units
have separate Tag & Value buses

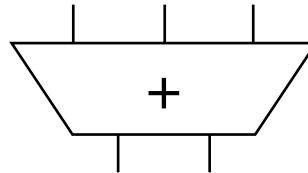
Cycle 0

Cycle

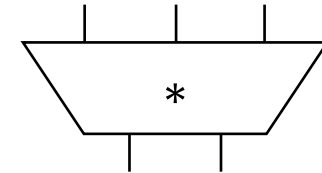
MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		3
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a						
b						
c						
d						



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x						
y						
z						
t						



Cycle 1

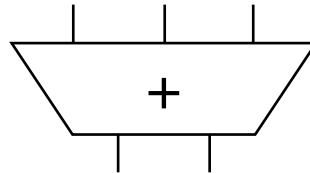
Cycle 1

F

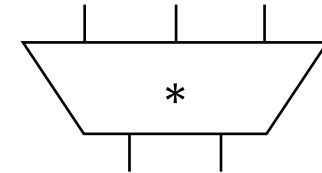
MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		3
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a						
b						
c						
d						



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x						
y						
z						
t						



Cycle 2

Cycle 1 2

MUL	R1, R2	→	R3
ADD	R3, R4	→	R5
ADD	R2, R6	→	R7
ADD	R8, R9	→	R10
MUL	R7, R10	→	R11
ADD	R5, R11	→	R5

F

D

F

MUL gets decoded and allocated into RS x

Step 1: Check if reservation station available. Yes: x

Step 2: Access the Register Alias Table

Step 3: Put source registers into reservation station x

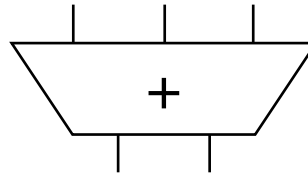
Step 4: Rename destination register R3 → x

R3 is now renamed to x.

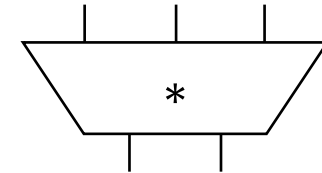
Its new value will be produced by the reservation station that is identified by tag x.

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a						
b						
c						
d						



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x		~			~	
y						
z						
t						



MUL in RS x is ready to execute in the next cycle!

Cycle 3

1. MUL in RS x starts executing

2. ADD gets decoded and allocated into RS a

Check readiness (Both sources ready?) → Wakeup

Ready → Dispatch the instruction to the MUL unit

Same Steps 1-4 for ADD... Rename R5 → a

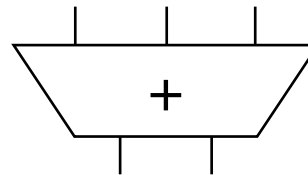
Cycle 1 2 3

MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

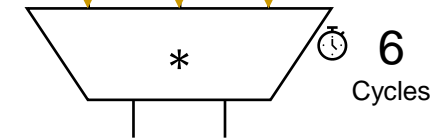
F D E₁
 F D
 F

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	a	
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a					~	
b						
c						
d						



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y						
z						
t						



ADD in RS a cannot execute in the next cycle: one source is not valid

Cycle 4

Cycle 1 2 3 4

MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

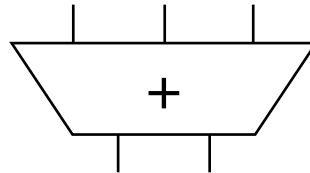
F D E₁ E₂
 F D -
 F D
 F

ADD in RS a waits because one source is not valid.

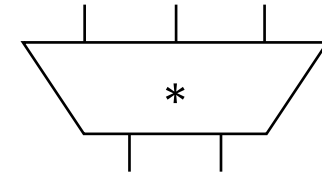
Rename R7 → b

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	a	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	1		10
R11	1		11

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	0	x		1	~	4
b		~			~	
c						
d						



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y						
z						
t						



ADD in RS b is ready to execute in the next cycle!

It will be executed out of order in the next cycle.

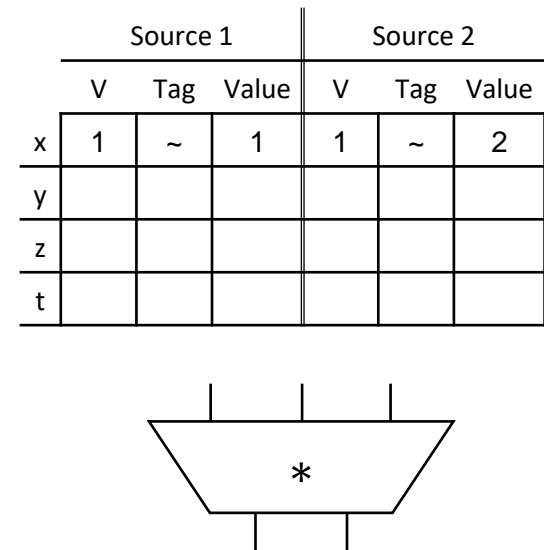
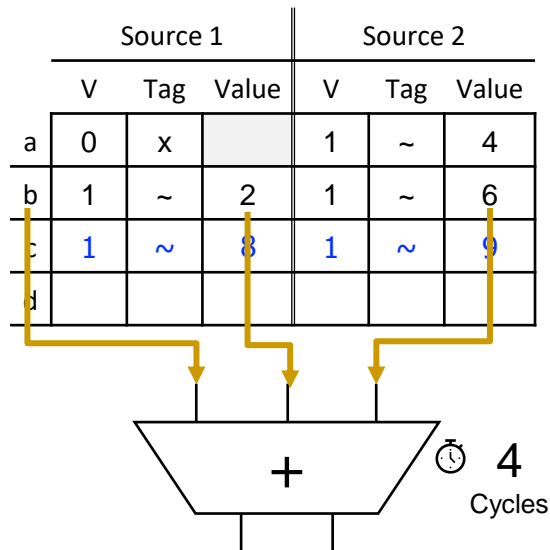
Cycle 5

Cycle 1 2 3 4 5

MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

F D E₁ E₂ E₃
 F D - -
 F D E₁
 F D
 F

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	a	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	1		11



ADD in RS c is ready to execute in the next cycle!

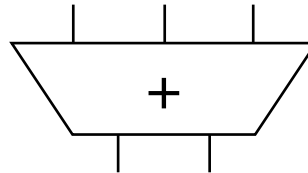
Cycle 6

MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

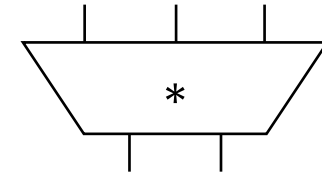
Cycle	1	2	3	4	5	6
F		D	E ₁	E ₂	E ₃	E ₄
		F	D	-	-	-
			F	D	E ₁	E ₂
				F	D	E ₁
					F	D
						F

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	a	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	0	x		1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d						



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	0	b		0	c	
z						
t						



Cycle 7

All six instructions are now decoded and renamed

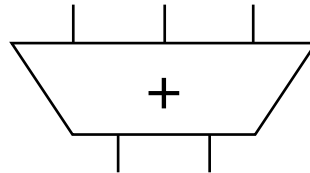
Note what happened to R5: Renamed twice!

MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

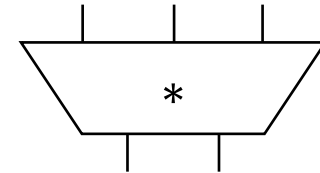
Cycle	1	2	3	4	5	6	7
F		D	E ₁	E ₂	E ₃	E ₄	E ₅
		F	D	-	-	-	-
			F	D	E ₁	E ₂	E ₃
				F	D	E ₁	E ₂
					F	D	-
						F	D

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	d	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	0	x		1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	0	b		0	c	
z						
t						



Cycle 8 (First Slide)

MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

Cycle 1 2 3 4 5 6 7 8

F D E₁ E₂ E₃ E₄ E₅ E₆
 F D - - - -
 F D E₁ E₂ E₃
 F D E₁ E₂
 F D -
 F D

MUL in RS x is done

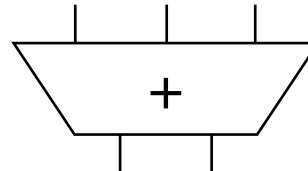
Broadcast MUL's tag (x)

- ✓ Check tag
- ✓ Check for invalidity

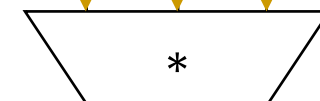
Broadcast MUL's result (2)

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1		2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	0	b		0	c	
z						
t						



x

2

03

ADD in RS a is ready to execute in the next cycle!

Cycle 8 (Second Slide)

Cycle 1 2 3 4 5 6 7 8

MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

F D E₁ E₂ E₃ E₄ E₅ E₆
 F D - - - - -
 F D E₁ E₂ E₃ E₄
 F D E₁ E₂
 F D -
 F D

ADD in RS b is also done

Broadcast ADD's tag (b)

- ✓ Check tag
- ✓ Check for invalidity

Broadcast ADD's result (8)

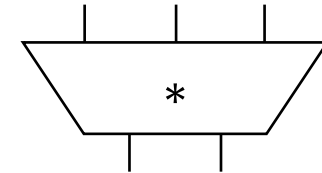
Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1		8	0	c	
z						
t						



b 8



MUL in RS y is still NOT ready to execute in the next cycle!

Cycle 8 (Third Slide)

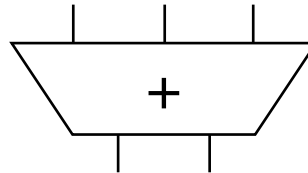
MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

Cycle 1 2 3 4 5 6 7 8

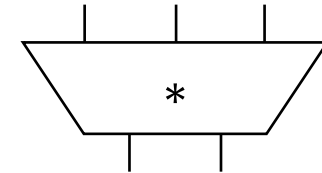
F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆
	F	D	-	-	-	-	-
		F	D	E ₁	E ₂	E ₃	E ₄
			F	D	E ₁	E ₂	E ₃
				F	D	-	-
					F	D	-

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	0	c	
z						
t						



Cycle 9

MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

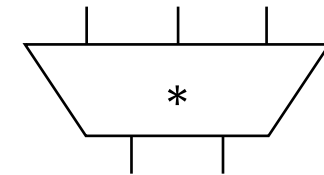
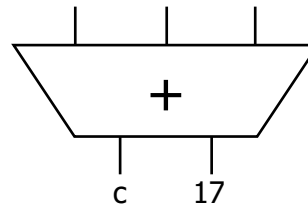
Cycle	1	2	3	4	5	6	7	8	9
F		D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W
		F	D	-	-	-	-	-	E ₁
			F	D	E ₁	E ₂	E ₃	E ₄	W
				F	D	E ₁	E ₂	E ₃	E ₄
					F	D	-	-	-
						F	D	-	-

Broadcast and Update

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



MUL in RS y is ready to execute in the next cycle!

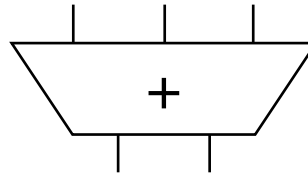
Cycle 10

MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

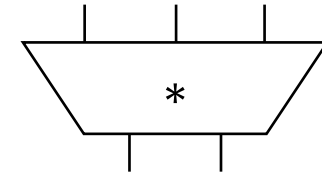
Cycle	1	2	3	4	5	6	7	8	9	10
F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W		
	F	D	-	-	-	-	-	E ₁	E ₂	
		F	D	E ₁	E ₂	E ₃	E ₄	W		
			F	D	E ₁	E ₂	E ₃	E ₄	W	
				F	D	-	-	-	E ₁	
					F	D	-	-	-	

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



Cycle 11

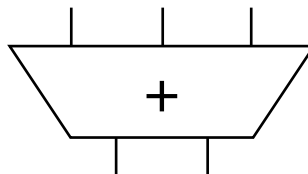
MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

Cycle 1 2 3 4 5 6 7 8 9 10 11

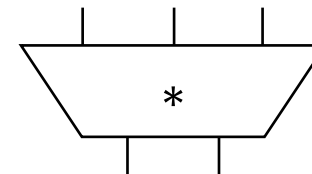
F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W		
	F	D	-	-	-	-	-	E ₁	E ₂	E ₃
		F	D	E ₁	E ₂	E ₃	E ₄	W		
			F	D	E ₁	E ₂	E ₃	E ₄	W	
				F	D	-	-	-	E ₁	E ₂
					F	D	-	-	-	-

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



Cycle 12

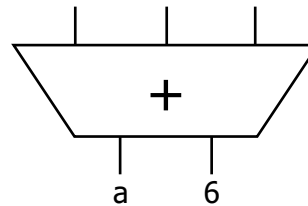
MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

Cycle	1	2	3	4	5	6	7	8	9	10	11	12
F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W				
	F	D	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄	
		F	D	E ₁	E ₂	E ₃	E ₄	W				
			F	D	E ₁	E ₂	E ₃	E ₄	W			
				F	D	-	-	-	E ₁	E ₂	E ₃	
					F	D	-	-	-	-	-	-

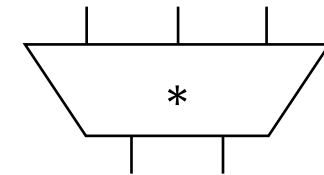
← Broadcast and Update

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	0	y	



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



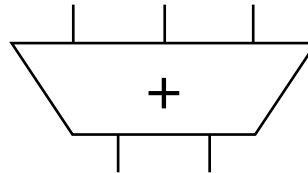
Cycle 13

MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

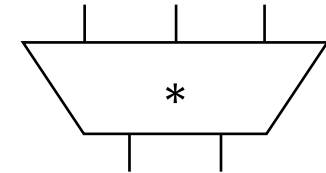
Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W					
	F	D	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄	W	
		F	D	E ₁	E ₂	E ₃	E ₄	W					
			F	D	E ₁	E ₂	E ₃	E ₄	W				
				F	D	-	-	-	E ₁	E ₂	E ₃	E ₄	
					F	D	-	-	-	-	-	-	-

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	0	y	



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



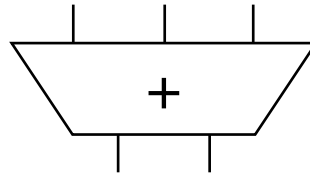
Cycle 14

MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

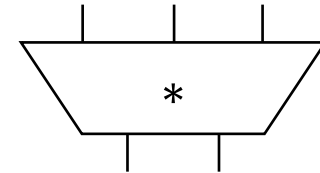
Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14
F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W						
	F	D	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄	W		
		F	D	E ₁	E ₂	E ₃	E ₄	W						
			F	D	E ₁	E ₂	E ₃	E ₄	W					
				F	D	-	-	-	E ₁	E ₂	E ₃	E ₄	E ₅	
					F	D	-	-	-	-	-	-	-	-

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	0	y	



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



Cycle 15

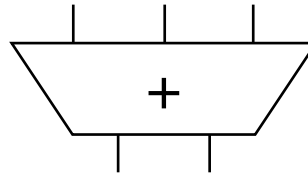
MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W							
	F	D	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄	W			
		F	D	E ₁	E ₂	E ₃	E ₄	W							
			F	D	E ₁	E ₂	E ₃	E ₄	W						
				F	D	-	-	-	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	-
					F	D	-	-	-	-	-	-	-	-	-

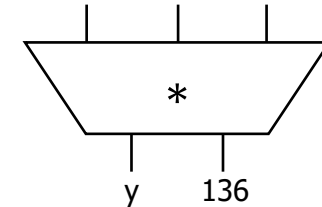
Broadcast and Update

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	1		136

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	1	~	136



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



ADD in RS d is ready to execute in the next cycle!

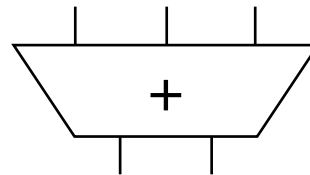
Cycle 16

MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

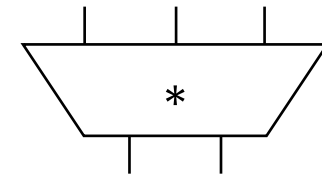
Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W								
	F	D	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄	W				
		F	D	E ₁	E ₂	E ₃	E ₄	W								
			F	D	E ₁	E ₂	E ₃	E ₄	W							
				F	D	-	-	-	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W	
					F	D	-	-	-	-	-	-	-	-	-	E ₁

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	1		136

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	1	~	136



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



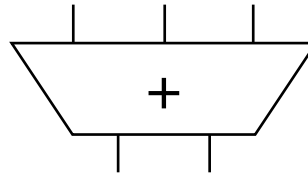
Cycle 17

MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

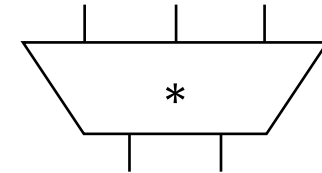
Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W									
	F	D	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄	W					
		F	D	E ₁	E ₂	E ₃	E ₄	W									
			F	D	E ₁	E ₂	E ₃	E ₄	W								
				F	D	-	-	-	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W		
					F	D	-	-	-	-	-	-	-	-	-	E ₁	E ₂

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	1		136

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	1	~	136



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



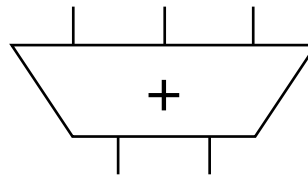
Cycle 18

MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

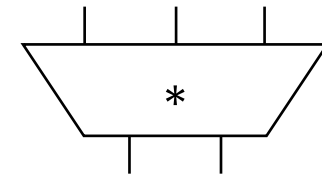
Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W										
	F	D	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄	W						
		F	D	E ₁	E ₂	E ₃	E ₄	W										
			F	D	E ₁	E ₂	E ₃	E ₄	W									
				F	D	-	-	-	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W			
					F	D	-	-	-	-	-	-	-	-	-	E ₁	E ₂	E ₃

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	1		136

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	1	~	136



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



Cycle 19

MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

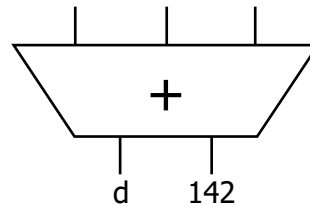
Cycle 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

F D E₁ E₂ E₃ E₄ E₅ E₆ W
 F D - - - - E₁ E₂ E₃ E₄ W
 F D E₁ E₂ E₃ E₄ W
 F D E₁ E₂ E₃ E₄ W
 F D - - - E₁ E₂ E₃ E₄ E₅ E₆ W
 F D - - - - - E₁ E₂ E₃ E₄

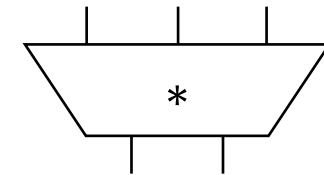
Broadcast and Update

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	1		142
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	1		136

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	1	~	136



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



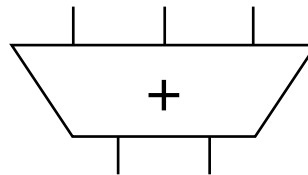
Cycle 20

MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

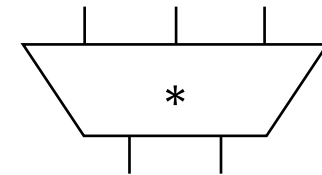
Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
F	D	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W												
	F	D	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄	W								
		F	D	E ₁	E ₂	E ₃	E ₄	W												
			F	D	E ₁	E ₂	E ₃	E ₄	W											
				F	D	-	-	-	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	W					
					F	D	-	-	-	-	-	-	-	-	-	E ₁	E ₂	E ₃	E ₄	W

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	1		142
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	1		136

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	1	~	136



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



Intrebări?

- Ce este necesar în hardware pentru a realiza difuzarea etichetelor și captarea valorii?
 - face o valoare valabilă
 - trezeste o instrucțiune

Fire, comparatoare și logică
- Eticheta trebuie să fie ID-ul intrării în stația de rezervare?

Nu, poate fi orice nume unic care permite conectarea de la producător la consumator
- Ce poate deveni potențial calea critică?
 - Tag broadcast → captura de valoare → instrucțiune trezire
- Cum puteți reduce potențialele căi critice?
 - Mai multe pipeline și predicții

State of RAT and RS in Cycle 7

Cycle 1 2 3 4 5 6 7

MUL R1, R2 → R3
 ADD R3, R4 → R5
 ADD R2, R6 → R7
 ADD R8, R9 → R10
 MUL R7, R10 → R11
 ADD R5, R11 → R5

F D E₁ E₂ E₃ E₄ E₅
 F D - - - -
 F D E₁ E₂ E₃
 F D E₁ E₂
 F D -
 F D

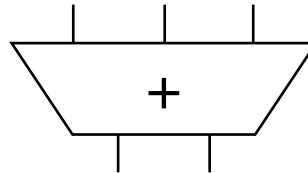
All 6 instructions are decoded and renamed

Note what happened to R5: Renamed twice!

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	d	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

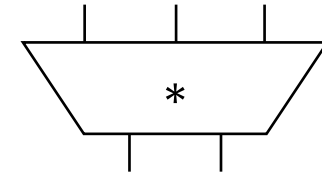
RS for ADD Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	0	x		1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	



RS for MUL Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	0	b		0	c	
z						
t						



Register Alias Table

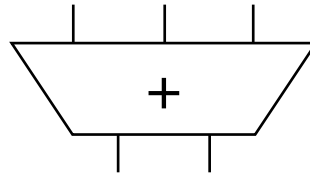
State of RAT and RS in Cycle 7

1. Desenați graficul fluxului de date pentru codul de execuție
2. Furnizați codul de execuție în ordine secvențială

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	d	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

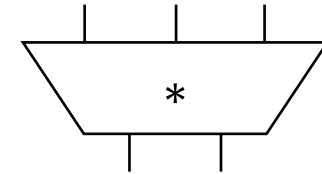
RS for ADD Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	0	x		1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	



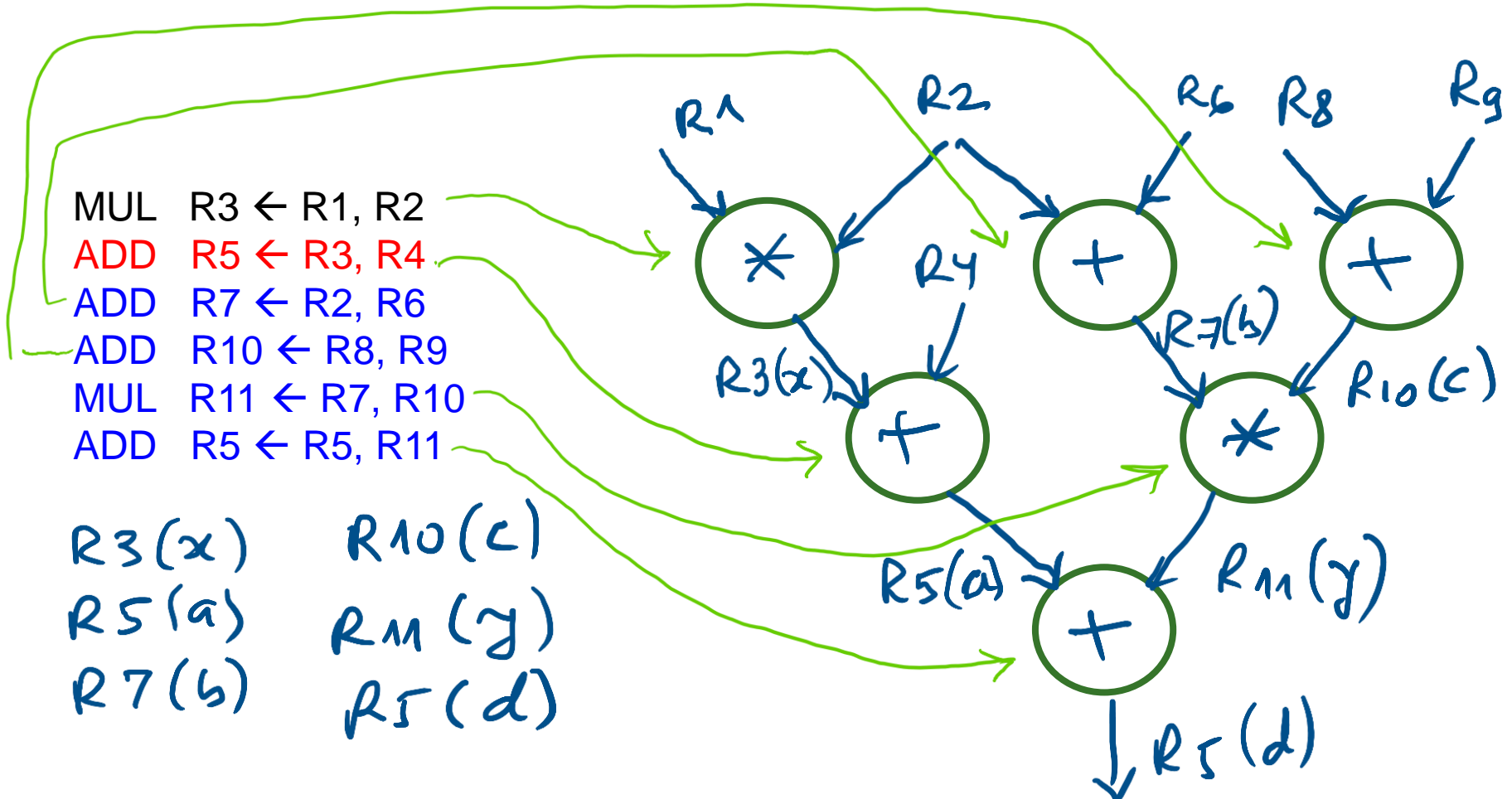
RS for MUL Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	0	b		0	c	
z						
t						



Register Alias Table

Dataflow Graph for Example



Desenați graficul fluxului de date pentru codul de mai sus

Intrebări privind proiectarea

- Când este dealocată intrarea unei stații de rezervare?
- Ar trebui ca stațiile de rezervare să fie dedicate fiecărei unități funcționale sau globale între unități funcționale?
 - Centralizat vs. Distribuit: care sunt compromisurile?
- Ar trebui stațiile de rezervare și ROB să stocheze valorile datelor sau ar trebui să existe un fișier registru fizic centralizat în care să fie stocate toate valorile datelor?
 - Care sunt compromisurile?
- Timp: exact când își transmite o instrucțiune eticheta?
- Multe alte opțiuni de design pentru motoarele OoO

Recall: Our Exercise (We Did This!)

MUL R3 \leftarrow R1, R2

ADD R5 \leftarrow R3, R4

ADD R7 \leftarrow R2, R6

ADD R10 \leftarrow R8, R9

MUL R11 \leftarrow R7, R10

ADD R5 \leftarrow R5, R11

Pipeline

F	D	E	W
---	---	---	---

ADD

F	D	E	E	E	E	W
---	---	---	---	---	---	---

MUL

F	D	E	E	E	E	E	E	E	E	W
---	---	---	---	---	---	---	---	---	---	---

- Presupunem ADD (execuție în 4 cicluri),
- MUL (execuție în 6 cicluri)
- Să presupunem un sumator și un multiplicator
- Câte cicluri ?
 - într-o mașină fără pipeline: 50 de cicluri ($4 \cdot 7 + 2 \cdot 11$)
 - într-o mașină cu pipeline de expediere în comandă, cu excepții imprecise (fără expediere și redirectionare)
 - într-o mașină de expediere ieșită din comandă, excepții imprecise (redirectionare)

For You: An Exercise, w/ Precise Exceptions

MUL $R3 \leftarrow R1, R2$

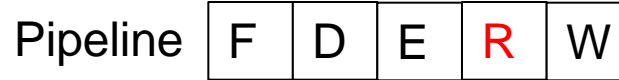
ADD $R5 \leftarrow R3, R4$

ADD $R7 \leftarrow R2, R6$

ADD $R10 \leftarrow R8, R9$

MUL $R11 \leftarrow R7, R10$

ADD $R5 \leftarrow R5, R11$



Presupuneți

- ADD (execuție în 4 cicluri),
- MUL (execuție în 6 cicluri)
- Să presupunem un adunător și un multiplicator
- Câte cicluri
 - într-o mașină cu conducte de expediere în comandă cu buffer de recomandă (fără redirectionare și redirectionare completă)
 - într-o mașină de expediere în afara comenzii canalizată cu buffer de recomandă (redirectionare completă)

Out-of-Order Execution with Precise Exceptions

- Idee: Utilizați un buffer de reordonare pentru a reordona instrucțiunile înainte de a le trimite la starea arhitecturală
- O instrucțiune actualizează RAT atunci când încheie execuția
 - De asemenea, numit fișier de registru frontend
- O instrucțiune actualizează un fișier separat de registru arhitectural atunci când se retrage
 - adică atunci când este cel mai vechi din mașină și a finalizat execuția
 - Cu alte cuvinte, fișierul de registru arhitectural este întotdeauna actualizat în ordinea programului
- Cu o excepție: resetati pipeline, copiați fișierul registru arhitectural în fișierul registru frontend

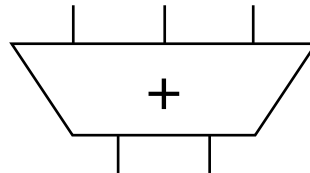
Recall: Our Initial OoO Machine

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		3
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Register Alias Table

RS for ADD Unit

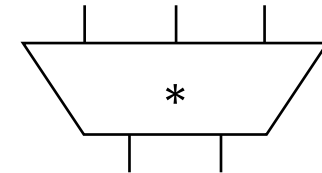
	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a						
b						
c						
d						



Tag Value

RS for MUL Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x						
y						
z						
t						



Tag Value

Add Arch Reg File & ROB for Precise Exceptions

Reorder Buffer (ROB)

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		3
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

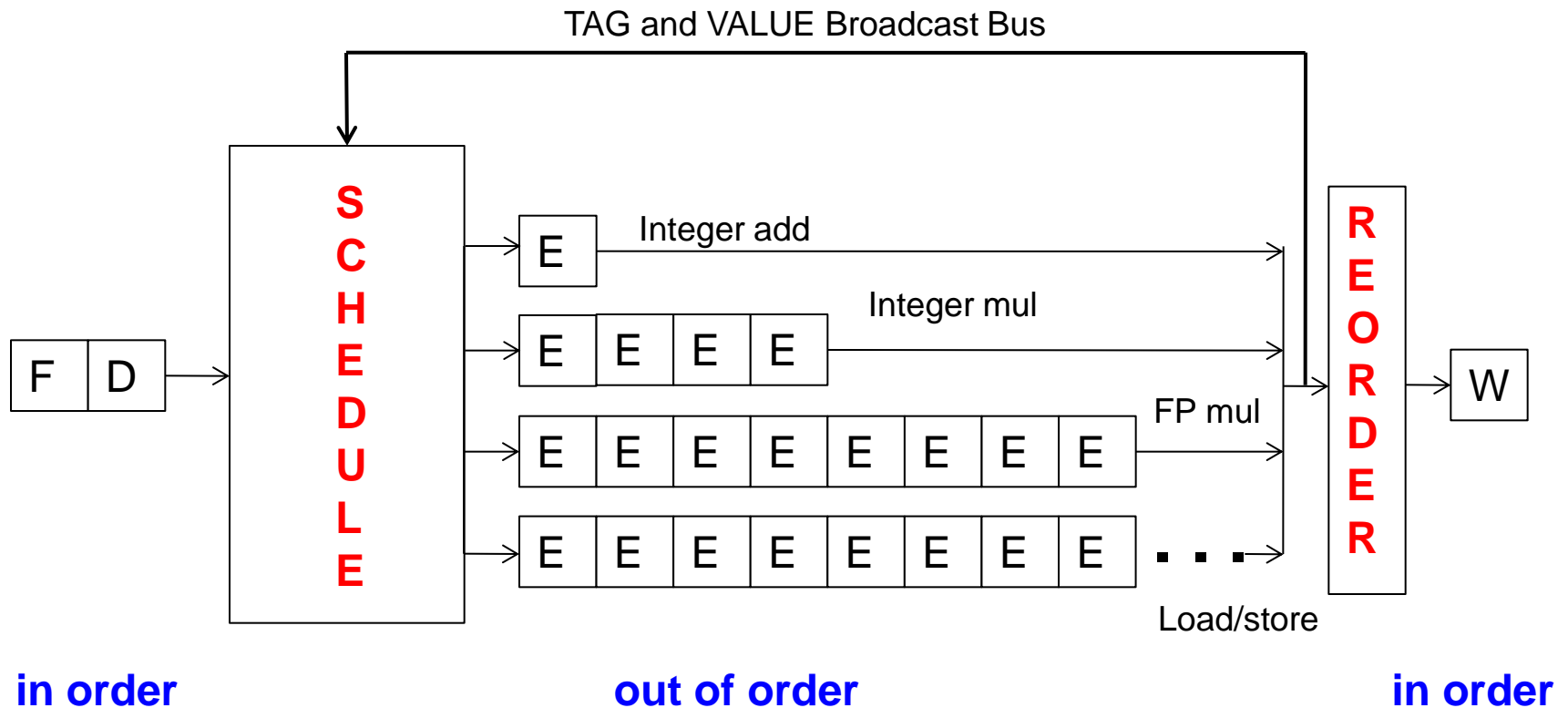
Frontend Register File

Entry 0				
Entry 1				
Entry 2				
Entry 8				
Entry 13				
Entry 14				
Entry 15				

Register	Value
R1	1
R2	2
R3	3
R4	4
R5	5
R6	6
R7	7
R8	8
R9	9
R10	10
R11	11

Architectural Register File

Out-of-Order Execution with Precise Exceptions



- 1: stații de rezervare (fereastra de programare)
- 2: Reordonare (buffer de reordonare, denumită fereastră de instrucțiuni sau fereastră activă)

One Issue: Value Replication All Over the Place

RS for ADD Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a						
b						
c						
d						

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		3
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

Frontend Register File

Reorder Buffer (ROB)

Entry 0				
Entry 1				
Entry 2				
Entry 8				
Entry 13				
Entry 14				
Entry 15				

RS for MUL Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x						
y						
z						
t						

Register	Value
R1	1
R2	2
R3	3
R4	4
R5	5
R6	6
R7	7
R8	8
R9	9
R10	10
R11	11

Architectural Register File

Getting Rid of Replicated Values

**Pointers
to PRF**

Register	PR
R1	18
R2	13
R3	10
R4	22
R5	14
R6	19
R7	17
R8	20
R9	3
R10	4
R11	1

**Frontend
Register Map**

PR	Value
PR1	1
PR2	2
PR3	3
PR4	4
PR5	5
PR6	6
PR7	7
PR8	8
PR9	9
PR10	10
PR11	11
PR12	12
PR13	13
PR14	14
PR15	15
PR16	16
PR17	17
PR18	18
PR19	19
PR20	20
PR21	21
PR22	22

**Physical Centralized
Register Value
File Storage**

Reorder Buffer (ROB)

Entry 0				
Entry 1				
Entry 2				
Entry 8				
Entry 13				
Entry 14				
Entry 15				

**Pointers
to PRF**

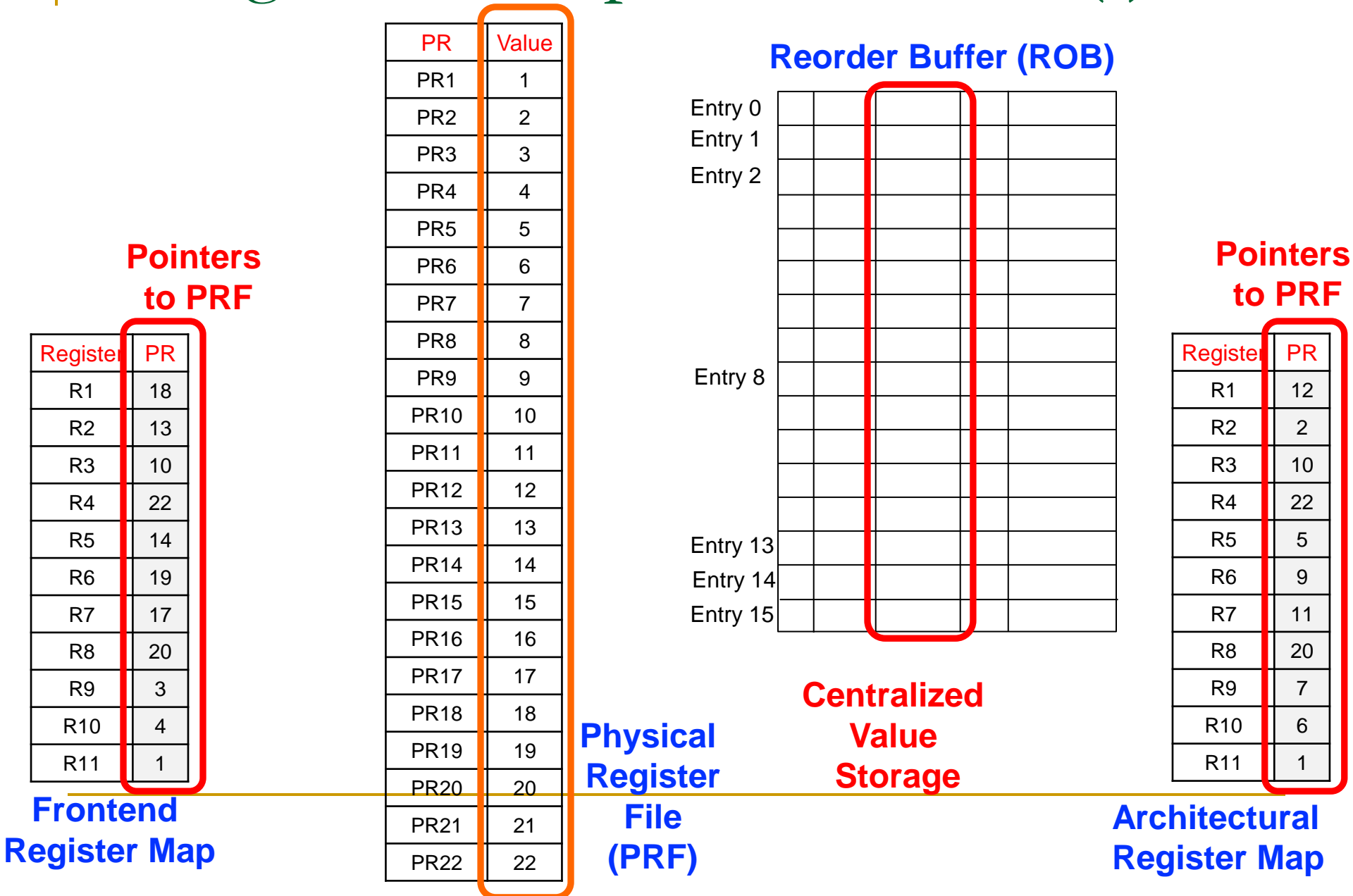
Register	PR
R1	12
R2	2
R3	10
R4	22
R5	5
R6	9
R7	11
R8	20
R9	7
R10	6
R11	1

**Architectural
Register Map**

Modern OoO Execution w/ Precise Exceptions

- Cele mai multe procesoare moderne folosesc următoarele
- Reordoneaza bufferul pentru a sprijini retragerea în ordine a instrucțiunilor
- Un singur fișier de registru (RF fizic) pentru a stoca toate registrele
 - Atât registrele speculative, cât și cele arhitecturale
 - INT și FP sunt încă separate
- Două registre de mapare stochează indicatorii către RF fizic
 - Maparea registrului viitor/frontend → folosită pentru redenumire
 - Maparea registrului arhitectural → folosită pentru menținerea stării precise
- Acest design evită replicarea valorii în RS, ROB etc.

Getting Rid of Replicated Values (I)



Getting Rid of Replicated Values (II)

At Decode/Rename: Allocate DestPR to Dest Reg

At Decode/Rename: Read and Update Frontend Register Map

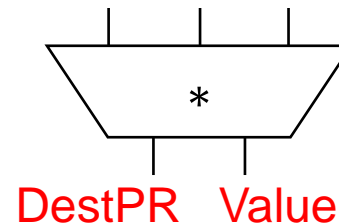
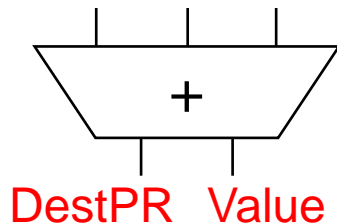
RS for ADD Unit

	Source 1	Source 2
	PR	PR
a		
b		
c		
d		

RS for MUL Unit

	Source 1	Source 2
	PR	PR
a		
b		
c		
d		

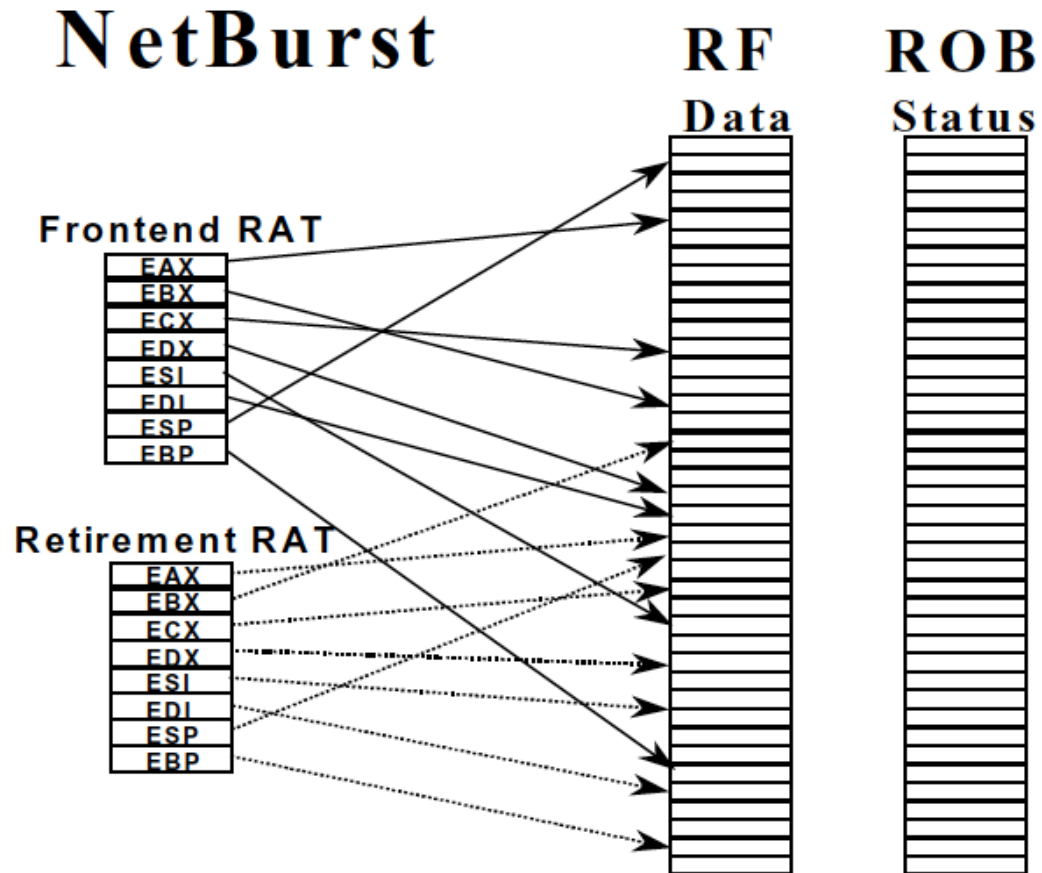
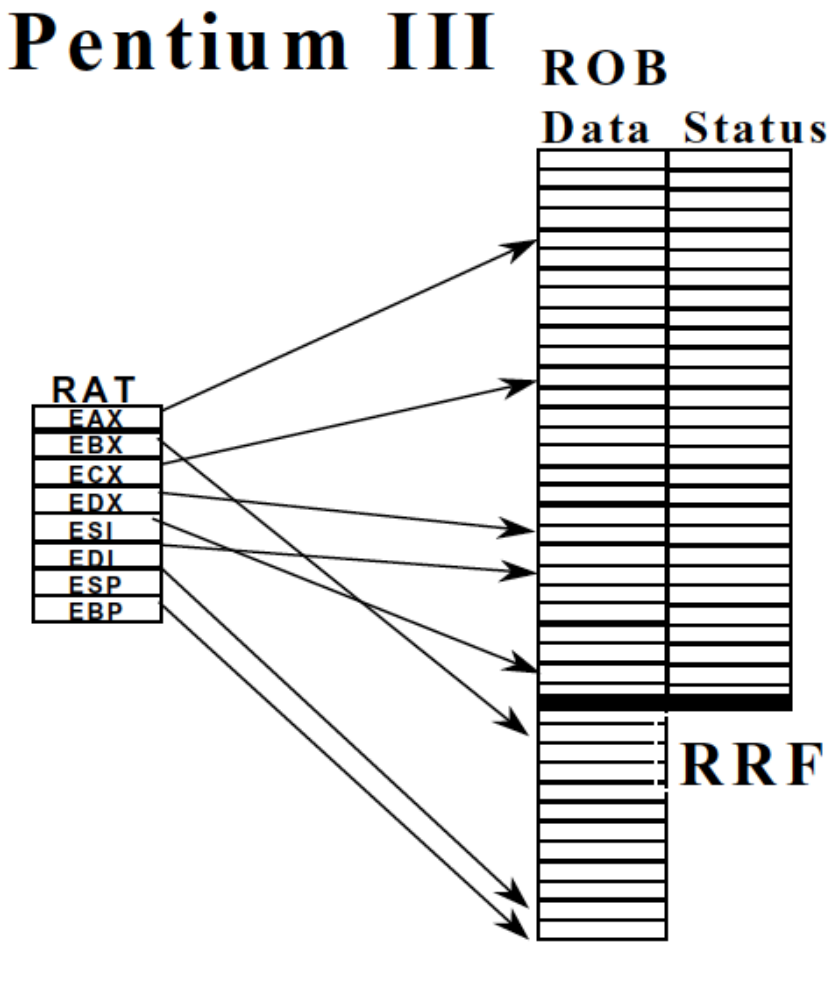
Before Execution: Access Physical Register File to Get Source Values



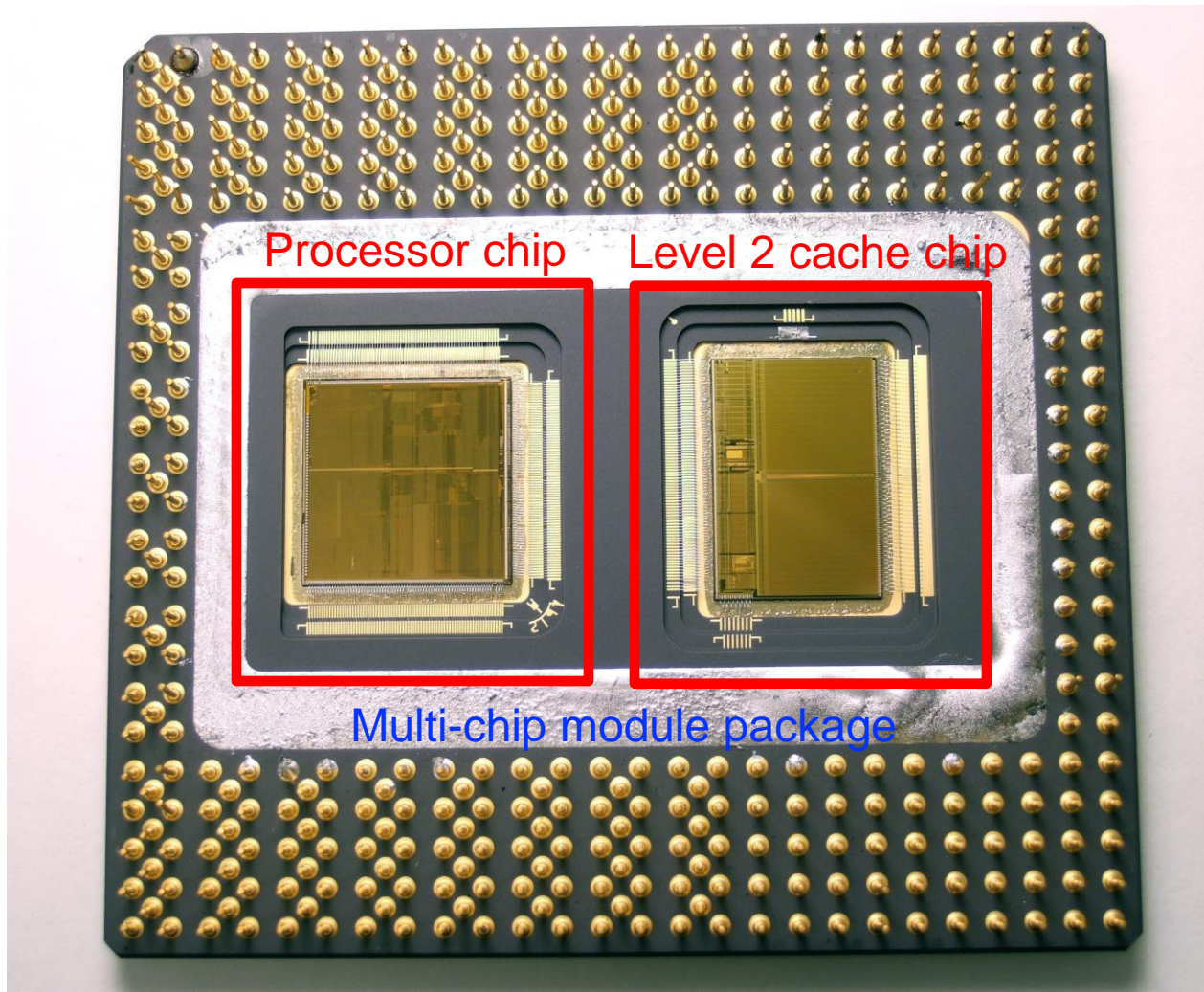
After Execution: Access Physical Register File to Write Result Values

At Retirement : Update Architectural Register Map with DestPR

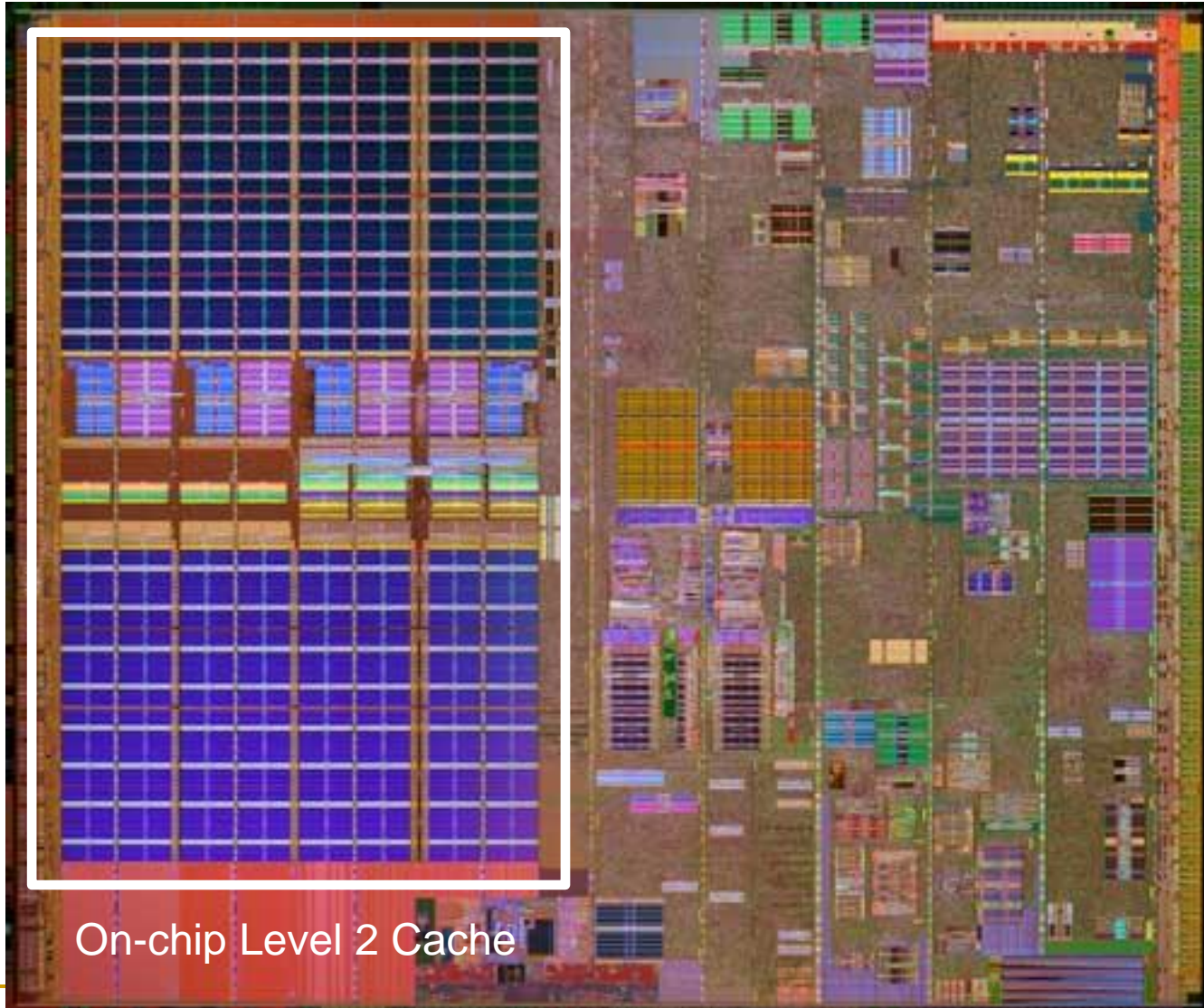
An Example from Modern Processors



Intel Pentium Pro (1995)



Intel Pentium 4 (2000)



Enabling OoO Execution, Revisited

- Leagă consumatorul unei valori de producător
 - Redenumirea înregistrării: asociați o „etichetă” cu fiecare valoare de date
- Tamponează instrucțiunile până când sunt gata
 - Introduceți instrucțiuni în stațiile de rezervare după redenumire
- Urmăriți gradul de pregătire a valorilor sursei unei instrucțiuni
 - Difuzati „eticheta” atunci când valoarea este produsă
 - Instrucțiunile compară „etichetele sursă” cu eticheta de difuzare → dacă se potrivesc, valoarea sursă devine gata
- 4. Când toate valorile sursă ale unei instrucțiuni sunt gata, trimiteți instrucțiunea către unitatea funcțională (FU)
- Treziți și selectați/planificați instrucțiunea

Summary of OOO Execution Concepts

- Redenumirea registrului elimină dependențele false, permite legătura dintre producător și consumatori
- Buffering-ul în stațiile de rezervare permite conducerii să se deplaseze pentru instrucțiuni independente
- Difuzarea etichetelor permite comunicarea (de disponibilitate a valorii produse) între instrucțiuni
- Trezire și selectare permite expedierea în afara comenzii

OOO Execution: Restricted Dataflow

- Un motor nefuncțional construiește dinamic graficul fluxului de date al unei părți a programului
 - care piesa?
- Graficul fluxului de date este limitat la fereastra de instrucțiuni
 - Fereastra de instrucțiuni: toate instrucțiunile decodificate, dar care nu au fost încă retrase
- O putem face pentru tot programul?
- De ce ne-am dori?
- Cu alte cuvinte, cum putem avea o fereastră mare de instrucțiuni?
- O putem face eficient cu algoritmul lui Tomasulo?

Întrebări de meditat

- De ce este benefică execuția OoO?
 - Ce se întâmplă dacă toate operațiunile durează un singur ciclu?
 - Toleranță la latență: execuția OoO tolerează latența operațiilor cu mai multe cicluri prin executarea de operațiuni independente concomitant
- Ce se întâmplă dacă o instrucțiune durează 1000 de cicluri?
 - Cât de mare de fereastră de instrucțiuni avem nevoie pentru a continua decodarea?
 - Câte cicluri de latență poate tolera OoO?
 - Ce limitează scalabilitatea toleranței la latență a algoritmului lui Tomasulo?
 - Dimensiunea ferestrei de instrucțiuni: câte instrucțiuni decodificate, dar neretrase încă, puteți păstra în mașină.

Exercise Continued

F D 1 2 3 4 5 6 W

F D - - - - - D 1 2 3 4 W

F - - - - - D 1 2 3 4 W

F D 1 2 3 4 W

F D - - - - D 1 2 3 4 5 6 W

F - - - - - D

D 1 2 3 4 W



Execution timeline w/ scoreboarding

31 cycles

F D 1 2 3 4 5 6 W

F D

F

→ E, 1 2 3 4 W

D 1 2 3 4 W

F D 1 2 3 4 W

F D

F

→ 1 2 3 4 5 6 W

D

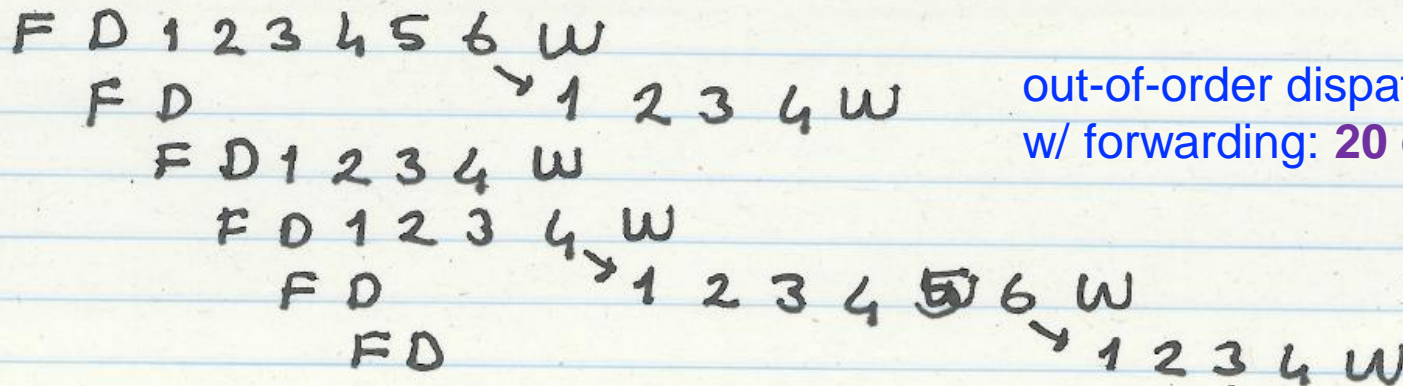
→ 1 2 3 4 W

in-order-dispatch pipelined machine
w/ forwarding: **25 cycles**

25 cycles

Exercise Continued

MUL R3 \leftarrow R1, R2
 ADD R5 \leftarrow R3, R4
 ADD R7 \leftarrow R2, R6
 ADD R10 \leftarrow R8, R9
 MUL R11 \leftarrow R7, R10
 ADD R5 \leftarrow R5, R11

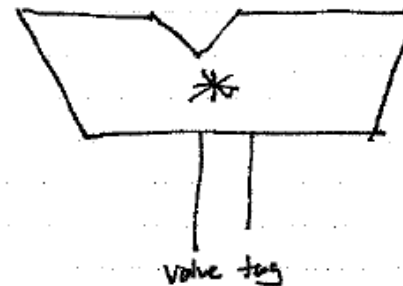
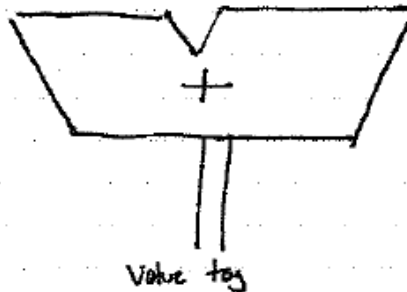
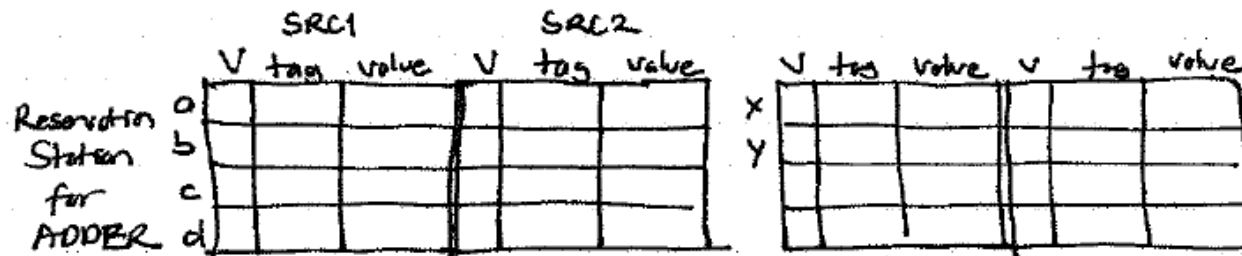
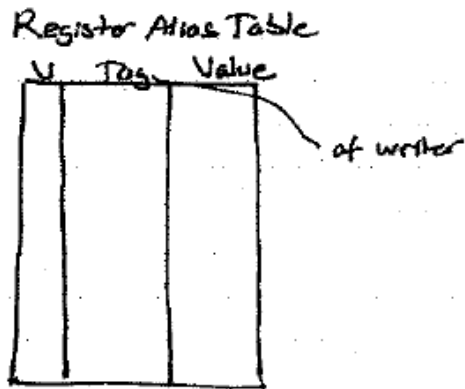


out-of-order dispatch pipelined machine
 w/ forwarding: **20 cycles**

Tomasulo's algorithm + full forwarding

20 cycles

How It Works



Assume
adder &
multiplier have
separate
buses

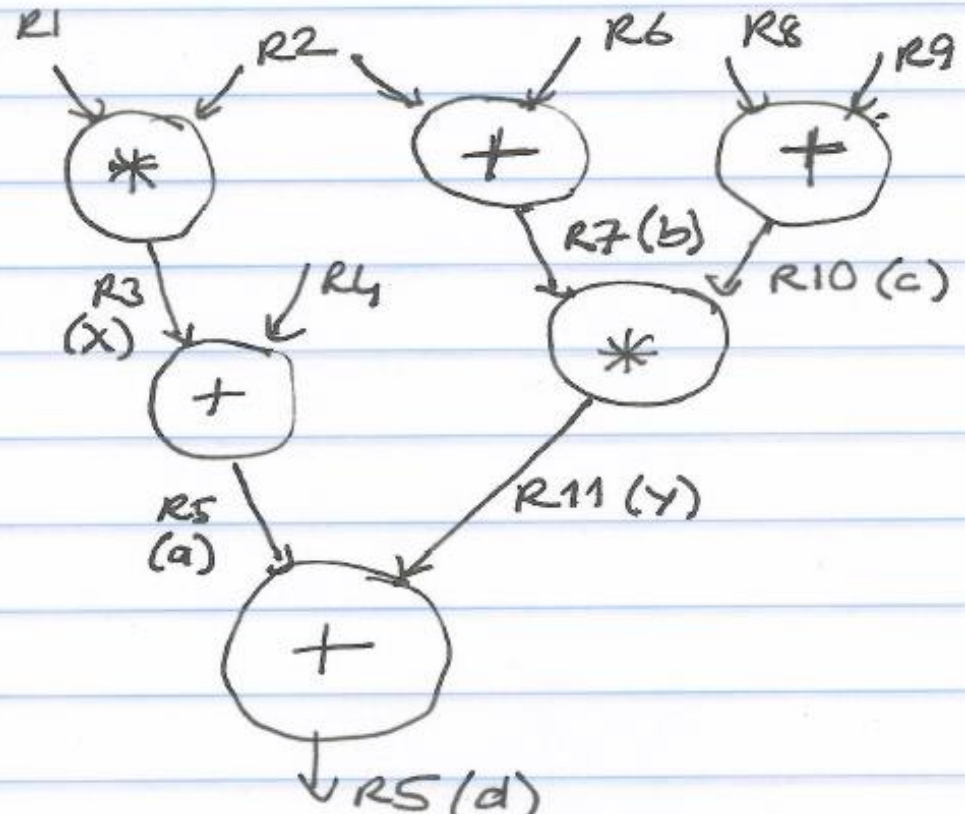
Corresponding Dataflow Graph (Reverse Engineered)

MUL R1, R2 → R3 (x)
ADD R3, R4 → R5 (a)
ADD R2, R6 → R7 (b)
ADD R8, R9 → R10 (c)
MUL R7, R10 → R11 (y)
ADD R5, R11 → R5 (d)

Dataflow graph

Nodes: operations performed by the instruction

Arcs: tags in Tomasulo's algorithm



We can “easily” reverse-engineer the dataflow graph of the executing code!









Exemplu care evidentiaza diferite structuri elementare

Consideram adunarea a doi vectori

$$Z = X + Y$$

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

$$z_i = x_i + y_i$$

Fiecare componenta are mantisa si exponent (caracteristica)

$$x_i = x_{ic} \cdot x_{im}$$

Operatia de adunare are 4 operatii elementare organizate in pipeline

- Compararea caracteristicilor
- Deplasarea mantisei (daca este cazul)
- Adunarea mantiselor
- Normalizarea rezultatului

Aceste operatii sunt independente – structura pipeline

Consideram aceste operatii dureaza $\sim t$ (pentru simplificare)