CURS 1 — 26.02.2019

Cap I: Aritmetica în sistemele de calcul (Computer Arithmetics)

1.1. Introducere

→ Algoritmul lui Robertson

• operanzi în C_2 și ne interesează să îl explicităm pe B: (biții lui B)

$A * B$

↑ multiplicand ↑ multiplier

$$B = (b_{n-1} b_{n-2} \dots b_i \dots b_1 b_0)_{C_2}$$

$$b_i \in \mathbb{B} = \{0, 1\} \text{ cu } i = \overline{0, n-1}$$

Formula lui Robertson: $$B_{C_2} = \left(-b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i\right)$$

$i = 0$: $b_0 \times A \times 2^0$

$+ \; b_1 \times A \times 2^1$

$+ \; \dots\dots$

$+ \; b_{n-2} \times A \times 2^{n-2}$

$- \; b_{n-1} \times A \times 2^{n-1}$

P

Lucru care se întâmplă în algoritmul lui Robertson

- Fiecare 1 are 2 clocki (shiftare + adunare), fiecare 0 are 1 clock => => greu pt 1111 1111, de ex. => => Booth
- Un pas în acest algoritm: $(b_{i-1} - b_i) * 2^i * A$
- Se consideră $b_{-1} = 0$
- $i = 0$: $(b_{-1} - b_0) \times 2^0 \times A$

  $i = 1$: $+ (b_0 - b_1) \times 2^1 \times A$

  $i = 2$: $+ (b_1 - b_2) \times 2^2 \times A$

  (...)

  $i = j-1$: $+ (b_{j-2} - b_{j-1}) \times 2^{j-1} \times A$

  $i = j$: $+ (b_{j-1} - b_j) \cdot 2^j \times A$

→ Algoritmul lui Booth

b_i	b_{i-1}	Op
0	0	0
0	1	+A
1	0	−A
1	1	0

...

$i = n-2$: $+\left(b_{n-3} - b_{n-2}\right) \times 2^{n-2} \times A$

$i = n-1$: $+\left(b_{n-2} - b_{n-1}\right) \times 2^{n-1} \times A$

- Termenii se vor simplifica
- Pentru termenii $i = j$ și $i = j+1$:

$$-b_j \times 2^j + b_j \times 2^{j+1} = b_j \times 2^j (-1+2) = b_j \times 2^j$$

- Practic se ajunge la: $P = \left(\sum_{i=0}^{n-2} b_i \times 2^i - b_{n-1} \times 2^{n-1}\right) \times A$

adică tot formula lui Robertson. Diferența e în faptul că Booth e mai eficient (mai puțini clockei)

- Ex: $B = 0\,1\,0\,1\,0\,1\,0\,1$ => 4 operații Robertson, dar 8 operații pt Booth: $B' = 0\,1\,0\,1\,0\,1\,0\,1\,|\,0$ (1 $\bar{1}$ 1 $\bar{1}$ 1 $\bar{1}$ 1 $\bar{1}$) =>

=> Booth modificat

- Ex: $B = 1111\,0000\,|\,0$ – Robertson = 4

$B' = 000\bar{1}\,0000$ – Booth = 1

→ Algoritmul lui Booth modificat

- combină Robertson și Booth, verifică cazurile de 1 sau 0 izolat

b_{i+1}	b_i	F	B''	F'
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	$\bar{1}$	0
1	0	1	0	1
1	1	0	$\bar{1}$	1
1	1	1	0	1

cazul de 01 de la Booth (+)

← 1 izolat

← nu știm B'' = referă la ce s-a făcut după, dar acum încă nu intrăm în pașii de lucru

F = starturile ciclului din care vin

F' = starturile ciclului modificat

- Avem nevoie de un bit în plus la stânga și: $b_n = b_{n-1}$ (e egal cu c. m. semnificativ)

- Tentru 11 : $-A \times 2^i + A \times 2^{i+1} = A \times 2^i(-1+2) = A \times 2^i \Rightarrow (+)$
  (10, 01)

- Tentru 11 : $+A \times 2^i - A \times 2^{i+1} = -A \times 2^i \Rightarrow (-)$
  (01, 10)

- Exemple:

> B = 0 0 1 0 1 0 1 0 1 0 — Robertson = 4
> B' = 1 1 1 1 1 1 1 1 — Booth = 8
> B'' = 0 1 0 1 0 1 0 1 — Booth mod. = 4
> F = 0 0 0 0 0 0 0 0 0

> B = 1 1 1 1 1 0 0 0 0 0 — Robertson = 4
> B' = 0 0 0 1 0 0 0 0 — Booth = 1
> B'' = 0 0 0 1 0 0 0 0 — Booth mod. = 1
> F = 1 1 1 1 0 0 0 0 0 0

- Exercitiu pt aplicarea lui Booth modificat:

> X = -105
> Y = -73
> => P = +7665 (trb. să dea)

overflow

COUNT	OVF	A	Q[8]	Q	F	M
000	0	0000 0000	(1)	1001 0111	0	1011 0111
	+	0100 1001			1	
	0	0100 1001				
		0010 0100	1	1100 1011		
001	0	0001 0010	0	1110 0101	1	
010	0	0000 1001	0	0111 0010	1	
011	+	0100 1001			1	
	0	0101 0010				
		0010 1001	0	0011 1001		
100	0	0001 0100	1	0001 1100	1	
101	+	1011 0111			0	
		1100 1011				
		1100 1011	1	1000 1110		
110	0	1111 0010	1	1100 0111	0	
111	+	0100 1001			1	
	0	0011 1011				
		0001 1101	1	1110 0011		

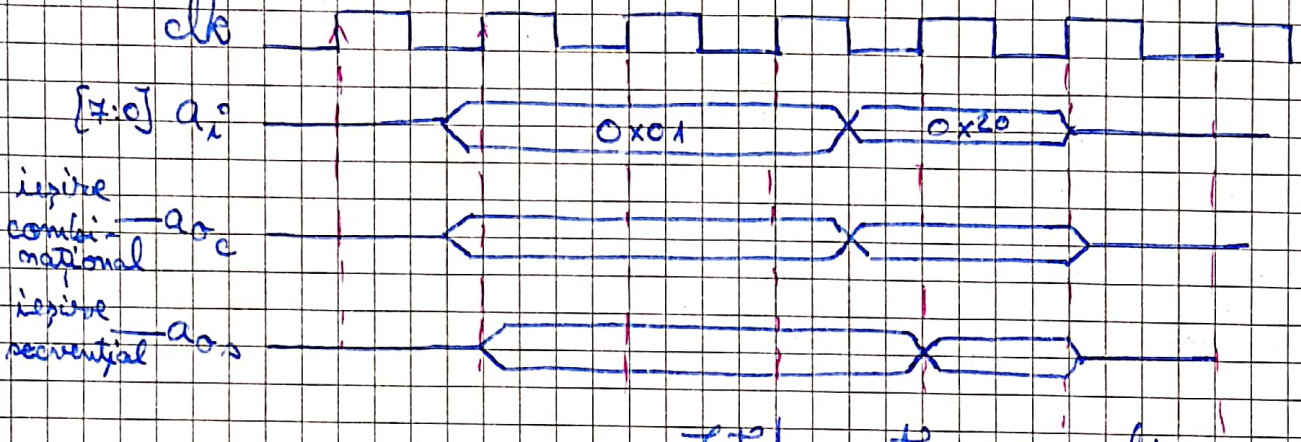
$X = 1\,1\,1\,0\,1\,0\,0\,1_{SM} = \boxed{1\,0\,0\,1\,0\,1\,1\,1}_{C_2}$

$Y = 1\,1\,0\,0\,1\,0\,0\,1_{SM} = \boxed{1\,0\,1\,1\,0\,1\,1\,1}_{C_2}$

$M = 1\,0\,1\,1\,0\,1\,1\,1 \Rightarrow -M = 0\,1\,0\,0\,1\,0\,0\,1$

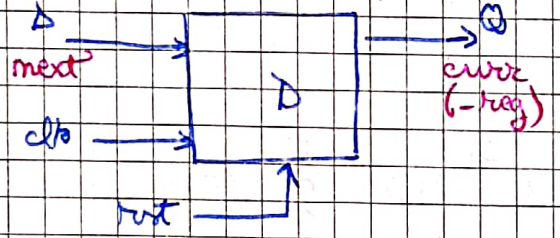
LAB 2

- Circuite combinaționale vs. secvențiale



- Circuit secvențial < Latch: activ pe palier
  Flip flop: activ pe front.

- Exemplu f.f: D ff



  - stare curentă: Q (cea stabilă, care menține semnalul pt. măcar un clock)
  - stare următoare: D

- Secvențial: always @ (posedge clk, posedge rst)
  begin
  reg <= next;
  end

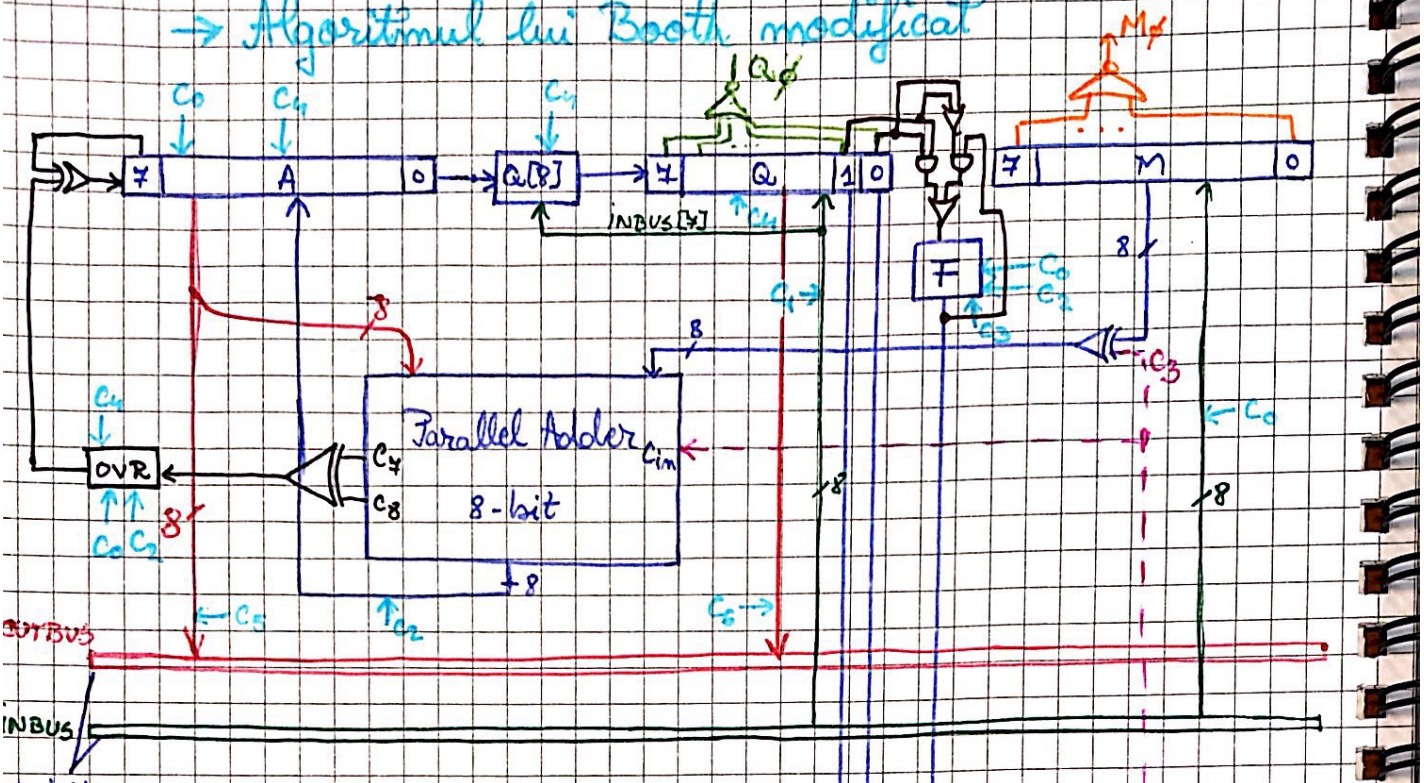
- Arhitectură cu ——— unitate de înmulțire
  ——— // ——— împărțire
  ——— // ——— memorie

CURS 2

5.03.2019

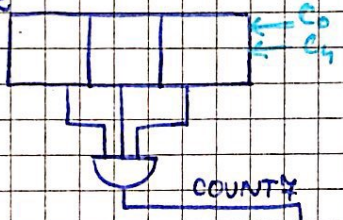
1.1. Introducere

→ Algoritmul lui Booth modificat

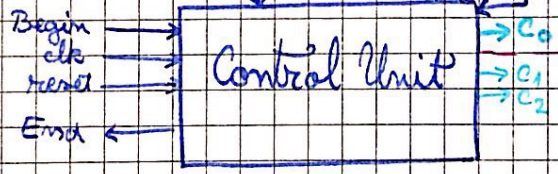


8-bit Data Buses

COUNTER

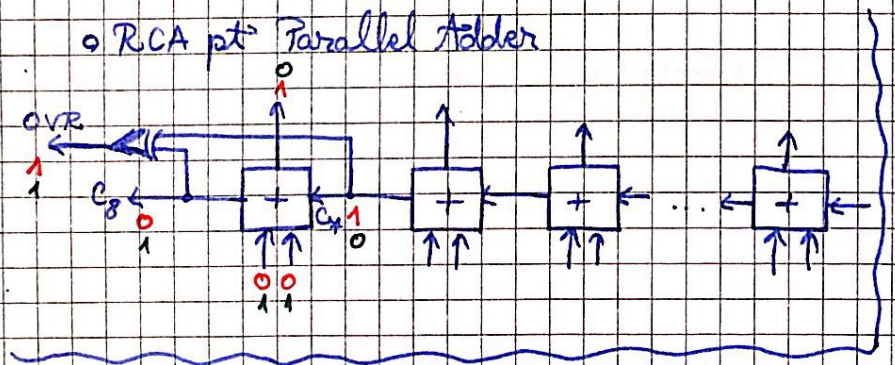


External Control Signals

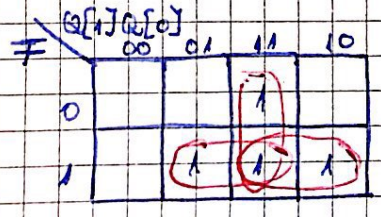


Internal Control Signals

○ RCA pt Parallel Adder



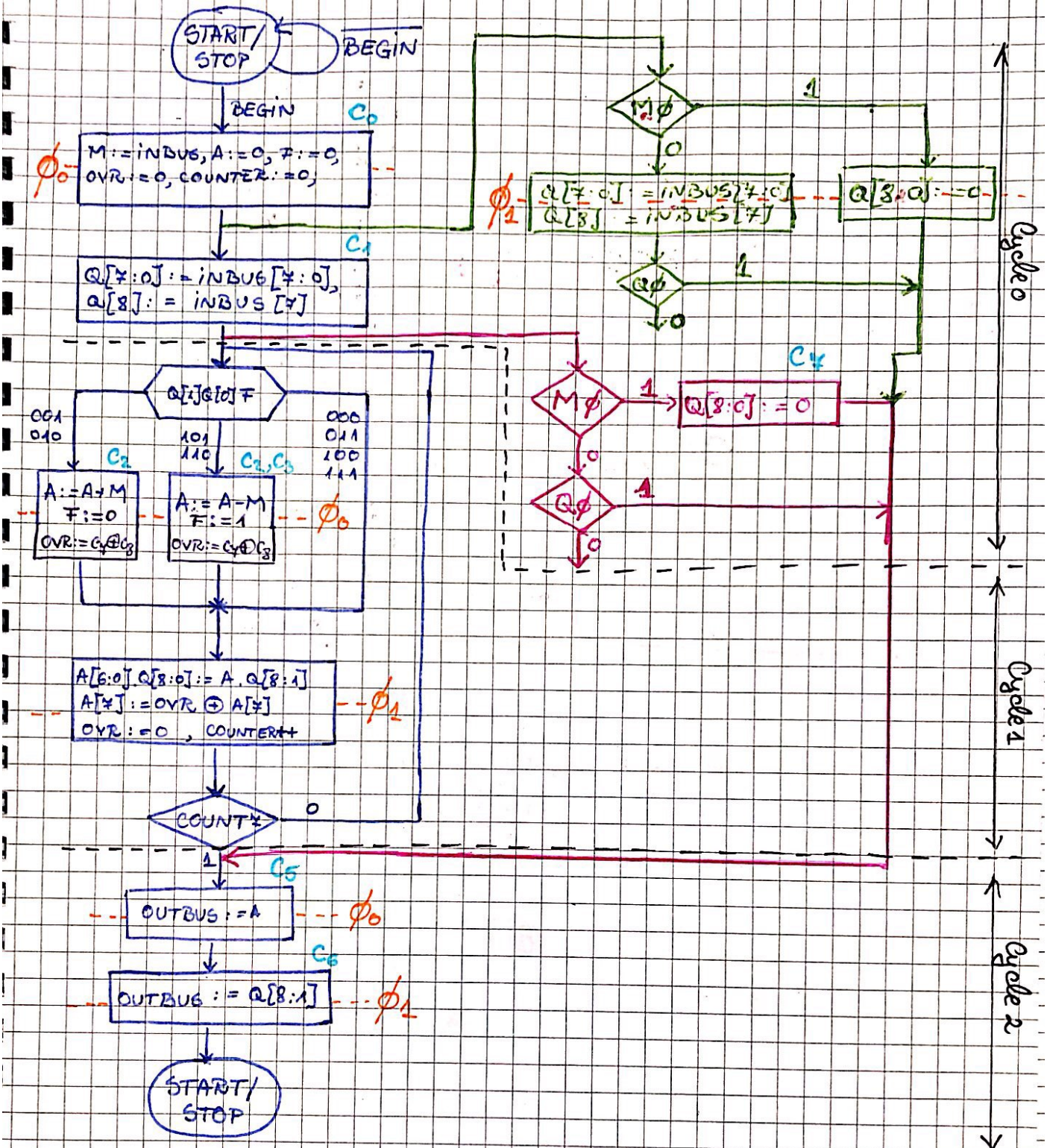
Q[1]	Q[0]	F	F'	OP
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	0
1	0	1	1	$\bar{1}$
1	1	0	1	$\bar{1}$
1	1	1	1	0



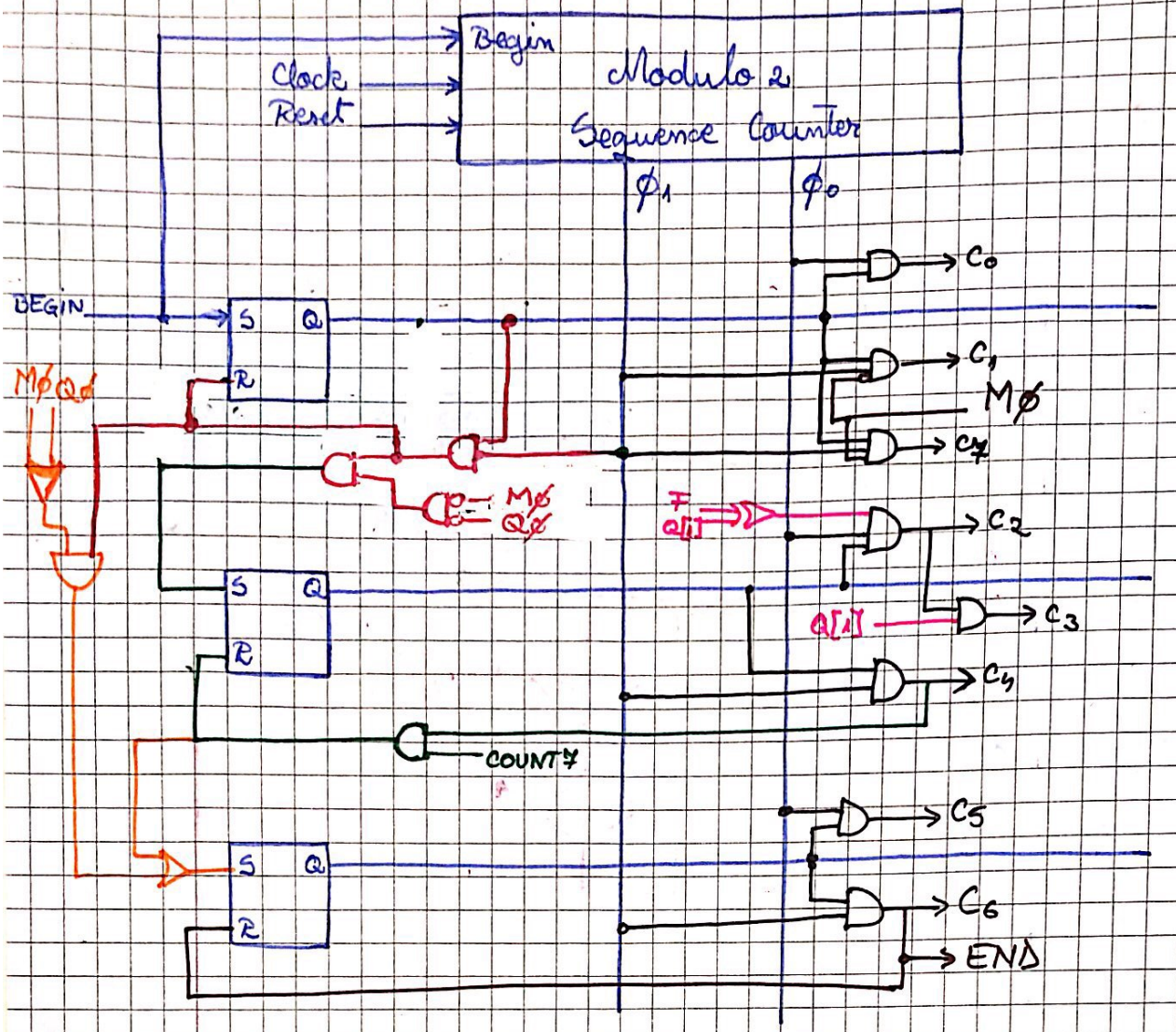
$\Rightarrow F' = Q_1 Q_0 + F(Q_0 + Q_1)$

- Q_ϕ este 1 dacă $Q_0 = 0$
- M_ϕ este 1 dacă $M = 0$

} Toate procesele aleg la început să evităm înmulțirea cu 0



Se poate implementa [illegible] sau [illegible], e mai eficient



1.2. Speeding up : higher radix

Radix -4 step i with weight 2^i

b_{i+1}	b_i	b_{i-1}	0?
0	0	0	0
0	0	1	+A
0	1	0	+A
0	1	1	+2A
1	0	0	-2A
1	0	1	-A
1	1	0	-A
1	1	1	0

$A \times 0 \times 2^i + A \times 0 \times 2^{i+1} = 0 \times 1 \times 2^i$

$+A \times 2^i + 0 \times A \times 2^{i+1} = +A \times 2^i$

$-A \times 2^i + A \times 2^{i+1} = A \times 2^i$

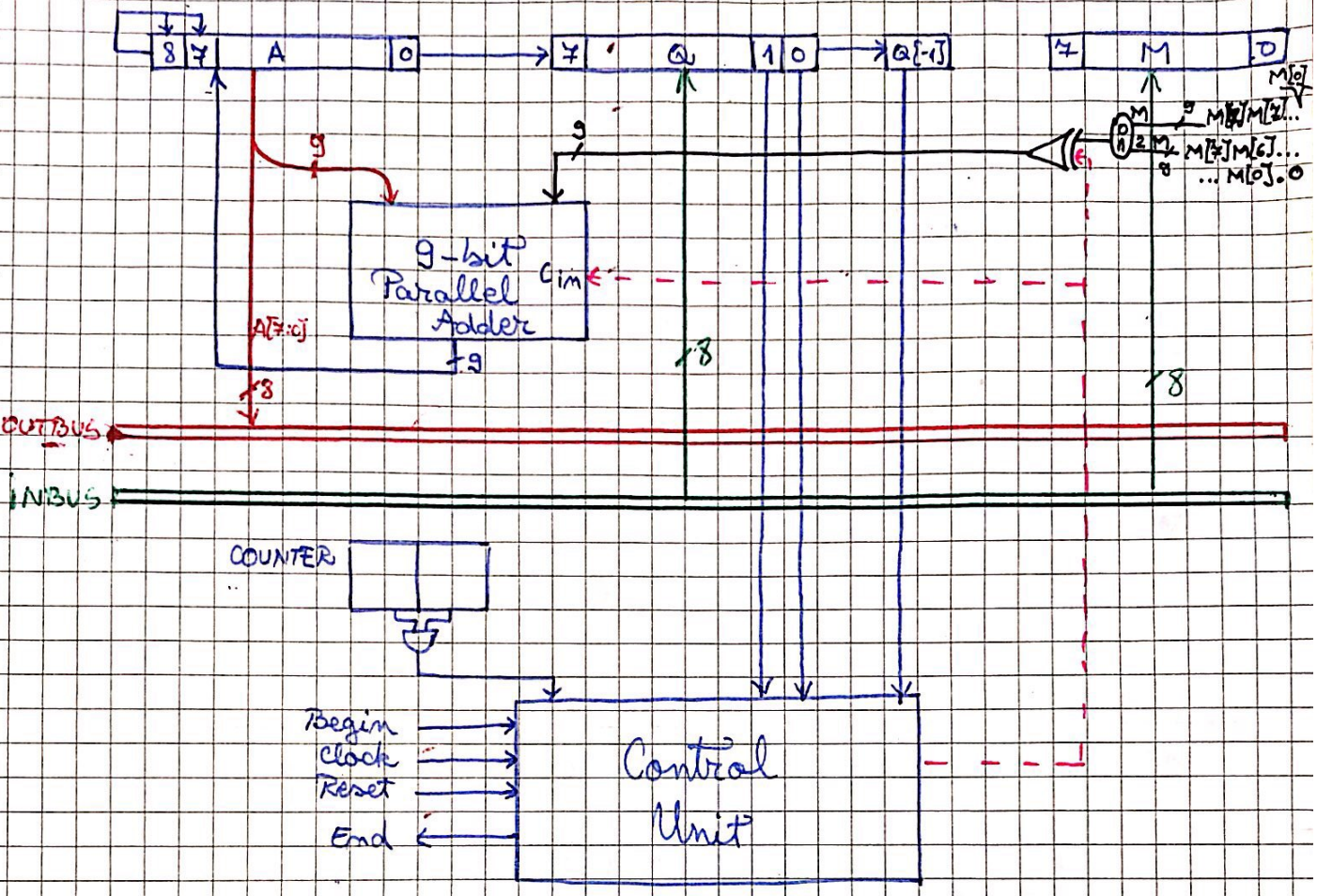
$0 + A \times 2^{i+1} = +2A \times 2^i$

$0 - A \times 2^{i+1} = -2A \times 2^i$

$+A \times 2^i - A \times 2^{i+1} = -A \times 2^i$

$-A \times 2^i + 0 = -A \times 2^i$

• Ca să salvăm clocki, trb. să putem face shiftări duble =>
=> A pe 9 biți.



LAB 2

- M = 13 = 0 1101 SM
- Q = 7 = 0 111 SM

Dorim să facem înmulțirea în Radix-2 pt Booth modificat

$M = -11 = 1\,1011_{SM} = 1\,0101_{C_2}$; $-M = 0\,1011$

$Q_1 = 5 = 0\,0101_{SM} = 00101_{C_2}$

COUNT	OVR	A	Q[5]	Q	F	M
000		00000 + 10101 = 10101	0	0 0101	0	10101
		10101 + 11010	1	0 0010	0	
001		11101 + 10101 = 10010	0	1 0001	0	
010	1	11001	0	0 1000	0	
011		11100	1	0 0100	0	
100		11110	0	1 0010		

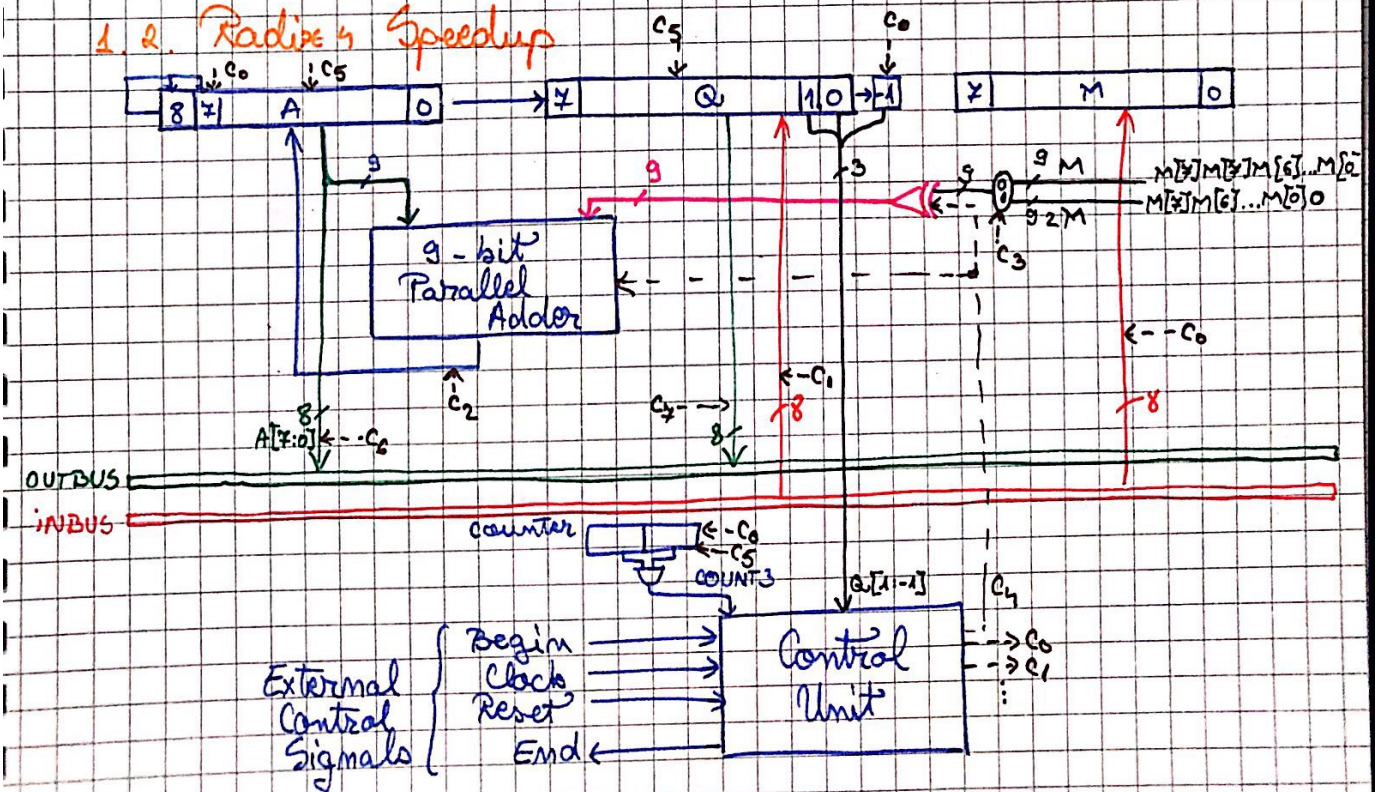
$11110\,01001 = 1\,000110111 = 1+2+4+16+32 =$

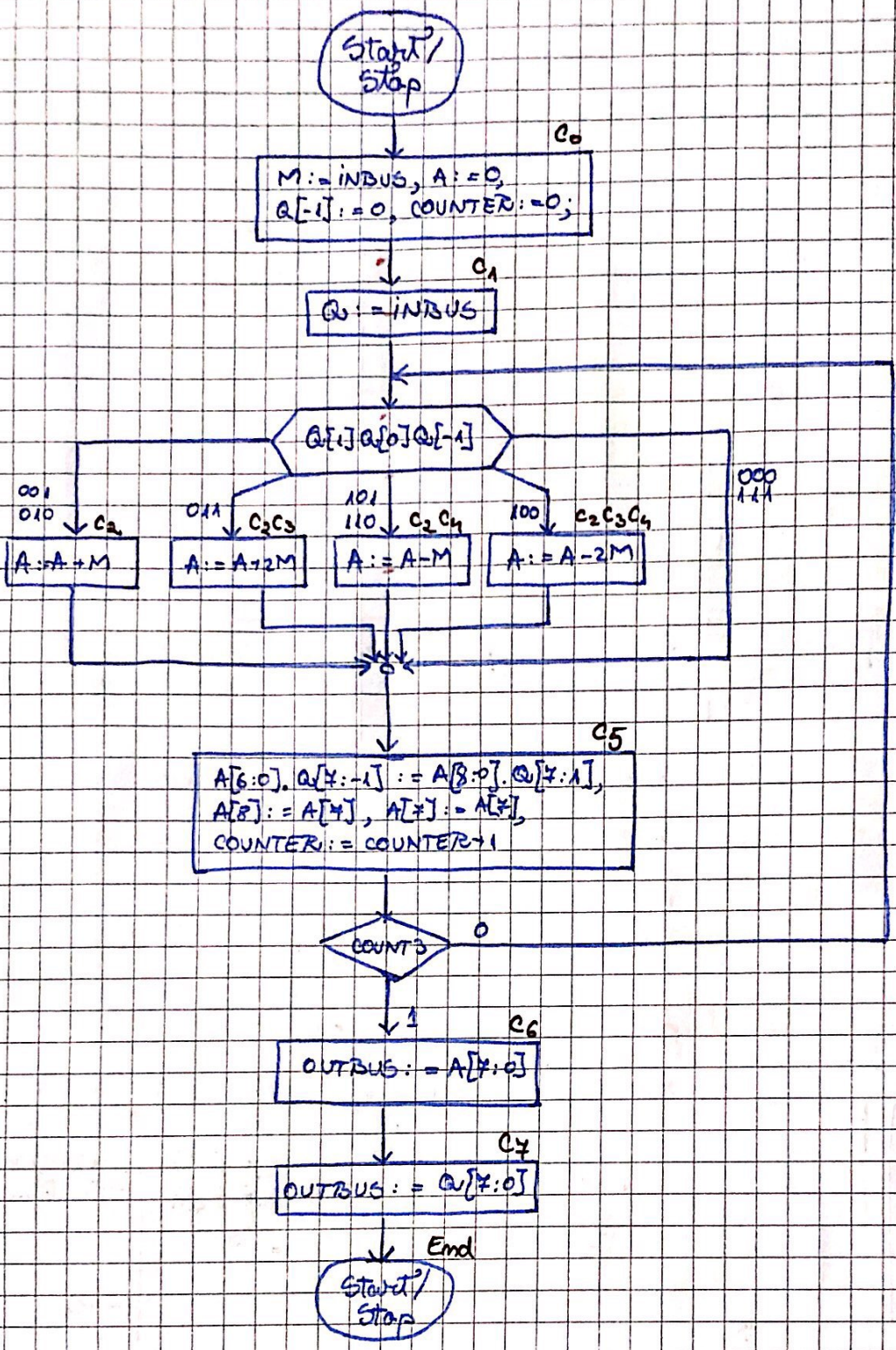
$= 3 + 20 + 32 = -55$

CURS 3

12.03.2019

1.2. Radix 4 Speedup





- Sinteza cu Sequence Counter a ordinogramei de mai sus

- Exemplu:

$X = -107 = 1110\ 1011_{SM} = 1\ 001\ 0101_{C_2}$

$Y = +92 = 1101\ 1100_{SM} = 1010\ 0100_{C_2}$

$P = -107 \times -92$

-214

963

9844

$M = 1\ 1001\ 0101$ (M pe 9 biți)

$-M = 0\ 0110\ 1011$

$2M = 1\ 0010\ 1010$

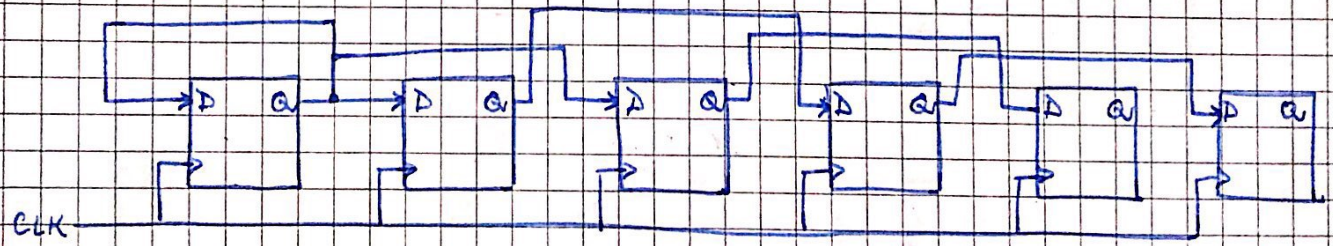
$-2M = 0\ 1101\ 0110$

COUNT	A	Q	Q[-1]	M
00	000000000 000000000	1010 0100 0010 1001	0	1001 0101 ← shiftare dublă
01	+110010101 110010101 111100101	0100 1010	0 +M -2M	
10	+010110110 010111011 000101110	1101 0010	1 -M	
11	+001010011 010011001 001001110	0111 0100		

$P = 0010\ 0111\ 0011\ 1010\ 0_{C_2=SM} = +9844$

↑ nr. pozitiv

Shiftarea aritmetică:



1.3.

1.3. Radix-8 Speedup

Q[2]	Q[1]	Q[0]	Q[-1]	Op
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	2
0	1	0	0	2
0	1	0	1	3
0	1	1	0	3
0	1	1	1	4
1	0	0	0	-4
1	0	0	1	-3
1	0	1	0	-3
1	0	1	1	-2
1	1	0	0	-2
1	1	0	1	-1
1	1	1	0	-1
1	1	1	1	0

$\rightarrow +M \cdot 2^i$

$\rightarrow -M \cdot 2^i + M \cdot 2^{i+1} = M \cdot 2^i$

$\rightarrow +M \cdot 2^{i+1} = 2M \cdot 2^i$

$\rightarrow -M \cdot 2^{i+1} + M \cdot 2^{i+2} = 2M \cdot 2^i$

$\rightarrow +M \cdot 2^i - M \cdot 2^{i+1} + M \cdot 2^{i+2} = 3M \cdot 2^i$

2^{i+2} 2^{i+1} 2^i

- Exemplu: $\overset{ni}{A}$ e pe 10 biți. Pt exemplul anterior o să analizăm grupele: 1 1 0 1 0 0 1 0 0 0

CNT	A	Q[8]	Q	Q[-1]	M
00	00 0000 0000	1	1010 0100	0	1001 0101
+	01 1010 1100			-4M	
	01 1010 1100				
	00 0011 0101	1	0011 0100	1	
01 +	01 0100 0001			-3M	
	01 0111 0110				
	00 0010 1110	1	1010 0110	1	
10 +	00 0110 1011			-M	
	00 1001 1001				
	00 0010 0011	0	0111 0100	1	

Counter-ul merge până la nr. de grupe în care împărțim Q-ul.

Extindem M pe 10 biți:

M = 11 1001 0101

-M = 000 110 1011

2M = 11 0010 1010

-2M = 0011 0101 10

3M = 1010 1111 11

-3M = 010 1000 0001

4M = 1001 0101 00

-4M = 0110 1011 00

1.4. Division Algorithms

5771 : 135 = 42 (=) Dividend = Divisor * Quotient + Remainder

540

= 371

270

101

5771 = 1011 0100 0101 1

135 =

1.4.1. Restoring Division (întregi fără semn)

COUNT	A	Q	M	
000	001 0110 1	000 1011 0	1000 0111	
	- 1000 0111			
	1010 0110	0001 0110		
	+ 1000 0111			
	0010 1101			
	010			← shiftare la stânga

CURS 4

1.4.1. Restoring Division

5771 | 135
101 | 42 ← semnul scăderii

nr. 5771

COUNT	S	A	Q	M
000	0	0 0 1 0 1 1 0 1	0 0 0 1 0 1 1 0	1 0 0 0 0 1 1 1
	−	1 0 0 0 0 1 1 1		
	1	1 0 1 0 0 1 1 0	0 0 0 1 0 1 1 0	
	+	1 0 0 0 0 1 1 1		← restaurare
	0	0 0 1 0 1 1 0 1		← shiftare la stânga
	0	0 1 0 1 1 0 1 0	0 0 1 0 1 1 0 0	
001	−	1 0 0 0 0 1 1 1		
	1	1 1 0 1 0 0 1 1	0 0 1 0 1 1 0 0	
	+	1 0 0 0 0 1 1 1		
	0	0 1 0 1 1 0 1 0		
	0	1 0 1 1 0 1 0 0	0 1 0 1 1 0 0 0	
010	−	1 0 0 0 0 1 1 1		
	0	0 0 1 0 1 1 0 1	0 1 0 1 1 0 0 1	
	0	0 1 0 1 1 0 1 0	1 0 1 1 0 0 1 0	
011	−	1 0 0 0 0 1 1 1		
	1	1 1 0 1 0 0 1 1	1 0 1 1 0 0 1 0	
	+	1 0 0 0 0 1 1 1		
	0	0 1 0 1 1 0 1 0		
	0	1 0 1 1 0 1 0 1	0 1 1 0 0 1 0 0	
100	−	1 0 0 0 0 1 1 1		
	0	0 0 1 0 1 1 1 0	0 1 1 0 0 1 0 1	
	0	0 1 0 1 1 1 0 0	1 1 0 0 1 0 1 0	
101	−	1 0 0 0 0 1 1 1		
	1	1 1 0 1 0 1 0 1	1 1 0 0 1 0 1 0	
	+	1 0 0 0 0 1 1 1		
	0	0 1 0 1 1 1 0 0		
	0	1 0 1 1 1 0 0 1	1 0 0 1 0 1 0 0	
110	−	1 0 0 0 0 1 1 1		
	0	0 0 1 1 0 0 1 0	1 0 0 1 0 1 0 1	
	0	0 1 1 0 0 1 0 1	0 0 1 0 1 0 1 0	
111	−	1 0 0 0 0 1 1 1		
	1	1 1 0 1 1 1 1 0	0 0 1 0 1 0 1 0	
	+	1 0 0 0 0 1 1 1		
	0	0 1 1 0 0 1 0 1		

$(42)_{10}$

Remainder = $(101)_{10}$ ⇒ 13 op. aritmetice

Ex:

```
1011 0110 | 1000
-1000     | 10110 ← Quotient = (22)10
 0110     |
-1000     |
 1110     |
+1000     |
 0110     |
-1000     |
 1110     |
-1000     |
 0110     |
-1000     |
 1110     |
+1000     |
 0110     |
```

↳ Remainder = 6_{10}

Divizor → n-bit

Quotient → n-bit

Remainder → n-bit

Dividend → (2n-1) bit

1.4.2. Non-Restoring Division

$r'_{i-1} = r_{i-1} - M$

↳ restul parțial ↳ cel care trebuia să dea

$r'_i = 2 r'_{i-1} = 2(r_{i-1} - M)$

↳ shiftare

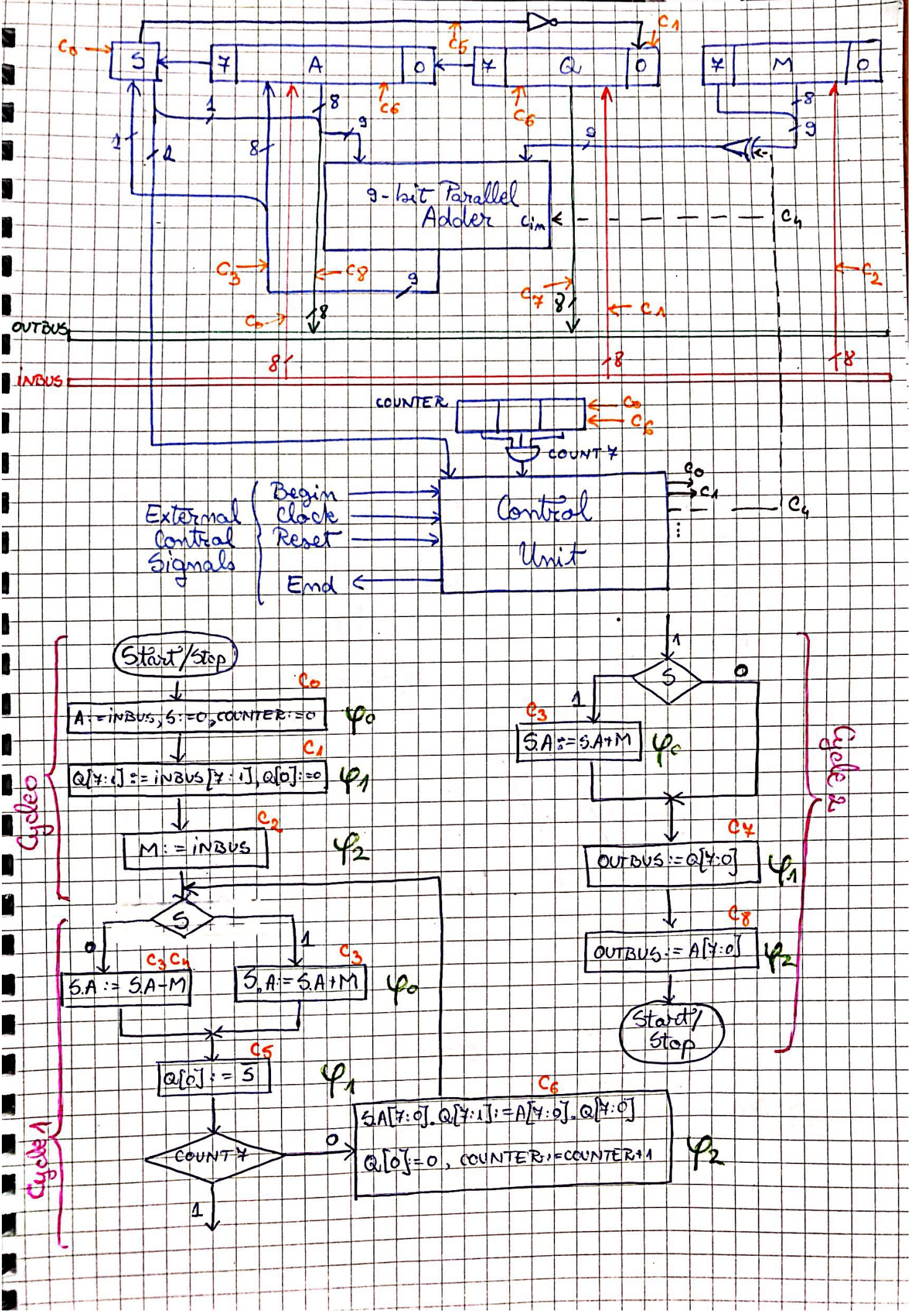
Dacă $q_i = 0 \Rightarrow M$ nu intră în $A \Rightarrow r'_i = 2 r_{i-1} - M = r'_i + M$

$q_i = 1 \Rightarrow M$ intră în $A \Rightarrow r'_i = 2(r_{i-1} - M) - M = r'_i - M$

COUNT	S	A	Q	M
000	0	0 0 1 0 1 1 0 1	0 0 0 1 0 1 1 0	1 0 0 0 0 1 1 1
	−	1 0 0 0 0 1 1 1		
	(1)	1 0 1 0 0 1 1 0	0 0 0 1 0 1 1 (0)	
	1	0 1 0 0 1 1 0 0	0 0 1 0 1 1 0 0	
001	+	1 0 0 0 0 1 1 1		
	(1)	1 1 0 1 0 0 1 1	0 0 1 0 1 1 0 (0)	
	1	1 0 1 0 0 1 1 0	0 1 0 1 1 0 0 0	
010	+	1 0 0 0 0 1 1 1		
	(0)	0 0 1 0 1 1 0 1	0 1 0 1 1 0 0 (1)	
	0	0 1 0 1 1 0 1 0	1 0 1 1 0 0 1 0	
011	−	1 0 0 0 0 1 1 1		
	(1)	1 1 0 1 0 0 1 1	1 0 1 1 0 0 1 (0)	
	1	1 0 1 0 0 1 1 1	0 1 1 0 0 1 0 0	
100	+	1 0 0 0 0 1 1 1		
	(0)	0 0 1 0 1 1 1 0	0 1 1 0 0 1 0 (1)	
	0	0 1 0 1 1 1 0 0	1 1 0 0 1 0 1 0	
101	−	1 0 0 0 0 1 1 1		
	1	1 1 0 1		
	1			
110	+	1 0 0 0 0 1 1 1		
	(0)	0 0 1 1 0 0 1 0		
	0	0 1 1 0 0 1 0 1	0 0 1 0 1 0 1 (0)	
111	−	1 0 0 0 0 1 1 1		
	(1)	1 1 0 1 1 1 1 0	0 0 1 0 1 0 1 (0)	
			Quotient = $(42)_{10}$	
PC	+	1 0 0 0 0 1 1 1		
	0	0 1 1 0 0 1 0 1	Remainder = $(101)_{10}$	

↳ pas de corecție pt. că ultimul S = 1.

⇒ 9 op. aritmetice



1.4.3. Algoritmul S.R.T.

Reprezentare pentru:

- radix 4: $\{\bar{2}, \bar{1}, 0, 1, 2\}$
- radix 8: $\{\bar{4}, \bar{3}, \bar{2}, \bar{1}, 0, 1, 2, 3, 4\}$
- radix 16: $\{\bar{8}, \bar{7}, \bar{6}, \ldots 6, 7, 8\}$
- radix 2: $\{\bar{1}, 0, 1\}$

$10\bar{1}0\bar{1} = -1 \cdot 2^0 + 0 \cdot 2^1 - 1 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 = 11$

Codificare: $1 \to \begin{smallmatrix}1\\0\end{smallmatrix}$; $0 \to \begin{smallmatrix}0\\0\end{smallmatrix}$; $\bar{1} \to \begin{smallmatrix}0\\1\end{smallmatrix}$ $\Rightarrow$

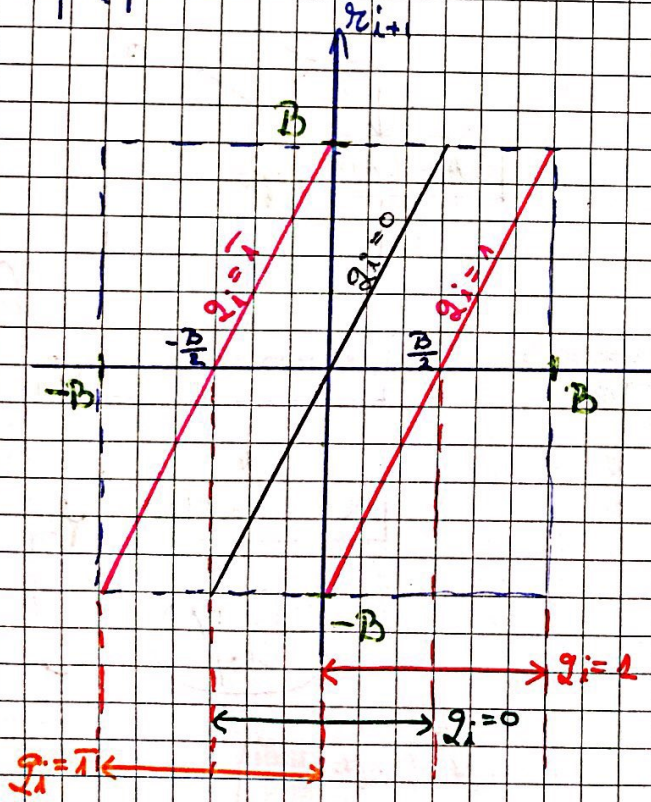
$\Rightarrow$
$$\begin{array}{r} 10000- \\ 00101 \\ \hline 01011 = (11)_{10} \end{array}$$

$r_{i+1} \leftarrow 2r_i - q_i \cdot B$

$q_i \in \{\bar{1}, 0, 1\}$

$|r_i| < B$

Aici: $\begin{cases} A \to P \\ Q \to A \\ M \to B \end{cases}$



$r_i = 0 \Rightarrow r_{i+1} = 2B - B = B \quad q_i = 1$

$r_{i+1} = 2r_i - B = 0 \Rightarrow$

$\Rightarrow r_i = \frac{B}{2}$

$r_{i+1} = -B = 2r_i - B \Rightarrow$

$\Rightarrow r_i = 0$

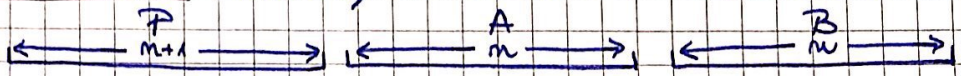
$q_i = -1$:

$M_{i+1} = B \Rightarrow M_i = 0$

$M_{i+1} = 0 \Rightarrow M_i = -B/2$

$M_{i+1} = -B \Rightarrow M_i = -B$

Algoritmul lui SRT: (radix 2) 2^k Divident 2^k Divisor

(1.) 

If B has k-leading 0's (zeros), shift P.A.B k positions left

(2.) For $i=0$ to $n-1$

If top 3 bits of register P are equal then $q_i=0$; Shift P.A left

If top 3 bits of P are not equal and $P>0$ then:

$q_i = 1$; Substract B from P; Shift P.A left

If top 3 bits of P are not equal and $P<0$ then:

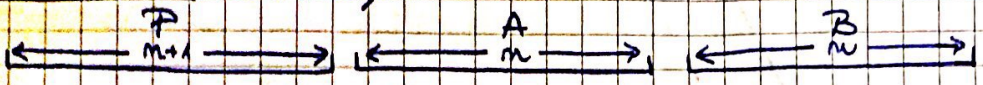
$q_i = \bar{1}$; Add P to B; Shift P.A left

(3.) If $P<0$ Add B to P; Substract 1 from A

(4.) Shift P right k positions

Algoritmul lui SRT: (radix 2)

2^k Divident 2^k Divisor



1. If B has k leading 0's (zeros), shift P.A.B k positions left
2. For $i=0$ to $n-1$
   If top 3 bits of register P are equal then $q_i=0$; Shift P.A left
   If top 3 bits of P are not equal and $P>0$ then:
   $q_i=1$; Substract B from P; Shift P.A left
   If top 3 bits of P are not equal and $P<0$ then:
   $q_i=\bar{1}$; Add P to B; Shift P.A left
3. If $P<0$ Add B to P; Substract 1 from A
4. Shift P right k positions

CURS 5 — 26.03.2019

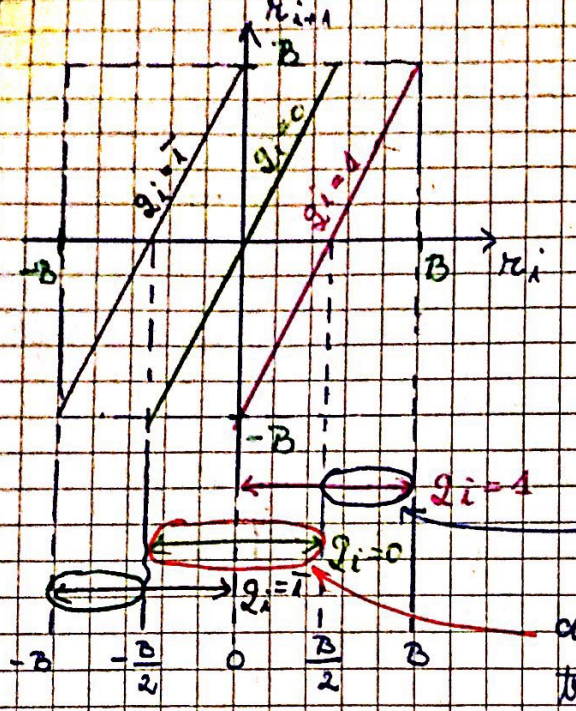
Exemplu S.R.T

$229 : 12 = 19$ $r=1$

COUNT	P	A	B
000	00000000000	11100101	00001100
	00000011100	10101000	11000000 — shift
	$q_0=0$		
	00001110010	10100000	
001	$q_1=0$		⊛ → 1 nu poate fi reținut în calc, deci folosim codificarea:
	00011100101	00000000	
010	$q_2=0$		00010100 −
	00111001010	00000000	00000001
011	$q_3=1$		00010011
	01110010101	00000001	Quotient = 19 (10)
	− 11000000		
	00010010101		
100	$q_4=0$		
	00100101010	00000010	
101	$q_5=1$		
	01001010100	00000101	
	− 11000000		
	11101010000		
110	$q_6=0$		
	11010100000	00001010	
111	$q_7=\bar{1}$		
	10101000000	00010101	
	+ 11000000		
	! 00010000		⊛

Shiftăm la dreapta cu 4 poziții

Remainder: 0000000001 = 1 (10)



r_i = rest partial

$r_{i+1} = 2r_i - q_i \cdot B$; $q_i \in \{\bar{1}, 0, 1\}$

pt. $q_i = 1$ luăm regm. asta pt $r_i \in [\frac{B}{2}, B]$
dar pt. $r_i \in [0, \frac{B}{2}]$ e mai ok $q_i = 0$
cea mai favorabilă situație pt. că nu trie să faci nimic

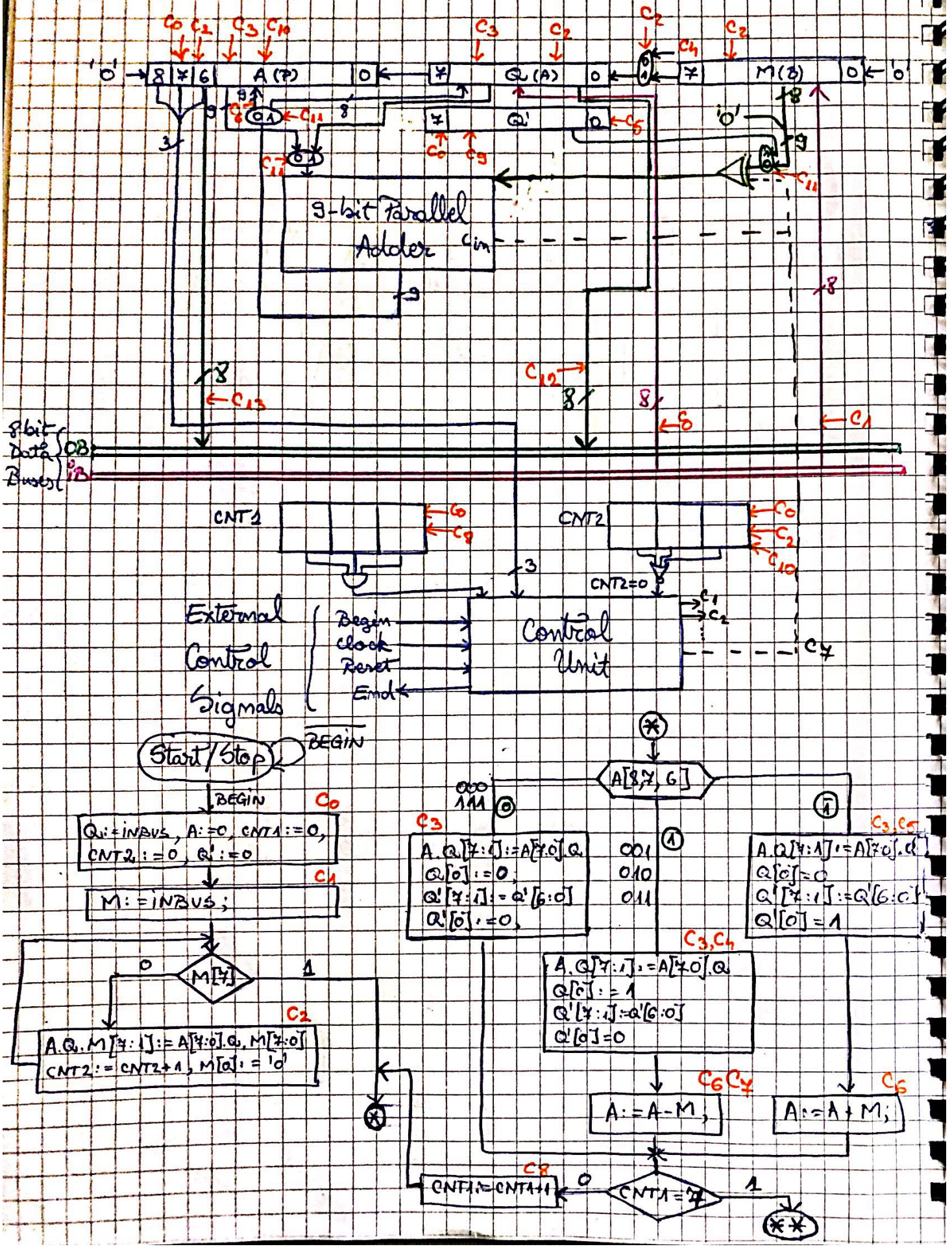
⊙ Exemplu S.R.I.T:

COUNT	P	A	B
000	0 0 0 0 0 0 0 0 0 0	1 1 1 0 1 0 0 1	0 0 0 0 0 1 0 0 1
	0 0 0 0 0 0 1 1 1 0	1 0 0 1 0 0 0 0	1 0 0 1 0 0 0 0
	$q_0=0$		
	0 0 0 0 0 1 1 1 0 1	0 0 1 0 0 0 0 0	
001	$q_1=0$		
	0 0 0 0 1 1 1 0 1 0	0 1 0 0 0 0 0 0	
010	$q_2=0$		
	0 0 0 1 1 1 0 1 0 0	1 0 0 0 0 0 0 0	
011	$q_3=1$		
	0 1 1 1 0 1 0 0 1	0 0 0 0 0 0 0 1	
	- 1 0 0 1 0 0 0 0		
	0 0 1 0 1 1 0 0 1		
100	$q_4=1$		
	0 1 0 1 1 0 0 1 0	0 0 0 0 0 0 0 1 1	
	- 1 0 0 1 0 0 0 0		
	0 0 0 1 0 0 0 1 0		
101	$q_5=0$		
	0 0 1 0 0 0 1 0 0	0 0 0 0 0 1 1 0	
110	$q_6=1$		
	0 1 0 0 0 1 0 0 0	0 0 0 0 1 1 0 1	
	- 1 0 0 1 0 0 0 0		
	1 1 1 1 1 1 0 0 0		
111	$q_7=0$		
	1 1 1 1 1 1 0 0 0 0	0 0 0 1 1 0 1 0 - 1	
Correction (pt. că 1)	+ 1 0 0 1 0 0 0 0	0 0 0 1 1 0 0 1	
	0 1 0 0 0 0 0 0 0	Quotient = $25_{(10)}$	
	(4 pași)		
Shift	0 0 0 0 0 1 0 0 0		

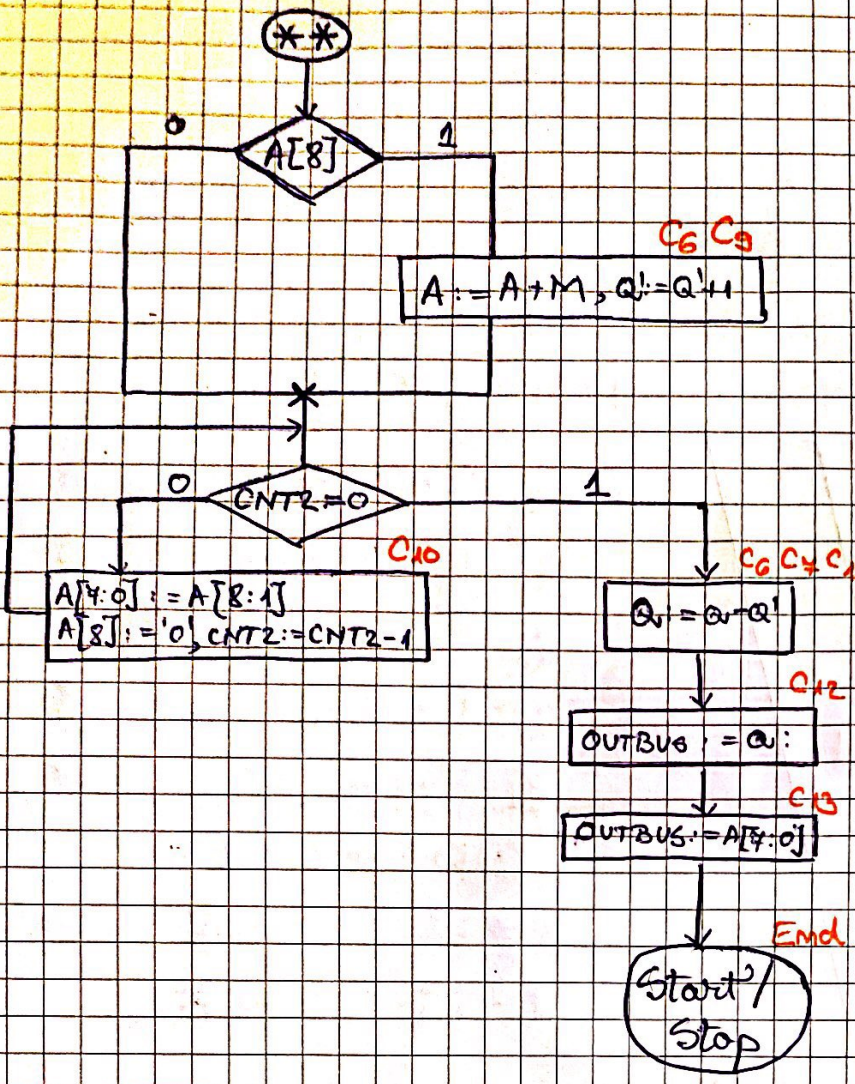
Remainder = $8_{(10)}$

CURS 5 (continuare)

1.4.4. Hardware implementation for radix-2 SRT



// CNT 2 trebuie să fie un registru capabil și de incrementare și de decrementare



1.5 Radix-4 SRT

$q_i \in \{\bar{2}, \bar{1}, 0, 1, 2\}$

$$r_{i+1} = 4r_i - q_i \cdot B$$

- $|r_i| < B \Rightarrow |r_{i+1}| < B$
  - insuficient

Fp. $r_i = B \Rightarrow r_{i+1} = 4B - q_i B = 4B - 2B = 2B$ (X)

- $|r_i| < \frac{2}{3} B \Rightarrow |r_{i+1}| < \frac{2}{3} B$

Fp. că $r_i = \frac{2}{3} B \Rightarrow r_{i+1} = \frac{8}{3} B - \frac{6}{3} B = \frac{2}{3} B$ (✓)

CURS 6 — 2.04.2019

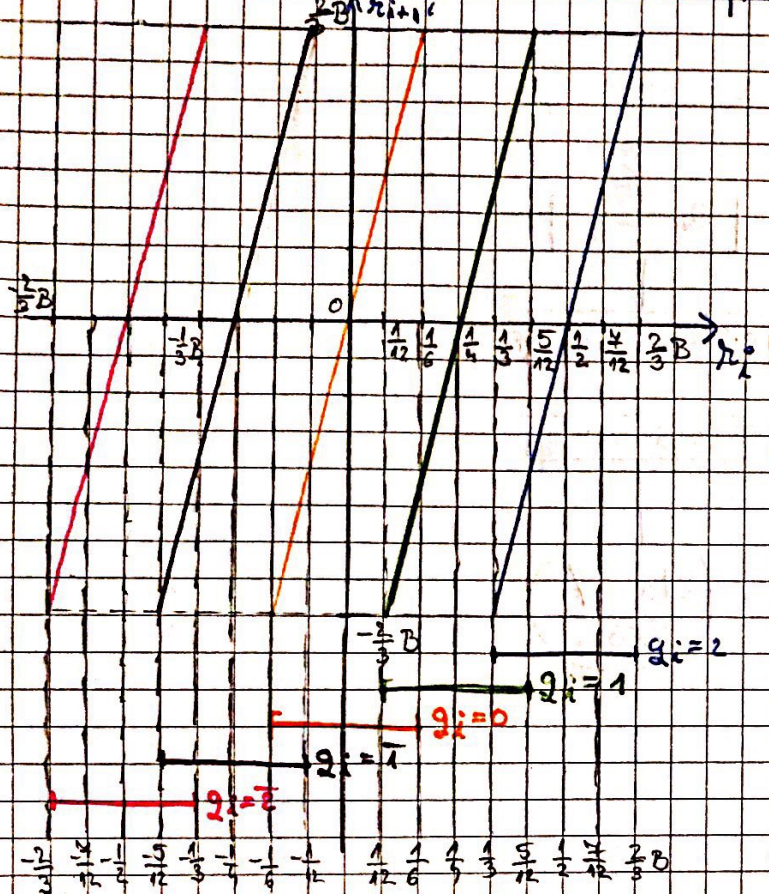
1.6. Radix-4 SRT

$q_i \in [\bar{2}, \bar{1}, 0, 1, 2]$

$r_{i+1} = 4 \cdot r_i - q_i \cdot B$

$|r_i| < B \Rightarrow$ Nu rezultă că $|r_{i+1}| < B \Rightarrow$

$\Rightarrow$ Trb. să ținem mereu $|r_i| < \frac{2}{3} B$



$q_i = 2 \Rightarrow r_{i+1} = 4r_i - 2B$

$q_i = 1 \Rightarrow r_{i+1} = 4r_i - B$

$q_i = 0 \Rightarrow r_{i+1} = 4r_i$

$q_i = \bar{1} \Rightarrow r_{i+1} = 4r_i + B$

$q_i = \bar{2} \Rightarrow r_{i+1} = 4r_i + 2B$

⊙ Codare radix 4:

$0 \to \begin{smallmatrix}0&0\\0&0\end{smallmatrix}$ $1 \to \begin{smallmatrix}0&1\\0&0\end{smallmatrix}$ $2 \to \begin{smallmatrix}1&0\\0&0\end{smallmatrix}$

$\bar{1} \to \begin{smallmatrix}0&0\\0&1\end{smallmatrix}$ $\bar{2} \to \begin{smallmatrix}0&0\\1&0\end{smallmatrix}$

⊙ Exemplu în radix 4:

229 : 12 = 19 rest 1

→ ne uităm la primii 6 biți ai lui P

COUNT	P	A	B
000	0 0 0 0 0 0 0 0 0 0	1 1 1 0 0 1 0 1	0 0 0 0 (1) 1 0 0
+1=	0 0 0 0 0 1 1 1 0	0 1 0 1 0 0 0 0	(1 1 0 0) 0 0 0 0
	↳ ne aflăm în [-4, 3] → $q_i = 0$		↳ după asta ne ghidăm în tabel (6) - dublu shift
+7=	0 0 0 1 1 1 0 0 1	0 1 0 0 0 0 ⓪	
01	↳ [3, 9] → $q_i = 1$		
	0 1 1 1 0 0 1 0 1	0 0 0 0 ⓪ ①	
	− 1 1 0 0 0 0 0 0		
+4=	0 0 0 1 0 0 1 0 1		
10	↳ [3, 9] → $q_i = 1$		
	0 1 0 0 1 0 1 0 0	0 0 ⓪ ① ①	
	− 1 1 0 0 0 0 0 0		
−6=	1 1 1 0 1 0 1 0 0		
11	↳ [−10, −4] → $q_i = -1$		
	1 0 1 0 1 0 0 0 0	⓪ ① ① $\bar{1}$	
	+ 1 1 0 0 0 0 0 0		
	0 0 0 0 1 0 0 0 0	Quotient:	
Shift	0 0 0 0 0 0 0 0 1	$0 \cdot 4^3 + 1 \cdot 4^2 + 1 \cdot 4^1 - 1 \cdot 4^0 = 16 + 4 - 1 = (19)_{10}$	
	Remainder = $(1)_{10}$	sau	
		0 0 0 1 0 1 0 0 −	
		0 0 0 0 0 0 0 1	
		1 0 0 1 1 = $(19)_{10}$	

cifre care trb. podiliate după coloanele de mai sus

wr. în C_2 →

⊙ Exemplu SRT radix 4:

238 : 10 = 23 r = 8

B = 0 1 0 1 0 0 0 0 0

2B = 1 0 1 0 0 0 0 0 0

COUNT	P	A	B
00	0 0 0 0 0 0 0 0 0 0	1 1 1 0 1 1 1 0	0 0 0 0 1 0 1 0
11 =	0 0 0 0 0 0 1 1 1 0	1 1 1 0 0 0 0 0	1 0 1 0 0 0 0 0
	↳ [-4, 3] => q_i = 0		b = 10
+7 =	0 0 0 0 1 1 1 0 1 1	1 0 0 0 0 0 ⓪	
01	↳ [3, 9] => q_i = 1		
	0 1 1 1 0 1 1 1 0	0 0 0 0 ⓪ ①	
	- 1 0 1 0 0 0 0 0		
+9 =	0 0 1 0 0 1 1 1 0		
10	↳ [8, 14] => q_i = 2		
	1 0 0 1 1 1 0 0 0	0 0 ⓪ ① ②	
	- 1 0 1 0 0 0 0 0		
=	1 1 1 1 1 1 0 0 0		
	↳ q_i = 0		
	1 1 1 1 0 0 0 0	⓪ ① ② ⓪	
CORR	+ 1 0 1 0 0 0 0 0	-1	
	0 1 0 0 0 0 0 0 0		
Shift	0 0 0 0 0 1 0 0 0		

R. ca restul a dat negativ

Remainder = $(8)_{10}$

Quotient: $2 \cdot 4^1 + 1 \cdot 4^2 - 1 = 8 + 16 - 1 = (23)_{10}$

	0	1	2	0	
sau:	0 0	0 1	1 0	0 0	-
	0 0	0 0	0 0	0 1	← de la corecție
	0 0	0 1	0 1	1 1	

$0\,0\,0\,1\,0\,1\,1\,1 = (23)_{10}$

← de la corecție (la ultima cifră se adaugă $\begin{smallmatrix}0\\0\end{smallmatrix}$ i = -1)

Cap II. Performance in computer systems

1. Performance is hard to assess in computer systems
2. Producers tend to present their product in the best possible light
3. Design decisions are based on performance assessments.

A computer system with 1 processor:

A. we replace the processor with a faster one

B. we add more processors to our computer system

User Response Time → System Response Time

Processor OS

CPU_{time} = No of clock cycles per program × clock cycle time =

= Instructions count × Clock cycles per instruction × clock cycle time

- $CPU_{time} = IC \times CPI \times$ Clock cycle time (CCT)
- $CPI = \sum_{i=1}^{n} \frac{IC_i}{IC} \times CPI_i$ unde IC_i - no. of instructions in class i
  CPI_i - CPI of class i.
- Performance of $X = \frac{1}{\text{Execution time } X}$
- Computer A is n times faster than computer B $\Rightarrow$

$\Rightarrow n = \frac{\text{Performance A}}{\text{Performance B}} = \frac{\text{Execution time B}}{\text{Execution time A}}$

Example: Pp. că avem 2 implementări ale aceleași arhitecturi. Computer A: CCT = 250 ps și CPI = 2,0 } același program (IC)
Computer B: CCT = 500 ps și CPI = 1,2 }

Care comp. e mai rapid și cu cât

$CPU_{time\,A} = IC \times 2,0 \times 250 = IC \times 500 ps$
$CPU_{time\,B} = IC \times 1,2 \times 500 = IC \times 600 ps$ } $\Rightarrow$

$\Rightarrow$ Comp. A is $\frac{IC \times 600}{IC \times 500} = 1,2$ times faster than comp. B.

CURS 7 9.04.2019

2.1 Performance Equation

CPU_{time} = Instruction Count x Clock cycles per instruction * Clock cycle time
= IC x CPI x Clock cycle time

Example: Fie 2 segmente de cod pt. care:

	CPI for each instruction class		
	A	B	C
CPI	1	2	3

- $CPU_{time} = IC \times CPI \times$ Clock cycle time
  ↳ CCT
- $CPI = \sum_{i=1}^{n} \frac{IC_i}{IC} \times CPI_i$ unde IC_i - no. of instructions in class i
  CPI_i - CPI of class i.
- Performance of $X = \frac{1}{\text{Execution time } X}$
- Computer A is n times faster than computer B =>

=> => $n = \frac{\text{Performance A}}{\text{Performance B}} = \frac{\text{Execution time B}}{\text{Execution time A}}$

Example: Pp. că avem 2 implementări ale aceleași arhitecturi: Computer A: CCT = 250 ps și CPI = 2,0 } aceleași program (UC)
Computer B: CCT = 500 ps și CPI = 1,2 }

Care comp. e mai rapid și cu cât

$CPU_{time\,A} = IC \times 2,0 \times 250 = IC \times 500$ ps
$CPU_{time\,B} = IC \times 1,2 \times 500 = IC \times 600$ ps } =>

=> Comp. A is $\frac{IC \times 600}{IC \times 500} = 1,2$ times faster than comp. B.

CURS 7 9.04.2019

2.1 Performance Equation

CPU_{time} = Instruction Count x Clock cycles per instruction * Clock cycle time
= $IC \times CPI \times$ Clock cycle time

Example: 2 segmente de cod pt. care:

	CPI for each instruction class		
	A	B	C
CPI	1	2	3

Code Sequence	Instruction Counts for each instruction class		
	A	B	C
1	2	1	2
2	4	1	1

$$CPU_{time_1} = IC_1 \times CPI_1 \times \text{Clock cycle time} = \underbrace{5}_{IC_1} \times \underbrace{\frac{2\times1+1\times2+2\times3}{5}}_{CPI_1} \times CCT$$

$$= 10 \text{ clock cycles} \times CCT$$

$$CPU_{time_2} = IC_2 \times CPI_2 \times CCT = \underbrace{6}_{IC_2} \times \frac{4\times1+1\times2+1\times3}{6} \times CCT =$$

$$= 9 \text{ clock cycles} \times CCT$$

← compilatorul mai bun

$$\frac{\text{Performance}_2}{\text{Performance}_1} = \frac{CPU_{time_1}}{CPU_{time_2}} = \frac{10}{9} \cong 1,1$$

Example: Fie un compilator vechi (CV) și unul nou (CN)

CV: $CPU_{time_{CV}} = IC_{old} \times CPI_{old} \times CCT$

CN: $CPU_{time_{CN}} = IC_{new} \times CPI_{new} \times CCT$

! CCT nu se schimbă pt. că se rulează pe același PC

$IC_{new} = 0,6 \times IC_{old}$

$CPI_{new} = 1,1 \times CPI_{old}$

$CPU_{time_{CV}} = 15$ sec

Deci $CPU_{time_{CN}} = 0,6 \times IC_{old} \times 1,1 \times CPI_{old} \times CCT = CPU_{time_{CV}} \times 0,6 \times 1,1$

$= 15 \times 0,6 \times 1,1 = 9,9$ sec

Example:
- Frequency of FP (floating point operations) op = 25%
- Average CPI of FP op = 4,0
- Avg. CPI of (#) other op = 1,33
- Frequency of FPSQR = 2%
- CPI of FPSQR = 20.

Two options: – to decrease CPI of FPSQR to 2

– to decrease the avg. CPI of all FP op. to 2.5

Which option is the best? (CPU_{time_1} vs. CPU_{time_2} (→) (→) CPI°_1 vs. CPI_2)

75%

$CPI_{original} = 0,25 \times 4,0 + 0,75 \times 1,33 \simeq 2$ clock cycles (c.c)

$CPI_1 = CPI_{original} - 2\% \times (20-2) = 2 - 0,02 \times 18 \simeq 1,64$ c.c.

$CPI_2 = 0,25 \times 2,5 + 0,75 \times 1,33 \simeq 1,625$ c.c.

=> Second option is better.

⊙ Example:

CPU_A

```
CMP  r1, r2
BEQ  F00
 .
 .
 .
F00: ADD  r6, r5, r6
 .
 .
 .
```

CPU_B

```
BEQ  r1, r2, F00
 .
 .
 .
F00 .
```

$CPU_{time\,A} = CCT_A \times IC_A \times CPI^{\circ}_A$

$CPI_A = 20\% \times 2 + 80\% \times 1 \simeq 1,2$ c.c } =>

2 c.c. – pt. branchuri condiționate

$=> CPU_{time\,A} = \underbrace{1,2}_{\simeq 1} \times IC_A \times CCT_A$

$CPU_{time\,B} = \underbrace{0,8 \times IC_A}_{IC_B} \times CPI_B \times \underbrace{1,25}_{CCT_B} \times CCT_A$

↳ 80% pt. că au dispărut cele 20% de la instr. de comparare

$CPI^{\circ}_B = 0,25 \times 2 + 0,75 \times 1 = 1,25$ c.c.

$CPU_{time\,B} = IC_A \times 1,25 \times CCT_A$

=> Nu se merită optimizarea B.

⊙ Example:

CPI = 2 — ADD r_1, r_2, # mem. loc (memory location) → (B)

LDR r_1, # mem. loc

CPI = 1 — ADD r_2, r_2, r_1

STR r_2, # mem. addr_2

$$\text{CPU time}_A = IC_A \times CPI_A \times CCT_A = IC_A \times \underline{1,57} \times CCT_A$$

↳ neoptimizat

$CPI_A = 0,43 \times 1 + 0,57 \times 2 = 1,57$ c.c.

↳ 43% durează un clock; ↳ 57% durează 2 clocki

instrucțiunile care nu se mai execută

$$\text{CPU time}_B = IC_A \underbrace{(1 - 0,43 \times 0,25)}_{IC_B} \times CPI_B \times CCT_A$$

↳ care nu se schimbă

IC_B = instrucțiuni ce rămân să se execute pt. 1 c.c

loadurile ↙

$$CPI_B = \frac{(0,75 \times 0,43) \times 1 + (0,25 \times 0,43) \times 2 + (0,21 - 0,25 \times 0,43) \times 2}{1 - 0,43 \times 0,5} +$$

store-urile; branchuri → nr. nou de instrucțiuni

$$+ \frac{(0,12 \times 2) + (0,24 \times 3)}{1 - 0,43 \times 0,5} \simeq 1,7$$

Deci $\text{CPU time}_B = IC_A \times \underline{1,7} \times CCT_A$

2.2 Real-world benchmarks

MOR DE PLICTISEALĂ KK

KK, nu bum facultă!!!

bucKK

KK

KK

KK

PAPA plic

A. Synthetic benchmarks
B. Toy programs
C. Kernels
D. Real programs mix

SPEC (Standard Performance Evaluation Corporation)
— Benchmarks suites

$$SPEC_A = \sqrt[n]{\prod_{i=1}^{n} SPEC_{A_i}}$$ — SPEC native

unde $SPEC_{A_i}$ = execution time on machine A of program i from the SPEC Benchmark Suite

$$\frac{SPEC_{reference}}{SPEC_A} = \frac{\sqrt[n]{\prod_{i=1}^{n} SPEC_{reference_i}}}{\sqrt[n]{\prod_{i=1}^{n} SPEC_{A_i}}} = \sqrt[n]{\prod_{i=1}^{n} \frac{SPEC_{reference_i}}{SPEC_{A_i}}}$$

$$\text{SPEC relativ} = \frac{\text{Performance A}}{\text{Performance}_{reference}} = \text{SPEC Ratio}_A$$

$$\frac{\text{Performance A}}{\text{Performance B}} = \frac{\text{SPEC Ratio A}}{\text{SPEC Ratio B}} = \frac{\sqrt[n]{\prod_{i=1}^{n} SPEC_{B_i}}}{\sqrt[n]{\prod_{i=1}^{n} SPEC_{A_i}}}$$

CNT	OVR	A	Q[8]	Q	F	M
000		00000000	1	10111011	0	11011011
	+	00100101				00100101 = M
	0	00100101				
		00010010	1	11011101	1	
001		00001001	0	11101101	1	
010	+	00100101				
	0	00101110				
		00010111	0	01110111	1	
011		00001011	1	00111011	1	
100		00000101	1	10011101	1	
101		00000010	1	11001110	1	
110	+	00100101				
		00101111				
		00010011	1	11100111	1	
111		00001001	1	11110011		
				P		

CURS 8 16.04.2019

2.3. Other metrics

MIPS - Million Instructions Per Second

$$MIPS = \frac{\text{Instruction Count}}{\text{Execution Time} \times 10^6} = \frac{\text{IC}}{\text{CPU}_{\text{time}}} = \frac{\text{IC}}{\text{IC} \times \text{CPI} \times \text{CCT} \times 10^6} = \frac{\text{Clock frequency}}{\text{CPI} \times 10^6}$$

10^0 10^3 10^6 10^9 10^{12} 10^{15} 10^{18} 10^{21} 10^{24}

kilo mega giga tera peta exa zetta yotta

10^{-24} 10^{-21} 10^{-18} 10^{-15} 10^{-12} 10^{-9} 10^{-6} 10^{-3} 10^0

yocto zepto atto femto pico nano micro mili

◎ Example:

$$\text{Clock frequency} = \text{Clock rate} = \frac{1}{20\,ns} = \frac{1}{20 \cdot 10^{-9}} = 50\ MHz$$

$$MIPS_{original} = \frac{50 \cdot 10^6}{CPI_{original} \cdot 10^6} = \frac{50}{CPI_{original}} = \frac{50}{1,57} = \boxed{31,85}$$

$$CPI_{original} = 0,43 \cdot 1 + 0,57 \cdot 2 = \boxed{1,57}$$

$$CPU_{time\ original} = IC_{original} \cdot 1,57 \times 20\,ns = \boxed{IC_{original} \times 31,4\,ns}$$

$$MIPS_{optimizat} = \frac{50}{CPI_{optimizat}} = \frac{50}{1,73} = \boxed{28,9}$$

50% = 0,5 (scad 50% din ALU)

$$CPI_{optimizat} = \frac{\left(0,43 \cdot \frac{50}{100}\right) \cdot 1 + 0,57 \times 2}{1 - 0,43 \cdot 0,5} = \boxed{1,73}$$

$$CPU_{time\ optimizat} = IC_{original} \times (1 - 0,43 \cdot 0,5) \times CPI_{optimizat} \times 20\,ns$$

$$= IC_{original} \times \underbrace{0,785 \times 1,73}_{1,355} \times 20\,ns =$$

$$= \boxed{IC_{original} \times 27,1\,ns}$$

Concluzie: Uneori MIPS variază invers proportional cu CPU_{time}

- MFLOPS – Million Floating Point Operations Per Second
- GFLOPS – Giga Floating Point Operations Per Second

$$GFLOPS = \frac{Instruction\ Count_{FP}}{CPU_{time} \times 10^9}$$

$$Relative\ MIPS_x = \frac{Execution\ time_{Reference}}{Execution\ time_x} \times MIPS_{Reference}$$

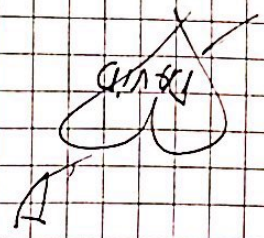
VAX 11/780 → 1 MIPS Machine

$$Normalised\ GFLOPS = \frac{Normalised\ IC_{FP}}{CPU_{time} \times 10^9}$$

Mc Mahon – Lawrence Livermore National Laboratories

Livermore Loops:

Native FP Operation	Normalised FP Operation
+, −, *, COMP	1
DIVIDE, SQRT	4
EXP, SIN, ...	8



2.4. Quantitative Principles of Computer Design

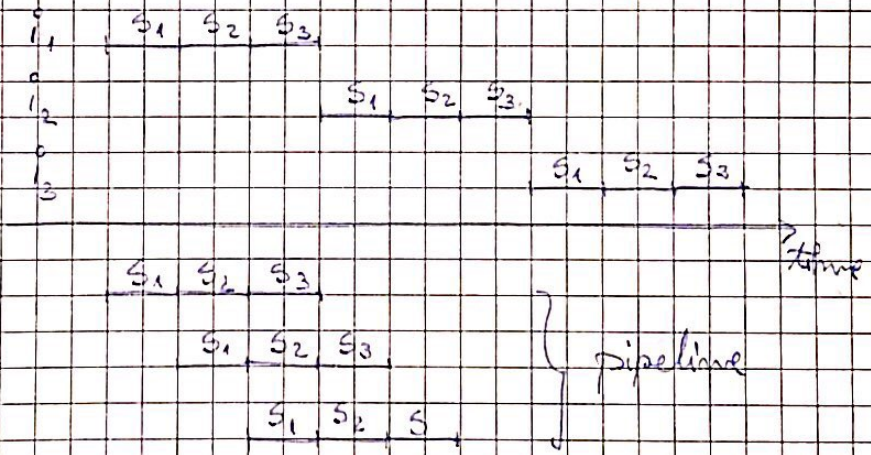
(1.) Use parallelism when possible:

↳ task-level parallelism

↳ data-level parallelism

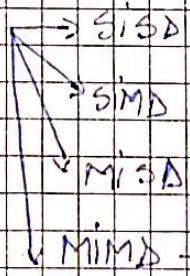
↳ logical-level parallelism (CLA)

↳ instruction-level pipeline



↳ system-level parallelism

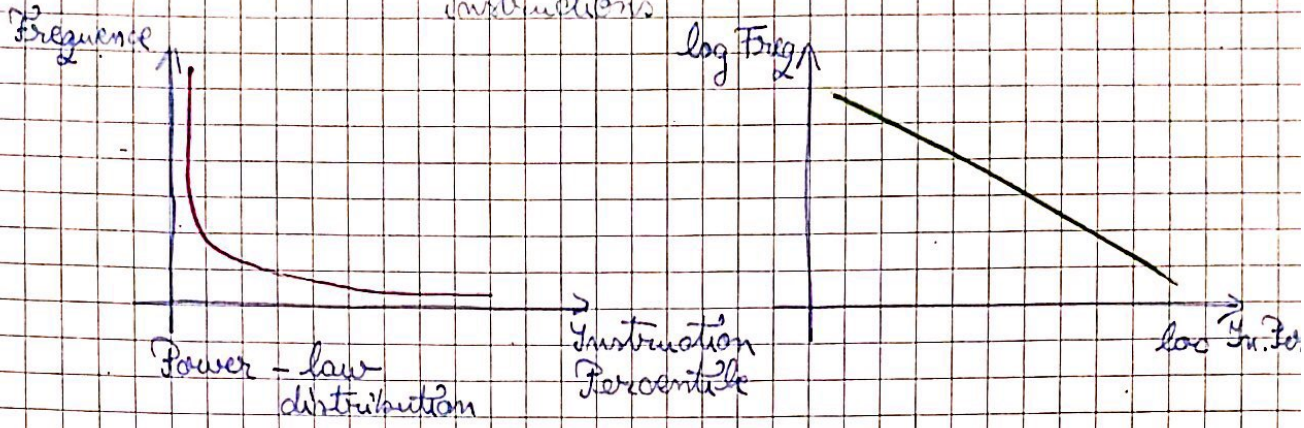
Flynn's Taxonomy



MIMD → NoC (Network On Chip)

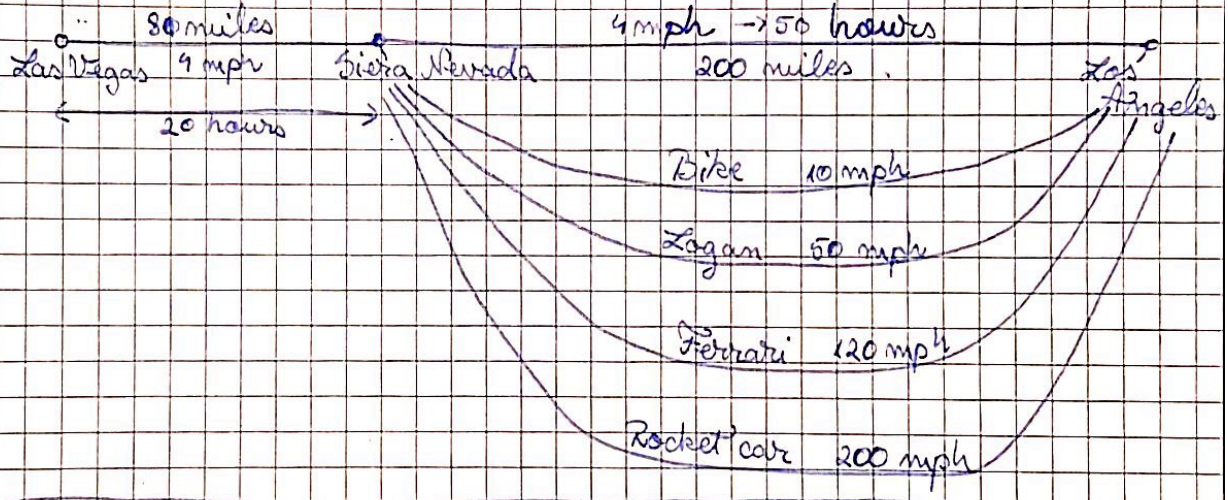
2. Localité principale :

Rule of thumb : 90% of the time the computer uses 10% of the instructions



3. Always optimise the common case :

Amdahl's Law :



Vehicle for 2nd part of the trip	Time for 2nd part	Speedup 2nd part	Total time	Overall Speedup
Feet	50	1	70	1
Bike	20	2,5	40	1,75
Logan	4	12,5	24	2,9
Ferrari	1,6	30	21,6	3,23
Rocket car	1	50	21	$\frac{70}{21} = 3,33$

$$\text{Speedup Overall} = \frac{\text{Time unoptimised}}{\text{Time optimised}} = \frac{\text{Time unop.}}{\text{Time unop.} \left(1 - \text{Fracture optimisation}\right) + \text{Fracture optimisation}}$$

$$\Rightarrow \text{Speedup}_{\text{overall}} = \frac{1}{\left(1 - \frac{\text{Fraction}}{\text{Optimised}}\right) + \frac{\text{Fraction optimised}}{\text{Speedup}}}$$

In the example:

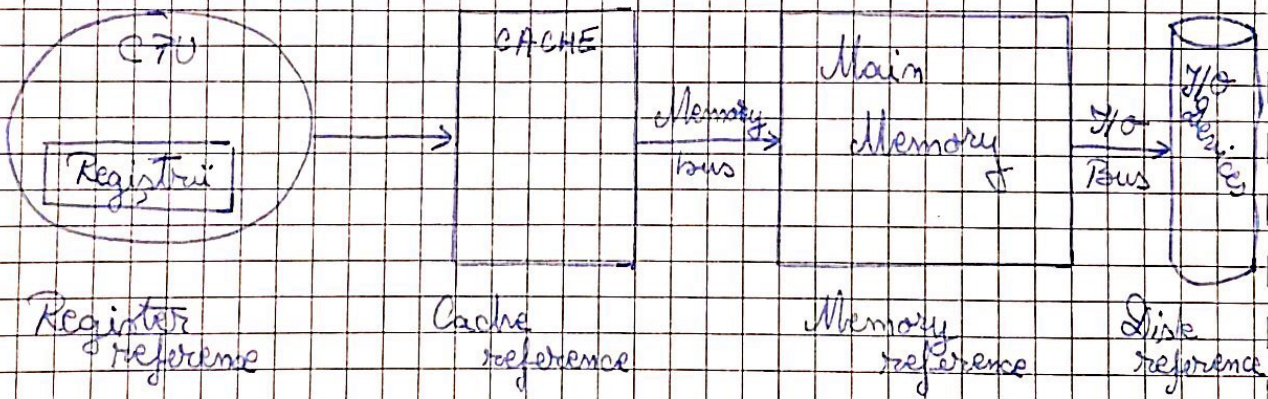
$$\text{Fraction}_{\text{optimised}} = \frac{200}{280} = \frac{20}{28} = \frac{5}{7}$$

$$\text{Speedup}_f = 60$$

$$\text{Speedup overall} = \frac{1}{\frac{2}{7} + \frac{5}{7 \cdot 60}} = \frac{70}{21} = 3,3$$

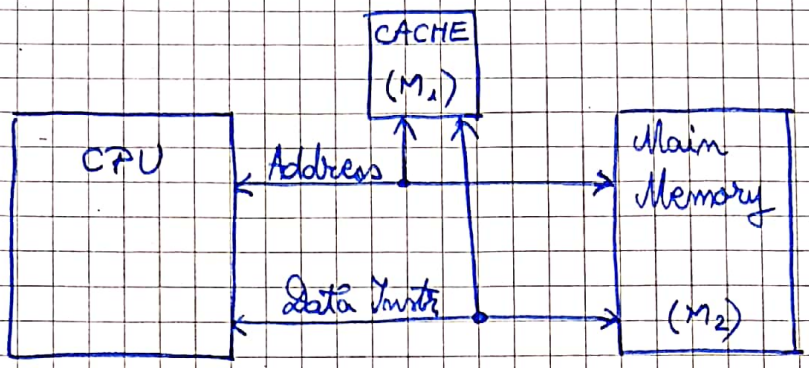
Când $\text{Speedup}_f \to \infty \Rightarrow \text{Speedup}_{\text{overall}} = \frac{1}{1 - \text{Fraction Optimised}}$

Cap III Memory hierarchy



CURS 9

3.1 Introduction



- Cache e aprox. 10% din Main Memory
- M_1, M_2 - notații conform Patterson și Hennessy
- Cea mai mică unitate de memorie: Byte;
- Word = 2^p bytes, $p \in \mathbb{N}$

ex: $p = 2$

Byte

Word	0	1	2	3

$p = 2 \rightarrow$ Word offset: $\begin{cases} 00 \\ 01 \\ 10 \\ 11 \end{cases}$

- Block = 2^r words, pe r - bit offset

3.2 Direct mapping

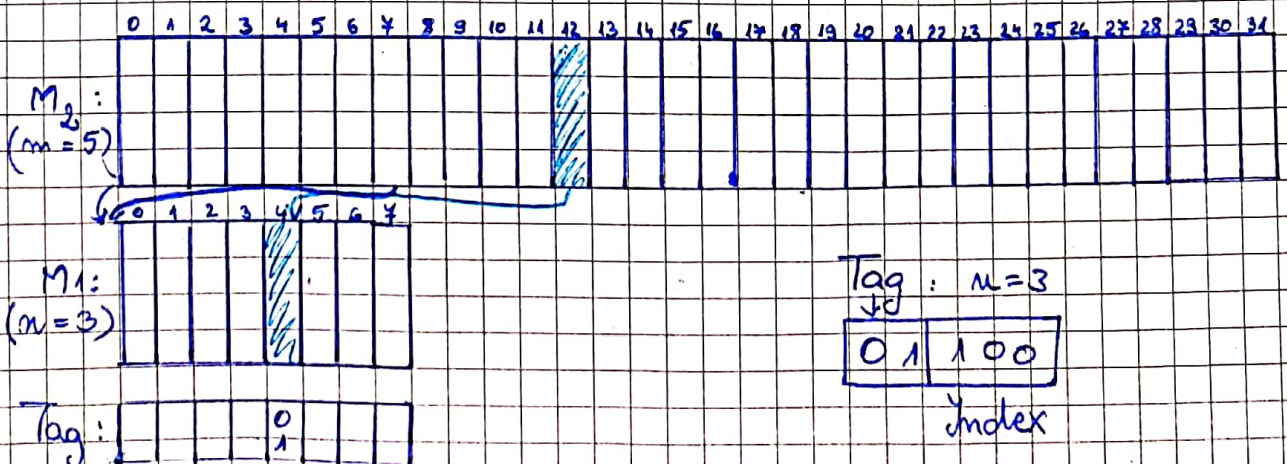
M_2: 2^m blocks, i-index; $i \in \{0 \dots 2^m - 1\}$

M_1: 2^n blocks, j-index; $j \in \{0 \dots 2^n - 1\}$

cu $m > n$

$$j \equiv i \bmod 2^n$$

ex: $m = 5$; $n = 3$



Tag: $n = 3$

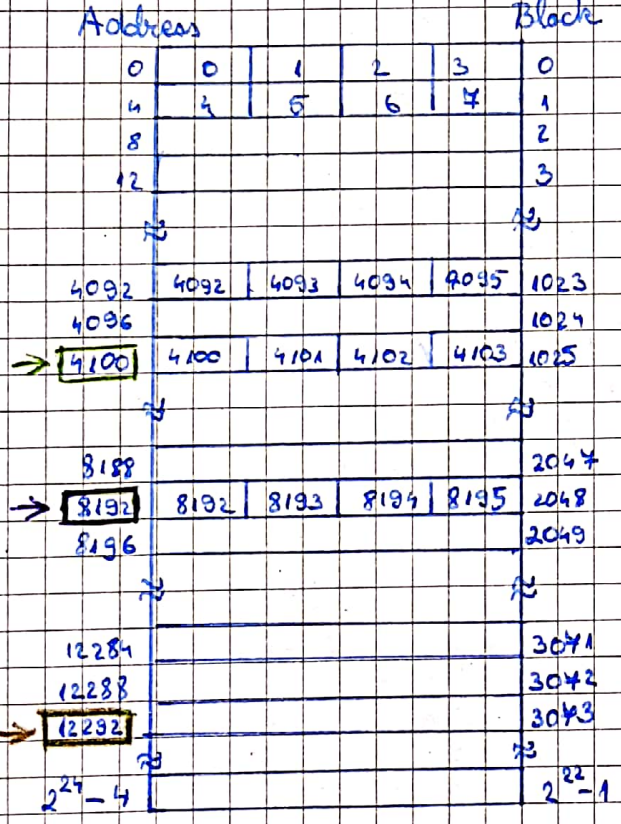
0 1	1 0 0
Tag	Index

Exemple 1:

Se consideră o memorie principală a cărei dimensiune e de 2^{24} cuvinte. Un cuvânt = 1 byte (p=0), un bloc e 4 cuvinte. Cum implementăm maparea directă a adresei specificate având o mem. ~~cha~~ cache de dimensiune 1k block?

La adresa 8192 avem o instrucțiune care are un cod ipotetic 230, la adr. 4100 codul 2715 și la adr. 12292 codul 170. Unde se mapează în cache?

$$4 = \frac{2^{24}\text{ words}}{2^2\text{ words}} = 2^{22}\text{ blocks}$$



M2

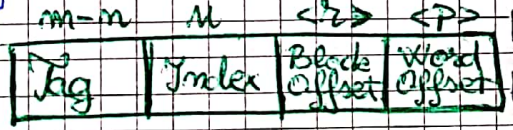
Cum arată adresa cuvântului din problemă?

Adresa va fi pe 24 de ~~bytes~~ biți

(24 x 1 byte)

↳ dim 2^{24}

În general:



La noi: Word Offset are lungimea 0 pt. că 1 word = 1 byte =>

=> p = 0.

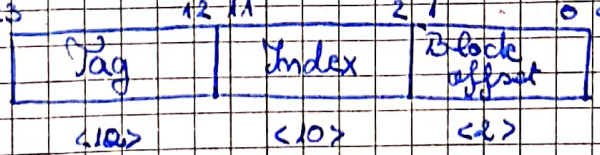
Cache size = 1 k blocks = 2^{10} blocks

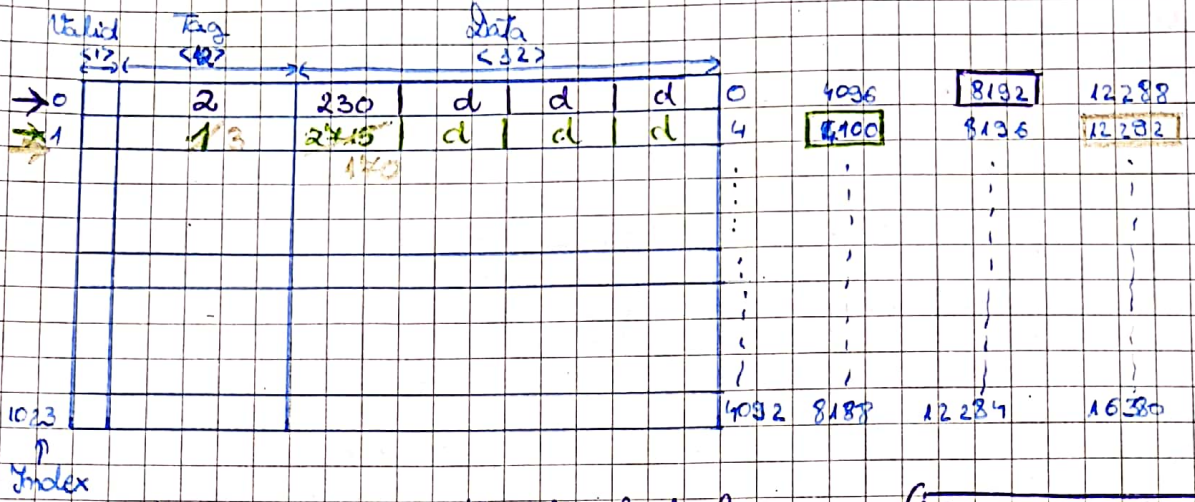
1 block = 2^2 words = 2^2 bytes = $2^2 \cdot 2^3$ = biți

1 block = 2^5 = 32 biți

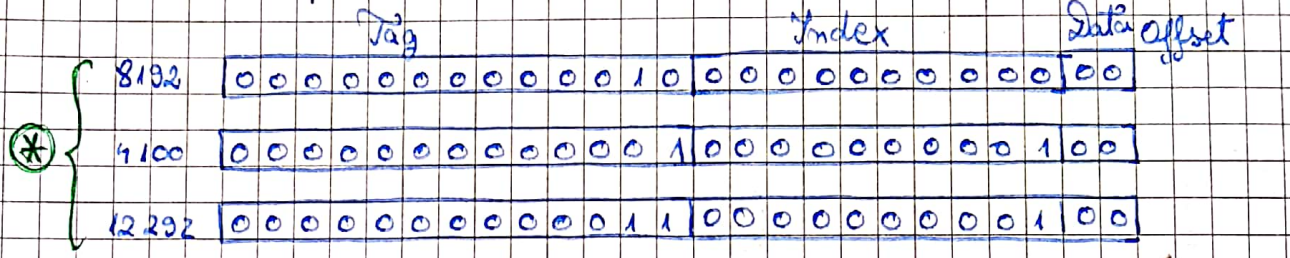
Deci, la noi:

Address:





→ aici se va mapa tot blocul de la 8192 ([8192 8193 8194 8195])

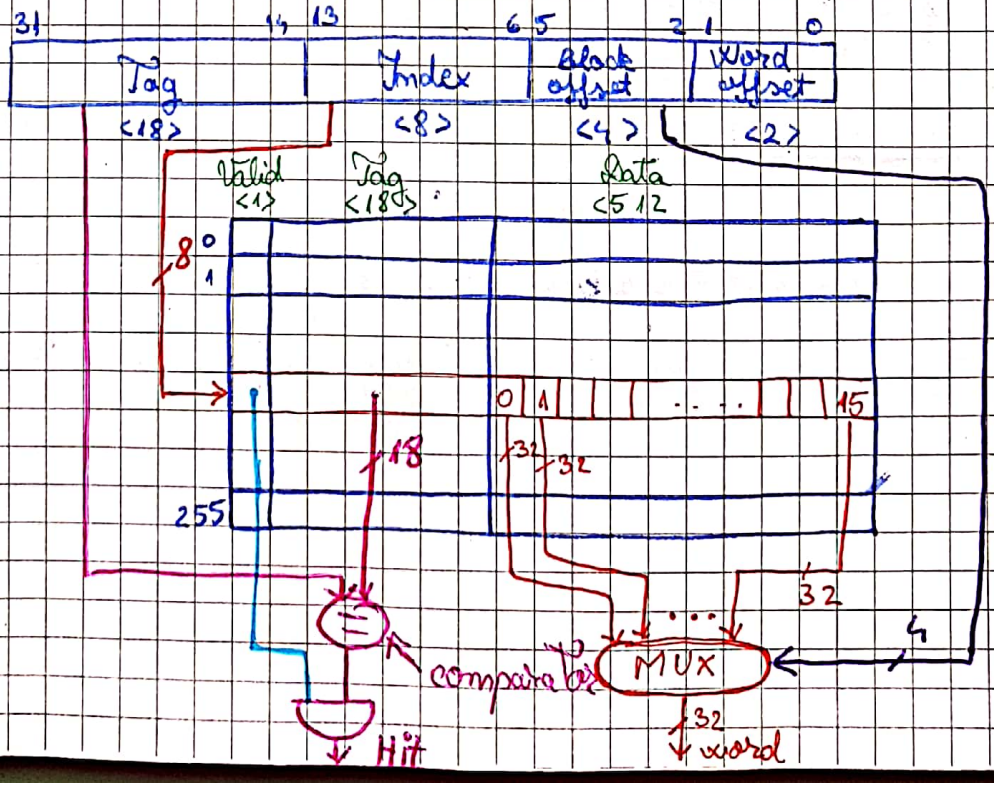


→ se va suprascrie de aici deja aici ce e la 12292

Example 2: Intrinsity's Fast Math.

1 block = $2^4 \times 2^2 \times 2^3 = 512$ biti
(words → bytes/word; biti/byte)

Data capacity = $2^8 \times 2^4 \times 2^2 = 2^{14}$ bytes = 16 kbytes
(block; words/block; bytes/word)



CURS 10 — 7.05.2019

3.3. Set-associative mapping

$M_2(0), M_2(1) \dots M_2(j) \dots M_2(2^m - 1)$

$M_1(0), M_1(1) \dots M_1(i) \dots M_1(2^n - 1)$, $m < n$ } blocuri din memoria principala

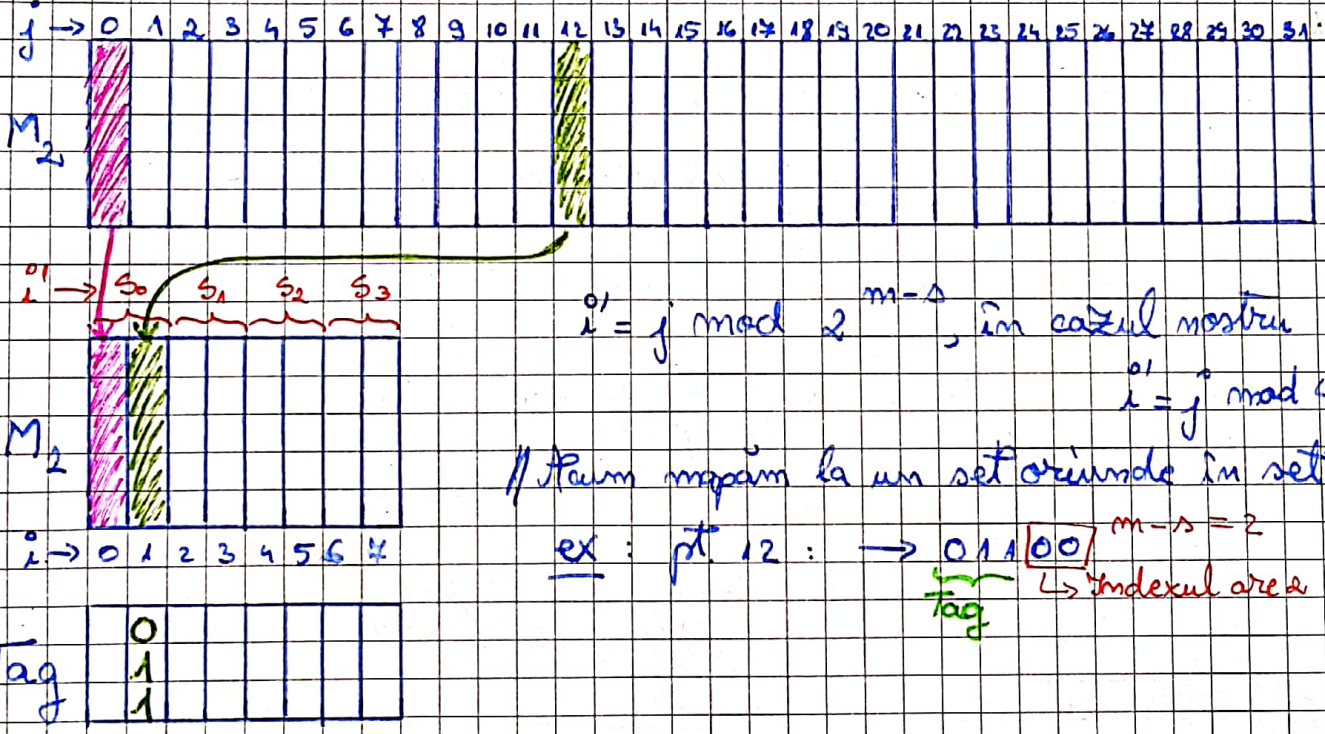
k-way set associative mapping

$k = 2^s$

$M_1'(0), M_1'(1), \dots M_1'(i') \dots M_1'(2^{m-s} - 1)$

Maparea se face: $i' = j \bmod 2^{m-s}$, $m' = m - s$

Example 1: $m = 3$; $n = 5$; $s = 1$ => M_2 : 32 blocuri



$i' = j \bmod 2^{m-s}$, in cazul nostru $i' = j \bmod 4$

// Facem mapam la un set oriunde in set

ex: pt 12: → 011 [00] $m - s = 2$ ↳ Indexul are 2 biti

Tag

Example 2: 2^{24} words - main memory (MM)

1 word = 1 B

1 block = 4 Kwords

Acasa: 2^{24} words = 2^{24} B = $\underbrace{2^4}_{16} \cdot 2^{20}$ B = 16 MiB ↳ mega

Address	M_2				Block number
0	0	1	2	3	0
4	4	5	6	7	1
8					2
12					3
≈					≈
4092					1023
4096					1024
4100	2715				1025
≈					≈
8188					2047
8192	230				2048
≈					≈
12284					3071
12288					3072
12292	170				3073
≈					≈
$2^{29}-4$					$2^{27}-1$

@ Address Code (de la ex. trecute)

8192 → 230
4100 → 2715
12292 → 170

} cu codificările ⊛ (cu 2 pagini mai înainte)

Cache : 1k-block

Cache size = 2^{10} blocks

Cache - 2 way Set Association (SA)

Index values = $\frac{2^{10}}{2^1} = 2^9$

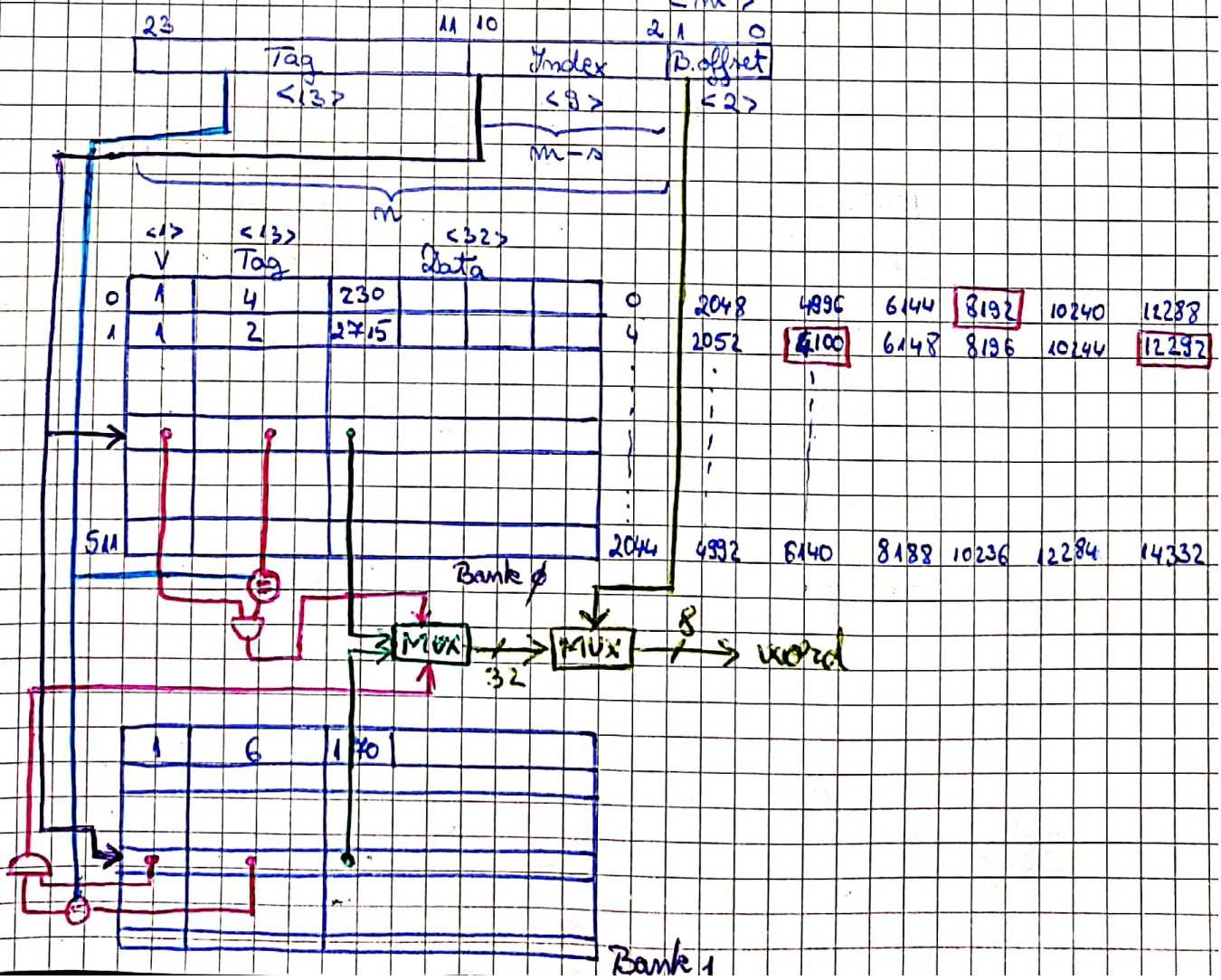
↑ set associativity

DM Address (Direct Mapping)

Tag	Index	B.offset	W.offset
<m-m'>	<m'>	<p>	<r>

2^s k-way SA Add.

Tag	Index	B.offset	W.offset
<m-m'+s>	<m'-s>	<p>	<r>



Example 3 : 4 - way SA

8 - sets

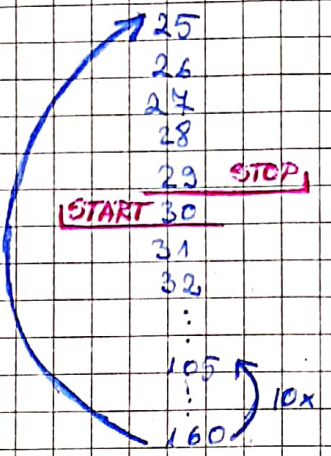
1 block = 4 words

Cache initially empty

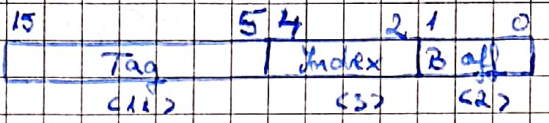
1 word = 1 byte

MM size = 2^{16} words

Instruction cache : $\frac{2^{16}}{2^2} = 2^{14}$ blocks



Address word format?



Saltul la instrucțiune se face de 10 ori. Care este Hit rate?

Bazându-ne pe tabelul de pe pagina următoare, obținem:

	V	Tag	Date
0	1	1	32 - 35
1	1	1	36 - 39
2	1	1	40 - 43
3	1	1	44 - 47
4	1	1	48 - 51
5	1	1	52 - 55
6	1	~~1~~ 0	~~56 - 59~~ 24 - 27
7	1	~~0~~	~~28 - 31~~ 156 - 159

Block 0

	V	Tag	Date
0	1	3	96 - 99
1	1	3	100 - 103
2	1	3	104 - 107
3	1	3	108 - 111
4	1	3	112 - 115
5	1	3	116 - 119
6	1	3	120 - 123
7	1	2	92 - 95

Block 2

$\begin{pmatrix} 1M \\ 1H \end{pmatrix} \begin{pmatrix} 1M \\ 3H \end{pmatrix} \times 32 \begin{pmatrix} 1M \\ 0H \end{pmatrix}$

$\begin{pmatrix} 0M \\ 360H \end{pmatrix} \cdot \begin{pmatrix} 1M \\ 2H \end{pmatrix} \begin{pmatrix} 1M \\ 1H \end{pmatrix}$

	V	Tag	Date
0	1	2	64 - 67
1	1	2	68 - 71
2	1	2	72 - 75
3	1	2	76 - 79
4	1	2	80 - 83
5	1	2	84 - 87
6	1	2	88 - 91
7	1	~~1~~	~~60 - 63~~
		0	28 - 31

Block 1

	V	Tag	Date
0	1	4	128 - 131
1	1	4	132 - 135
2	1	4	136 - 139
3	1	4	140 - 143
4	1	4	144 - 147
5	1	4	148 - 151
6	1	4	152 - 155
7	1	3	124 - 127

Block 3

$\begin{pmatrix} 36M \\ 661H \end{pmatrix}$

$\Rightarrow$ Hit rate $= \frac{661}{607} \approx 95\%$

M_2

Address	Block index
0 - 3	0
4 - 7	1
8 - 11	2
12 - 15	3
16 - 19	4
20 - 23	5
24 - 27	6
28 - 31	7
32 - 35	8
36 - 39	9
40 - 43	10
44 - 47	11
48 - 51	12
52 - 55	13
56 - 59	14
60 - 63	15
64 - 67	16
68 - 71	17
72 - 75	18
76 - 79	19
80 - 83	20
84 - 87	21
88 - 91	22
92 - 95	23
96 - 99	24
100 - 103	25
104 - 107	26
108 - 111	27
112 - 115	28
116 - 119	29
120 - 123	30
124 - 127	31
128 - 131	32
132 - 135	33
136 - 139	34
140 - 143	35
144 - 147	36
148 - 151	37
152 - 155	38
156 - 159	39
160 - 163	40

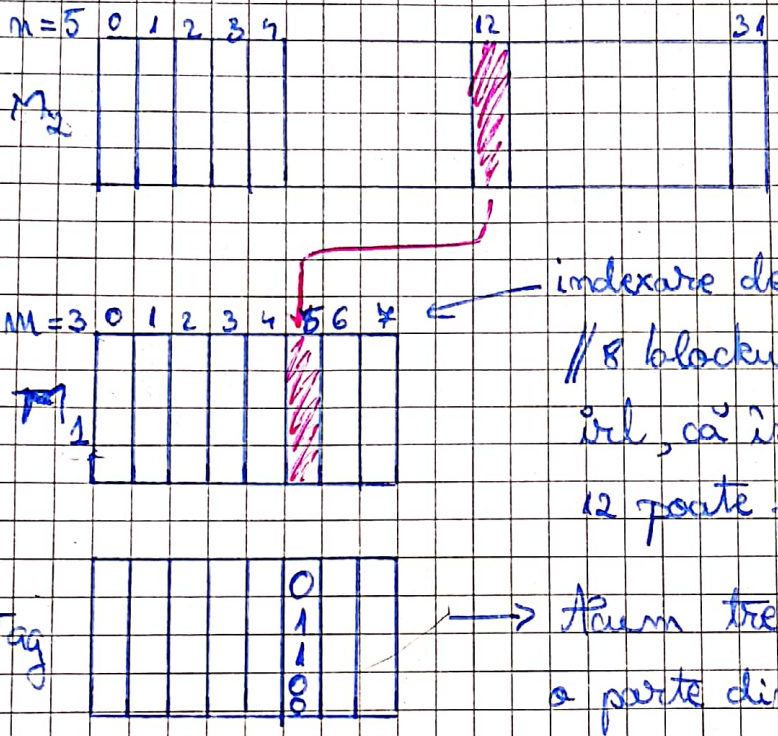
CURS 11 14.05.2019

3.4. Full associative mapping

k - associativity : $k = 2^s$

$s = 0 \Rightarrow$ direct mapping

$s = m \Rightarrow$ full associativity unde 2^m = nr. of blocks in cache



indexare de forma

// 8 blocuri ce nu mai sunt indexate ird, că is toate la același index.

12 poate fi mapat oriunde: ex: pe 5.

⟶ Acum trebuie scris tot nr, nu doar o parte din el.

Exemplu:

2^{24} words

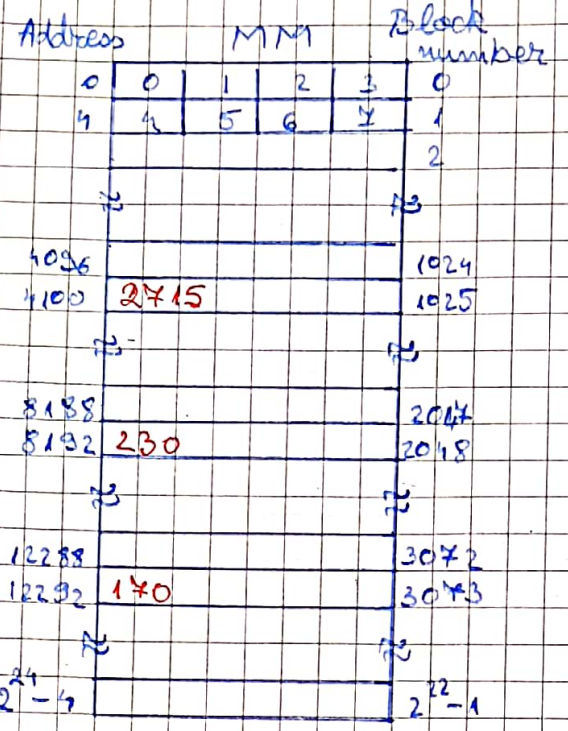
MM size = 16 MiB = 2^{24} B = $\frac{2^{24}\text{B}}{2^2\text{ B/block}}$ = 2^{22} blocks (*)

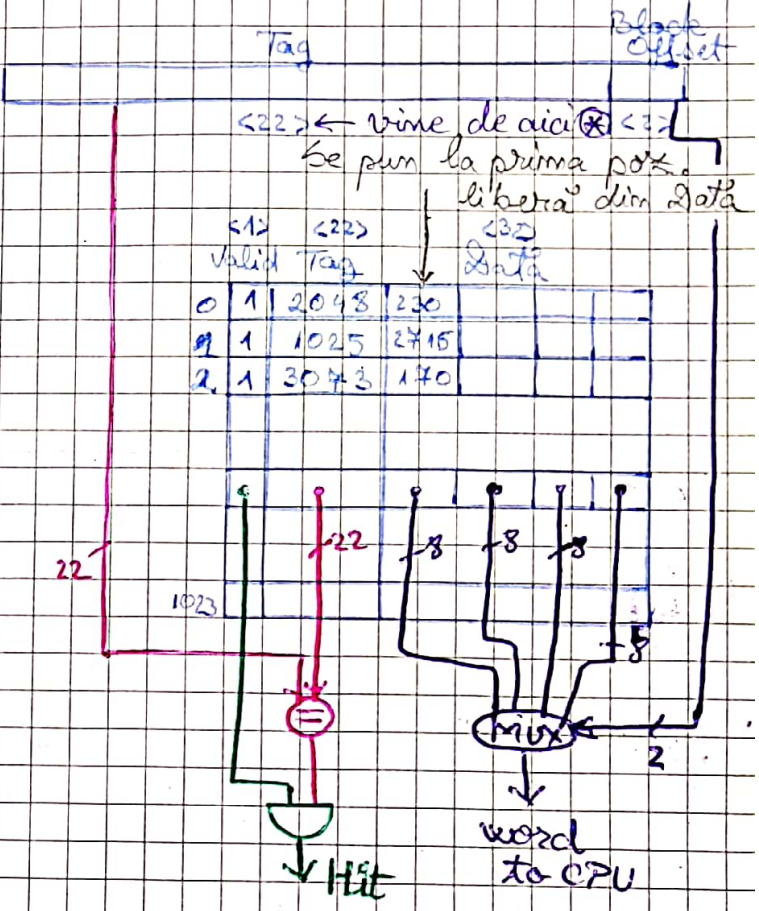
1 word = 1B

1 block = 4 words

Cache data size = 1 k blocks

Address	Code
8192	230
4100	2715
12292	170





3.5. Cache performance metrics

Miss: atunci când CPU nu găsește informația căutată în Cache

Miss Rate = 1 − Hit Rate

Hit: atunci când CPU găsește ce caută în Cache

Miss Causes
- Compulsory (Cold Start misses): missuri apărute când resetezi sistemul și cache-ul e gol
  - → soluție: bigger blocks
- Conflict: 2 blockuri se mapează la același index (ce se întâmplă la set-associative)
  - → soluție: crești set associativity
- Capacity: memorie Cache mică
  - → soluție: bigger cache

Average Memory Access Time (AMAT) =

$= T_{ac} \cdot \text{Hit rate} + (T_{ac} + \text{Miss Penalty}) \cdot \text{Miss rate}$

$= T_{ac} + \text{Miss Penalty} \times \text{Miss rate}$

T_{ac} = timp de acces la cache

Hit rate + Miss rate = 1

Miss Penalty - timpul pe care îl ia ~~o~~ informații să fie încărcată în cache.

Bigger block <-> Miss Rate scade, dar Miss Penalty crește

CPU_{time} (real ori ideal) = Clock cycles for program × CCT =

= (Execution clock cycles + Memory stall clock cycles) × CCT

= IC × (Exec. clock cycles per instruction + Memory stall clock cycles per instruction) × CCT

(Exec. clock cycles per instruction = CPI)

Memory stall clock cycles per instruction =

= Memory accesses per instruction × Miss Rate × Miss Penalty

(Memory accesses per instruction = Misses per instruction)

Reader:

$CPU_{time} = IC \times (CPI_{exec.} + \text{Memory accesses per instr.} \times \text{Miss Rate} \times \text{Miss Penalty}) \times CCT$

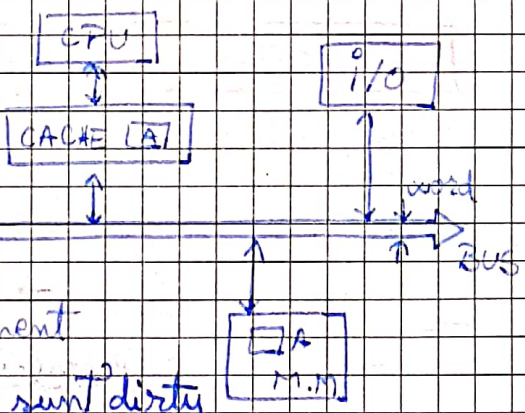
3.6. Write policies : 10% writes

Cache / Memory coherence → Update

a) Write through : implicit update

b) Write back : update on replacement

- dirty bit: dacă blocurile sunt dirty, bitul e 1 și doar acele blocuri sunt înlocuite



What happens in case of miss?

=> Allocate

α) Write Allocate

β) Write No Allocate

Pairings { Write through + Write No-Allocate
Write back + Write Allocate

3.4. Replacement policies (au sens doar la set și full asociativitate)

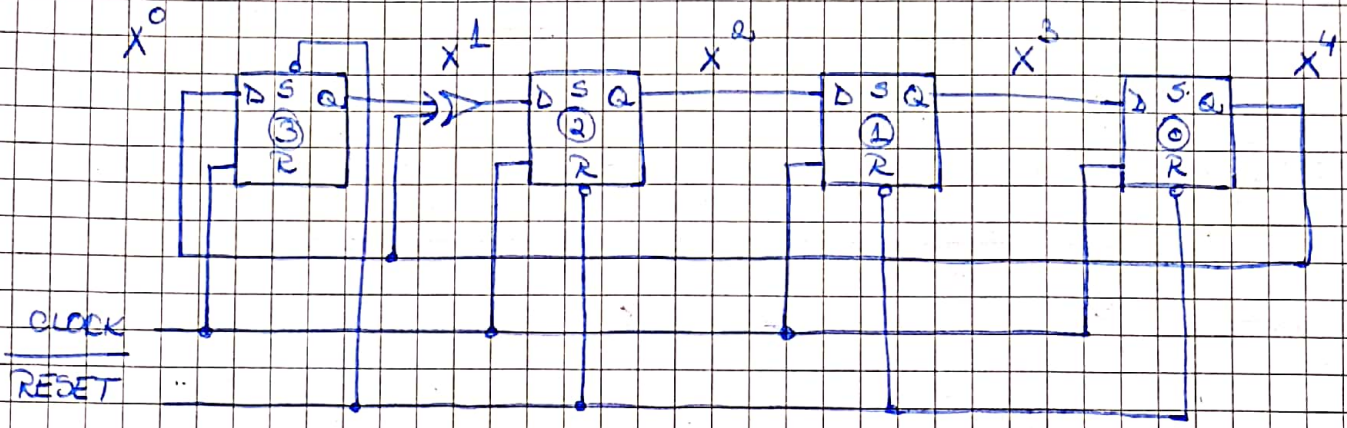
a) random : când SA e mică

b) FIFO

c) Least recently used – LRU

Modul de generare random a numerelor prin polinom indivizibil

$G(X) = X^4 + X + 1$



D_3	D_2	D_1	D_0	EZ
1	0	0	0	1
0	1	0	0	2
0	0	1	0	4
0	0	0	1	8
1	1	0	0	3
0	1	1	0	6
0	0	1	1	12
1	1	0	1	11
1	0	1	0	5
0	1	0	1	10
1	1	1	0	7
0	1	1	1	14
1	1	1	1	15
1	0	1	1	13
1	0	0	1	9
1	0	0	0	1

Perioada de repetiție : $2^4 - 1 = 15$ clock-uri

CURS 12 21.05.2019

3.8. Replacement policies

a) Random

b) FIFO

c) LRU - Least Recently Used

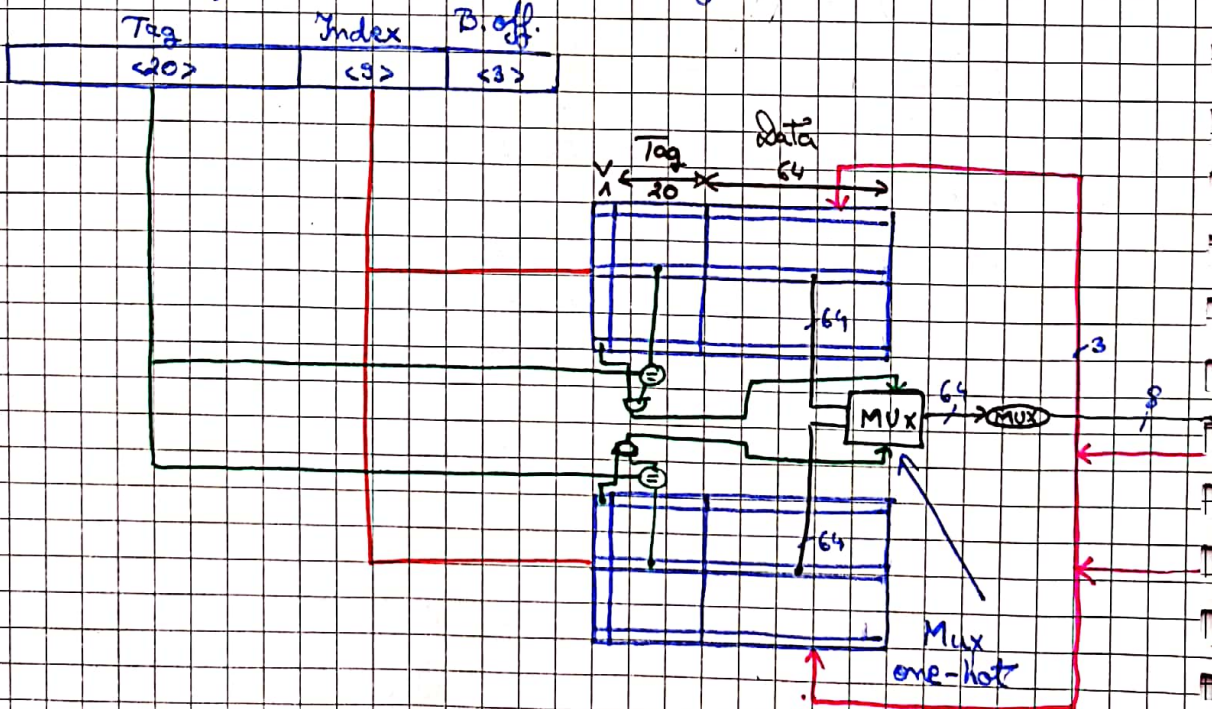
No. of "age" bits = $\log_2$ (set associativity)

Example 1: Pp. că avem un set asociativ pe 4 căi cu adrese în zecimal și pp. că toate adresele se mapează la același index

		Data 0	Data 1	Data 2	Data 3	Age 0*	Age 1*	Age 2*	Age 3*	
		0	0	0	0	0	0	0	0	
Miss →	14	14	0	0	0	0	0	0	0	
(cache M →	7	14	7	0	0	1	0	0	0	
gol) M →	6	14	7	6	0	2	1	0	0	
M →	12	14 (c.m. VECHI)	7	6	12	3	2	1	0	
M →	10	10	7	6	12	0	3	2	1	
(Hit) H →	6	10	7	6	12	1	3	0	2	← Cei cu vârstă mai mare rămân la fel, cei cu mai mică se incrementează
H →	7	10	7	6	12	2	0	1	3	
M →	14	10	7	6	14	3	1	2	0	
H →	7	10	7	6	14	3	0	2	1	
M →	3	3	7	6	14	0	1	3	2	
H →	14	3	7	6	14	1	2	3	0	

Aici de fapt e cuv. de la adresa 14, nu "14"

Example 2: Mașina Vax 11/780 + 2-way SA



2^3 words / block = 2^3 B / block = $2^3 \cdot 2^3$ bits / block = 64 bits / block

Cache Size (data) = s.a $\times 2^{index} \times$ block size = $2 \times 2^9 \times 2^3 B = 2^{13} B$ = 8 KiB

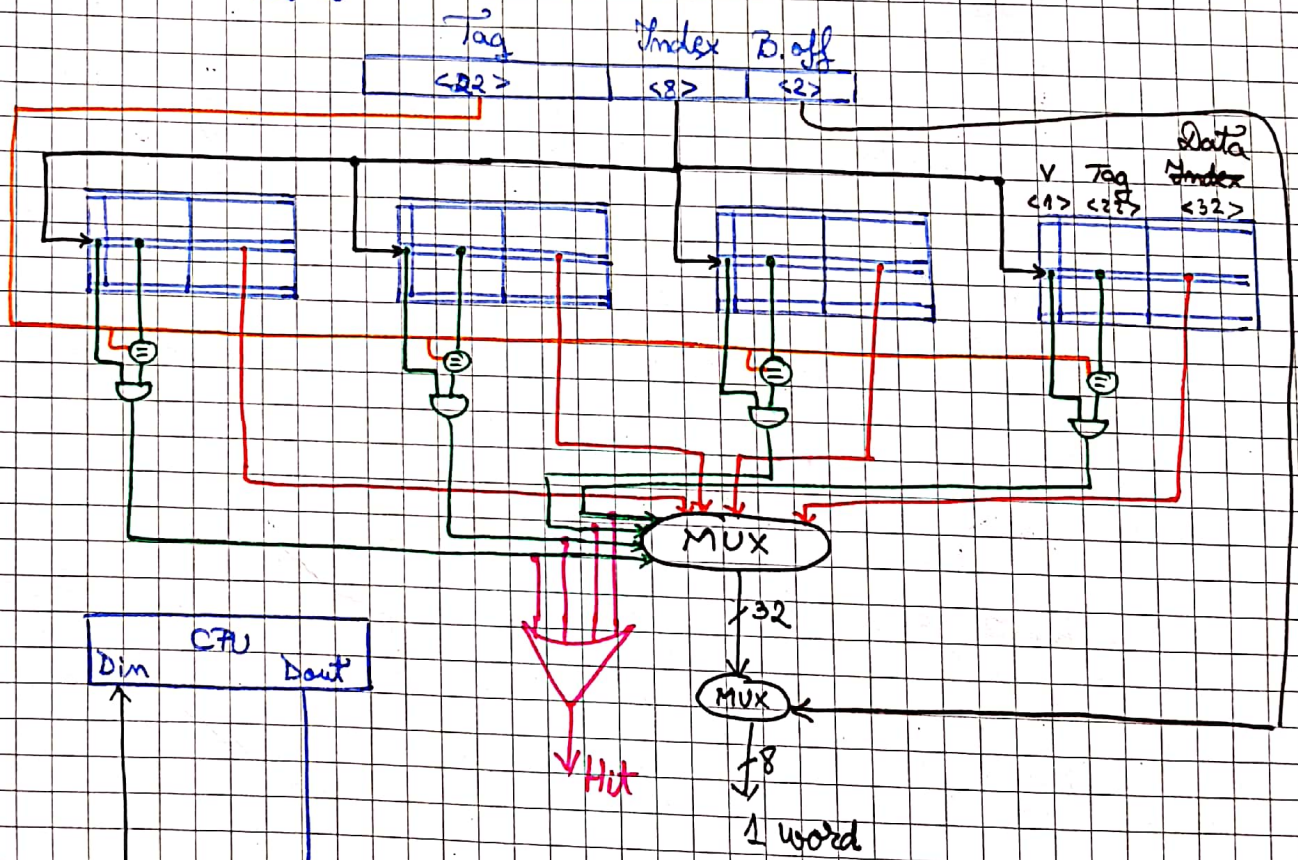
↑ set associativity

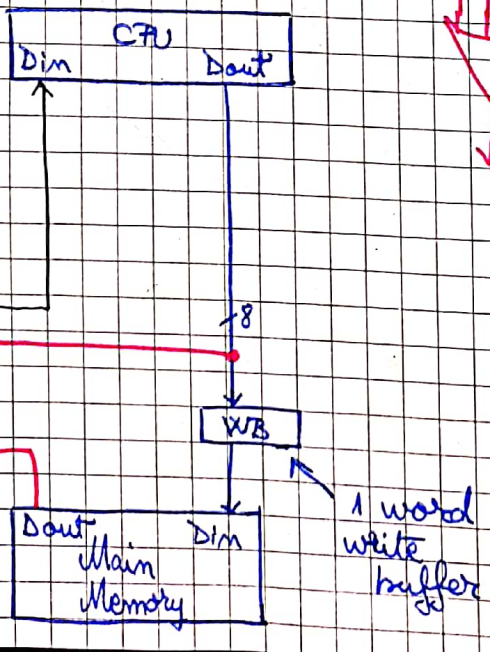
Write Policy → Write Through (Write Buffer) → No Allocate in case of a Write miss.

Example 3:

- 4 way - SA ; 1 word = 1 B ; 1 block = 4 words = 2^2 words
- Cache size = 2^{10} blocks = 1 KiBlocks
- MM size = 4 GiB

$$\frac{\text{No. of blocks}}{\text{s.a}} = \frac{2^{10}}{2} = 2^8 \text{ blocks / bank}$$





Example 4: Se dau 2 memorii cache diferite ale căror 2 sisteme de calcul de ~~ost~~ altfel identice.

Instruction miss rate ← ~~Index~~ ; Data miss rate

cache A: 2 way s.a ; $I_{miss} = 2\%$; $D_{miss} = 1,4\%$

cache B: 4 way s.a. ; $I_{miss} = 1,6\%$; $D_{miss} = 1,4\%$

A și B sunt mașini de calcul load/store, 20% din instrucțiuni fiind load/store.

Absolut miss penalty = 200 ns

$CCT_A = 20$ ns

La B, pt. că s.a e mai mare, mux e mai complicat, timpul e mai mare, deci CCT_B se mărește cu 10% (s-a degradat)

$CPI_{ideal} = 3$ clock cycle-uri

Care mașină e mai bună?

$$CPU_{time} = IC \times \left(CPI_{ideal} + \underbrace{\text{Memory accesses per instr.} \times \text{Miss Rate}}_{\text{Misses per instr.}} \times \text{Miss Penalty}\right) \times CCT$$

Misses per instr. $A = 1 \times I_{miss} + 0,2 \times D_{miss} = 0,02 + 0,2 \times 0,014 = 0,0234$ (20%)

Misses per instr. $B = 0,016 + 0,2 \times 0,014 = 0,0188$

Miss Penalty $A = \left[\frac{200 \text{ ns}}{20 \text{ ns}}\right] = 10$ clock cycles (CCT_A)

Miss Penalty $B = \left[\frac{200 \text{ ns}}{22 \text{ ns}}\right] = 10$ clock cycles (trb. să fie mereu nr. întreg)

$CCT_B = CCT_A + 20\% \cdot CCT_A$

$CPU_{time\ A} = IC \times (3 + 0,0234 \times 10) \times 20 \text{ ns} =$

$= IC \times 3,234 \times 20 = IC \times 64,64$ ns

(timpul mediu pt. o instrucțiune, luând în considerare imperfecțiunea cache-ului)

$CPU_{time\ B} = IC \times (3 + 0,0188 \times 10) \times 22 \text{ ns} = IC \times 70,136 \text{ ns}$

Example: Admitem un cache de 64 kBytes:

CCT = 20 ns

Media nr. de accese la instrucțiune: Memory access per instr. = 1,3

$CPI_{ideal} = 1,5$

Pt. acest cache, dacă se adoptă set asociativ, det. o degradare a frecv. clock-ului cu 8,5 %. În acest context, se analizează cache-ul în felul urm.: mapare directă vs. 2 way S.A.

Statistic: Miss rate $_{DM} = 3,9\%$

Miss rate $_{2\text{ way S.A}} = 3\%$

Care soluție e mai ok știind că în ambele cazuri, pt. Miss Penalty = 200 ns (expresie temporală).

În mod normal, Miss Penalty e exprimat în clockuri

$$AMAT = HCCT + Miss\ Rate \times \frac{Miss\ Penalty}{Time}$$

$$CPU_{time} = IC \times \left(CPI_{ideal} \times CCT + Mem.\ acc\ per\ instr. \times Miss\ Rate \times \frac{Miss\ Penalty}{Time}\right)$$

$$\frac{Miss\ Penalty}{Time} = Miss\ Penalty\ clocks \times CCT$$

$$AMAT_{DM} = 20\ ns + 0,039 \times 200\ ns = 20\ ns + 7,8\ ns = 27,8\ ns$$

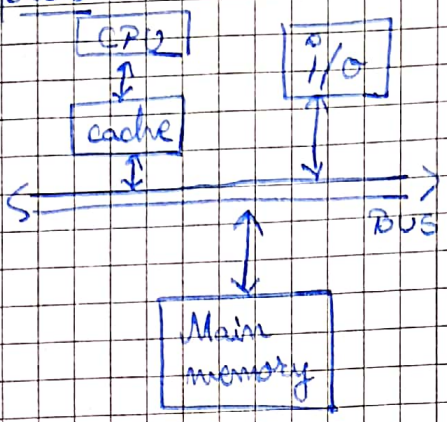
Better $AMAT_{SA2} = \underbrace{21,7}_{=20+8,5\%\cdot 20} + 0,03 \times 200\ ns = 27,7\ ns$

Better $CPU_{time\ DM} = IC \times (1,5 \times 20\ ns + 1,3 \times 0,039 \times 200) \cong IC \times 40,14\ ns$

$CPU_{time\ 2\ SA} = IC\ (1,5 \times 21,7\ ns + 1,3 \times 0,03 \times 200) \cong IC \times 40,35\ ns$

CPU_{time} ia în considerare mai mult ca AMAT, deci DM e mai bun.

Example 6:



Cât la sută din BUS Bandwidth este „mâncat" de imperfecțiunea cachelui?

Se dă un sist. de calcul cu:

BUS Bandwidth = 10^9 words/sec

Pp. că: Miss Rate = 10%

1 block = 4 words

CPU reference frequency = 10^8 words/sec (reads & writes)

BUS width = 2 words

Pp. că în (*) mom., 35% din blocurile cachelui au fost modificate (blockuri dirty). Cache are o politică write-allocate, frecvența scrierilor e de 30%.

Alternative de design, știind că Bandwidth e mâncat:

a) write back (WB)

b) write true (WT)

a) Nb. de accese la Bus datorită imperfecțiunii cache = $10^8 \times 0,1 \times \left(\%\text{Reads} \times \text{Read Miss Penalty} + \%\text{Writes} \times \text{Write Miss Penalty}\right)$

$= 10^8 \times 0,1 \times 2,7$

Read Miss Penalty $_{WB} = \left(\frac{\text{Block size}}{\text{Bus width}}\right) \times \text{Bus writes} \times 0,35 +$ ↳ blockurile dirty

↳ nr. de accese pe bus — Update

$+ \left(\frac{\text{Block size}}{\text{Bus width}}\right) \cdot \text{Bus reads}$ — Allocate

$= 2 \cdot 0,35 + 2 = 2,7 =$ Write miss penalty

% Bus Used $_{WB} = \frac{10^7 \cdot 2,7}{10^9} = \frac{2,7}{100} = 2,7\%$

b) Temă: WT

Nr. acc... = $10^8 \times 0,1 + (\%$ Reads $\times$ Read Miss Penalty + % Writes $\times$ $\times$ Write Miss Penalty) + $10^8 \times 0,9 \times \underbrace{0,3}_{\%\text{Writes}} \times \underbrace{\text{Write Hit Penalty}}_{=1}$

Read Miss Penalty = Write Miss Penalty =

$= \underbrace{2 \text{ BUS Reads}}_{\text{Allocate}} + 0,35 \cdot \underbrace{2 \text{ BUS Writes}}_{\text{Update for Main Memory}}$

CVRS 13

28.05.2019

b) Nr. acces = $\underbrace{10^8 \text{ words/second}}_{\text{Frequency of memory references}} \times \underbrace{0,1}_{\text{Miss Rate}} \times (0,3 \text{ Write Miss Penalty} +$

$+ 0,7 \text{ Read Miss Penalty}) + 10^8 \times 0,9 \times 0,3 \text{ Write Hit Penalty}$

Read Miss Penalty = 2 BUS Reads (nu mai updateaza)

Write Miss Penalty = 2 BUS Reads + 1 BUS Write =

= 3 BUS Accesses

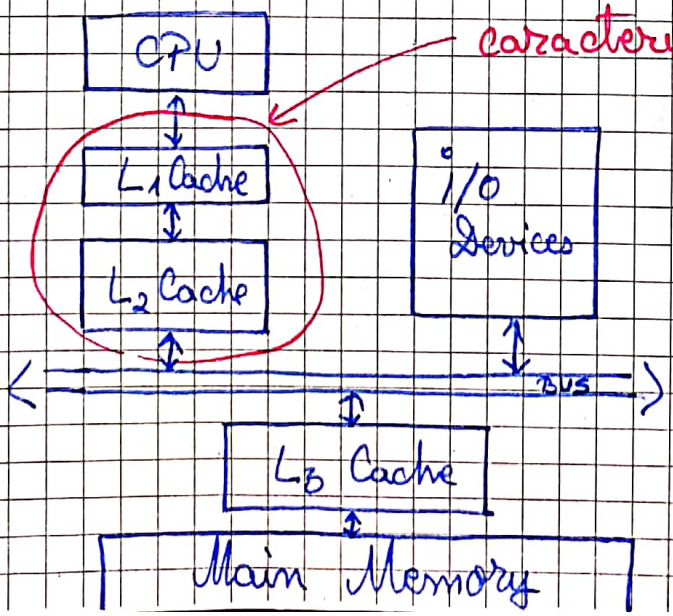
Write Hit Penalty = 1 BUS Write = 1 BUS Access

Asadar: Nr. acces: $10^8 \cdot 0,1 \, (0,3 \cdot 3 + 0,7 \cdot 2) + 10^8 \cdot 0,9 \cdot 0,3 \cdot \underline{1} =$

$= 10^7 \cdot 2,3 + 2,7 \cdot 10^7 = 5 \cdot 10^7$

% Bus Used$_{WT}$ $= \frac{10^7 \cdot 5}{10^9} = \frac{5}{100} = 5\%$

3.9. Reducing miss rate with multi-level caches



Nr. acces... = $10^8 \times 0,1 + (\%$ Reads $\times$ Read Miss Penalty $+ \%$ Writes $\times$ $\times$ Write Miss Penalty$) + 10^8 \times 0,9 \times 0,3 \times$ Write Hit Penalty

Writes $= 1$

Read Miss Penalty = Write Miss Penalty =

= 2 BUS Reads + 0,35 · 2 BUS Writes

Allocate — Update for Main Memory

CURS 13

28.05.2019

b) Nr. acces = 10^8 words/second $\times$ 0,1 $\times$ (0,3 Write Miss Penalty +

Frequency of memory references — Miss Rate

+ 0,7 Read Miss Penalty) + $10^8 \times 0,9 \times 0,3$ Write Hit Penalty

Read Miss Penalty = 2 BUS Reads (nu mai updateaza)

Write Miss Penalty = 2 BUS Reads + 1 BUS Write =

= 3 BUS Accesses

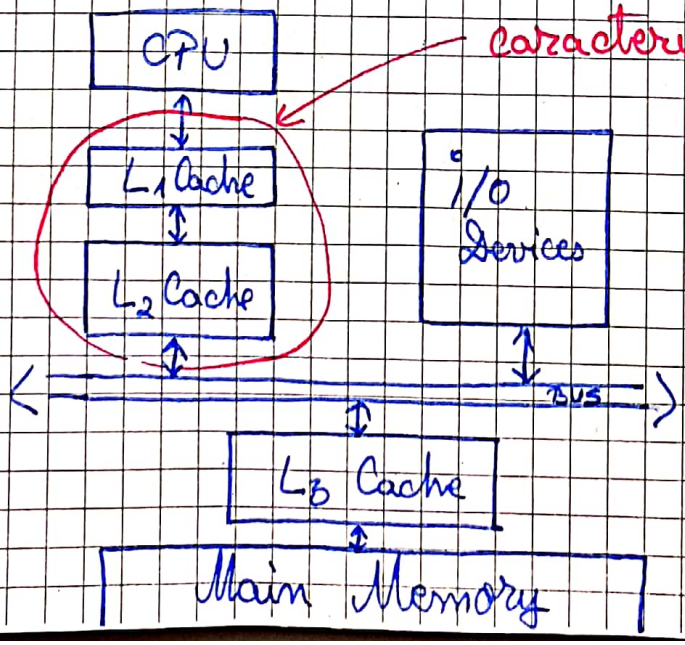
Write Hit Penalty = 1 BUS Write = 1 BUS Access

Acasa: Nr. acces: $10^8 \cdot 0,1\ (0,3 \cdot 3 + 0,7 \cdot 2) + 10^8 \cdot 0,9 \cdot 0,3 \cdot 1 =$

$= 10^7 \cdot 2,3 + 2,7 \cdot 10^7 = 5 \cdot 10^7$

% Bus Used $_{WT} = \frac{10^7 \cdot 5}{10^9} = \frac{5}{100} = 5\ \%$

3.9. Reducing miss rate with multi-level caches



Example 1: Se dă un sist. de memorie cu 2 nivele de cache unde CPI ideal = 1,0 clock cycle.

Clock Rate = 4 GHz

MM access time = 100 ns (incluzând Miss Penalty în MM)

Miss Rate per instruction = 2% (în L_1)

Se câte ori va fi mai rapid procesorul dacă adăugăm L_2 cache cu un timp de acces de 5 ns și pt. hit-uri și pt. miss-uri și e suficient de mare cât să reducă Miss Rate la MM la 0,5% din cazuri.

♥ Doar cu L1:

CPU time original = IC × (CPI ideal + Memory accesses per instruction × Miss Rate × Miss Penalty) × CCT

Misses per instruction = 0,2

$$\text{CPU time original} = IC \times (1 + 0{,}02 \times 400) \times 0{,}25\,ns = 2{,}25\,ns \times IC$$

$$CCT = \frac{1}{4 \cdot 10^9\,Hz} = 0{,}25\,ns \left(= \frac{1}{\text{Clock Rate}}\right)$$

$$\text{Miss Penalty} = \left\lceil \frac{100\,ns}{0{,}25\,ns} \right\rceil = 400\ c.c.\ (\text{clock cycles})$$

↳ găsește info în MM.

♥ L_1 cu L_2:

$$\text{CPU time}_{2L} = IC \times \left[CPI_{ideal} + (0{,}02 - 0{,}005) \times 20 + 0{,}005\,(400\ c.c. + 20\ c.c.) \right] \times 0{,}25\,ns = 0{,}85\,ns \times IC$$

Miss Penalty L_2 ↓

(0,02 − 0,005): info. nu nimerește în L_1 și hit-uie în L_2

0,005: info nu nimerește nici în L_1, nici în L_2

$$\text{Miss Penalty}_{L_2} = \left\lceil \frac{5\,ns}{0{,}25\,ns} \right\rceil = 20\ c.c.$$

↳ miss penalty când nu găsește în L_1 și găsește în L_2

Așadar: $\frac{\text{CPU time orig}}{\text{CPU time 2L}} = \frac{2,25}{0,85} = 2,64$

3.10. Virtual Memory Mechanism

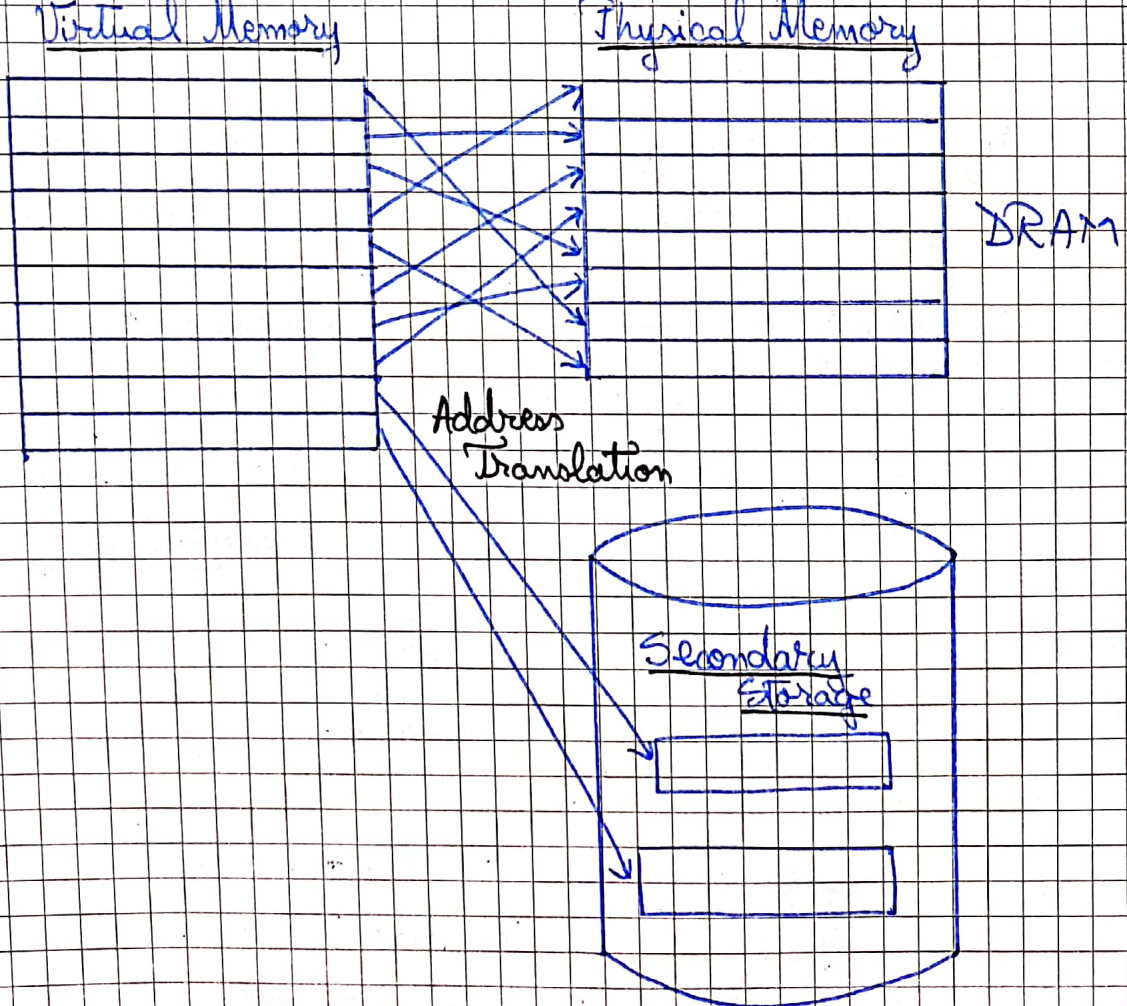
Virtual Machines, probleme puse pt. implementarea lor:

1. Protect processes from one another
2. Share the limited memory among processes (VMs)

Pentru asta, programatorii foloseau overlay, dar acum se foloseste o masină virtuală ce foloseste un spatiu de memorie virtuală (SMV) ce urmează a fi mapat pe adresele fizice.

SMV se fragmentează în: → Segments (variable size)
→ Pages (fixed size) ← cel care ne interesează pe noi

> Mecanismul de memorie virtuală paginat:

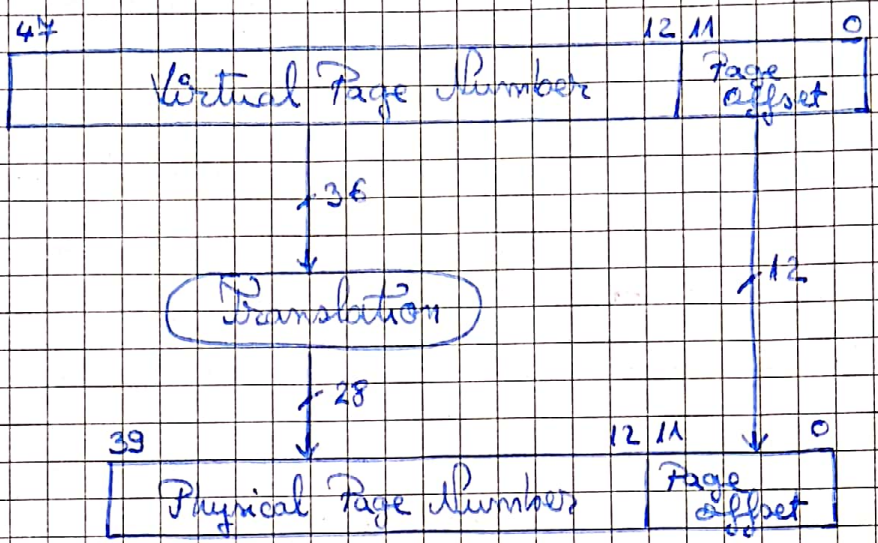


- Exemplu pt. Address Translation pe ARM V8:

64 Bit Address word
16 MSB are unused

64 - 16 = 48 - 12 = 36

40 - 12 = 28



Page size = 2^{12} B = 4 KiB (B = Bytes) ← pe 12 biți exprimăm adresa unui octet

Physical memory size = 2^{40} B = 1 TiB = 2^{28} pages = 256 Mi Pages

Virtual memory size = 2^{48} B = 256 TiB = 2^{36} pages = 64 Gi Pages

Paginile virtuale se mapează în memoria fizică doar când procesul este activ (posedă CPU la un moment dat)

Denumiri: page → granule ← Memory Management Unit

page fault → MMU exception

↳ când pagina nu e în MM și trebuie adusă de pe disc prin mecanismul de memory swapping

Problemă luată în considerare pt. construirea unui MMU: Misses cost a lot! (Secondary storage is 100 000 x slower than MM). Rezolvări:

(1) relatively big pages: 4 KiB ÷ 64 KiB

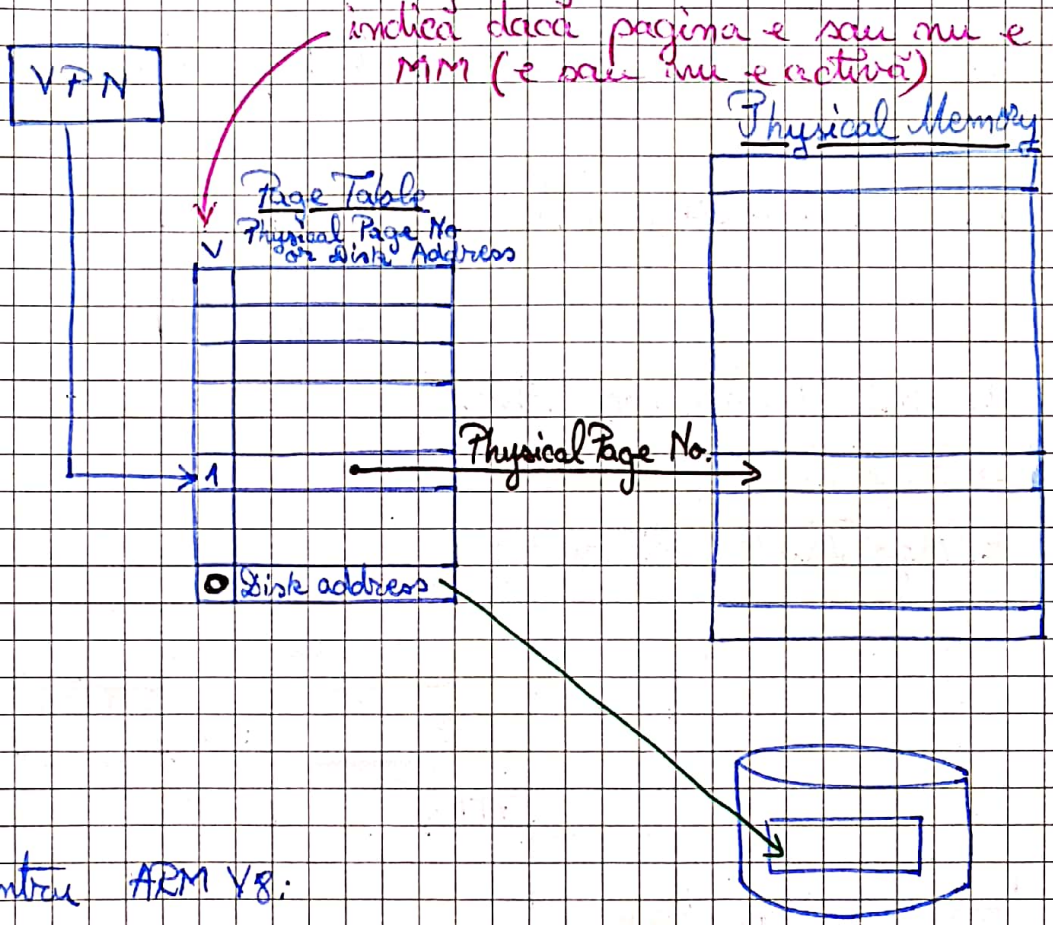
Servers → 32 KiB ÷ 64 KiB

Embedded → 1 KiB ÷ 4 KiB

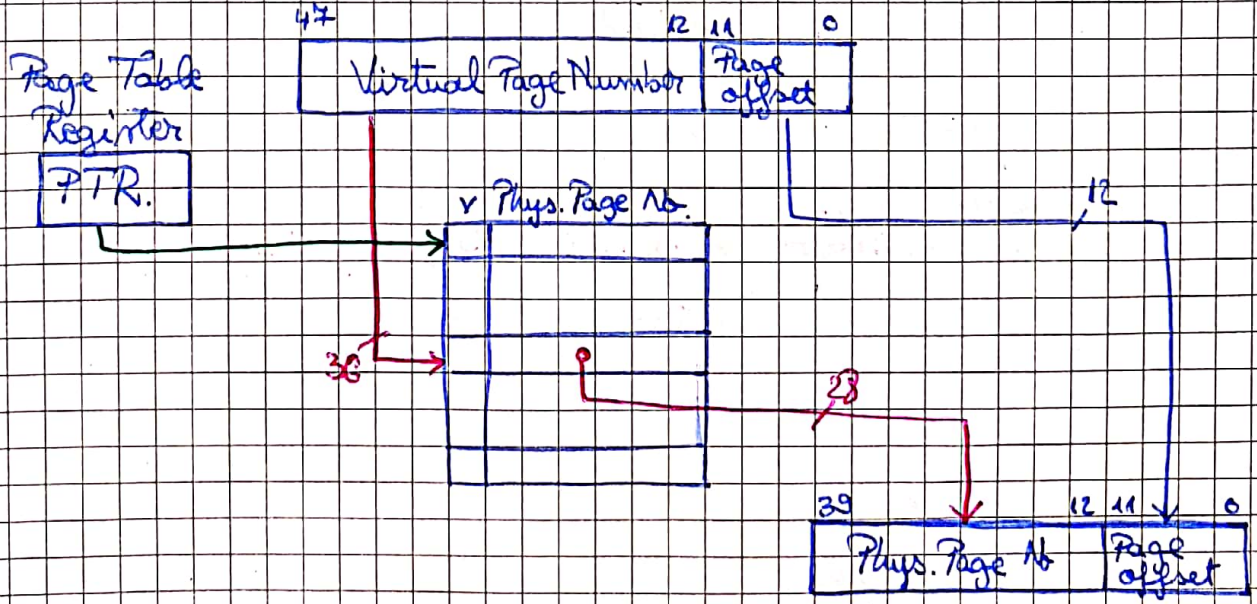
② WT e imposibil -> Folosim doar WB

③ Replacement is decided in software (nu mai folosesc metode random) - approximate LRU

④ Nu îți poți permite să cauți tag-uri ->
=> Folosim doar full-associativity mapping cu Page Tables



♥ Pentru ARM V8:



Problems: (1) Page Table Size

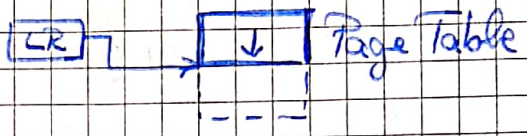
ARM - 1 PT Entry = 64 bits = 2^3 B

$2^{36} \times 2^3 = 2^{39} = 0{,}5$ TiB

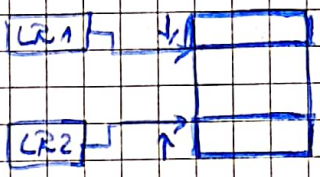
(2) Page Table Reads (Page Table is placed in the Physical Memory)

Solutions:

(1) A. Limit register



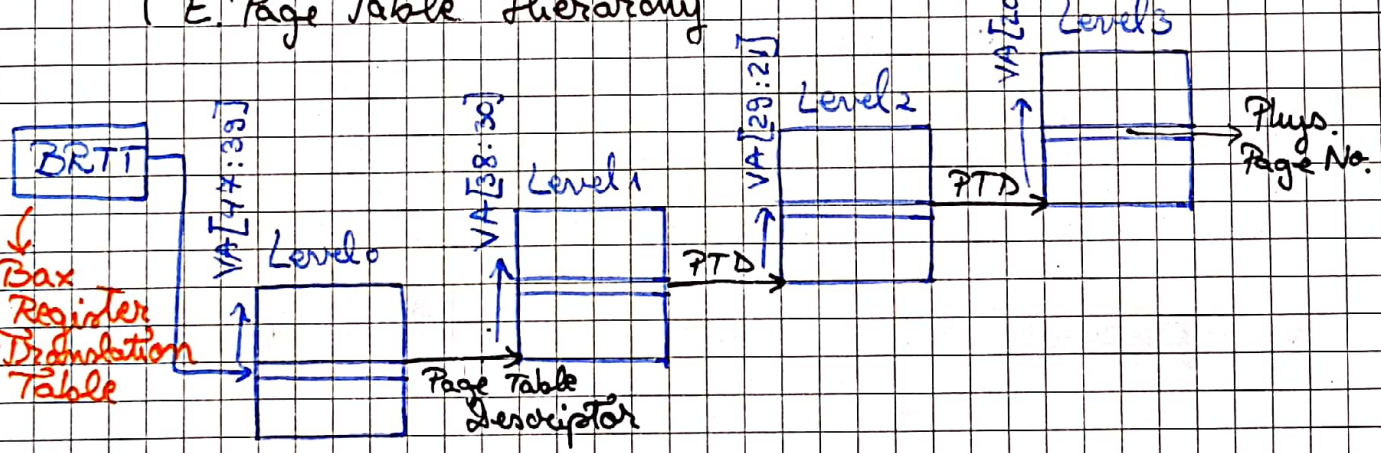
B. Heap & Stack growth



C. Hash (inverse page table)

D. Page the Page Table

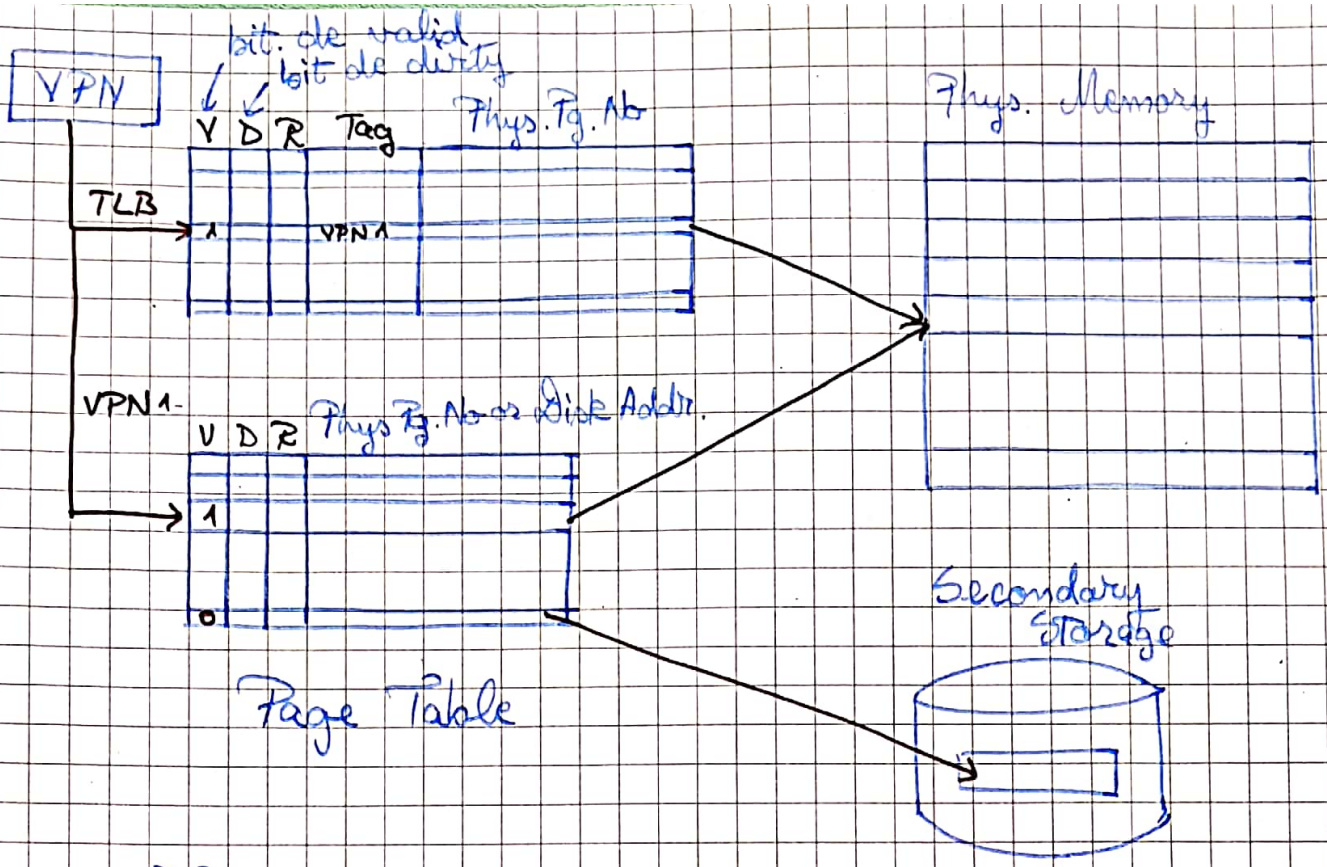
E. Page Table Hierarchy



$2^9 \times 2^3 B = 2^{12} B = 4$ KiB

(2) Translation Lookaside Buffer (TLB)

- Cache the Page Table



Fast MATH