

ooooooo
ooooooooo

oo
oo
oo
oooo

o
oo
oooooooooooo
oooo
oooooooooo
oooooooooooooooooooo

Architecture des ordinateurs

Jérémy Fix

CentraleSupélec

jeremy.fix@centralesupelec.fr

2017-2018

```

ooooooo
ooooooooo

```

```

oo
oo
oooo

```

```

o
oo
oooooooooooo
oooo
oooooooooooo
oooooooooooooooooooo

```

Architecture des ordinateurs

ou plutôt

Réalisation **électronique** d'une machine **programmable**
manipulant des représentations **numériques**

Electronique :



Programmable :

```

LDai 0x0010
STA 0x1000
for i in range(10):
    res = res + v[i]

```

Numériques 01100100100100100010101 A92FFC04

Particularités de ce cours

- pas d'examen
- les travaux de laboratoire ne sont pas notés
- TD (1h30) regroupés en BE (3h), sur machine

Equipe pédagogique



J. Fix



H. Frezza-Buet



J.L. Gutzwiller

Philosophie du cours

Introduire les concepts sous-jacents à n'importe quelle architecture sans être un cours sur une architecture spécifique (ARM, Intel, MIPS, ..)

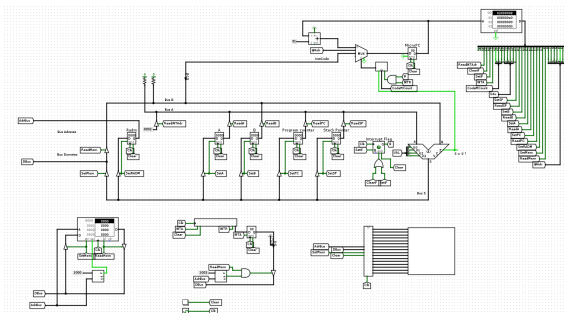
Abstraction progressive allant du bit à une architecture exécutant un jeu vidéo

Au programme

- 4 x 1h30 \Rightarrow représentations numériques, électronique, premier chemin de données, séquenceur
- 1 BE (3h) et 1 TL (4h30)
- 2 x 1h30 \Rightarrow programmation
- 1 BE (3h) : Programmation
- 1 x 1h30 \Rightarrow mémoires, périphériques et interruptions
- 1 BE (3h) et 1 TL (4h30) : périphériques et ordonnanceur

On va introduire :

- le codage binaire,
- les transistors (→ ELAN - J. Maufroy)
- les systèmes logiques (→ SLEA - Y. Houzelle)
- la programmation (→ FISDA - F. Pennerath)
- les périphériques, interruptions



ooooooo
ooooooooo

oo
oo
oooo

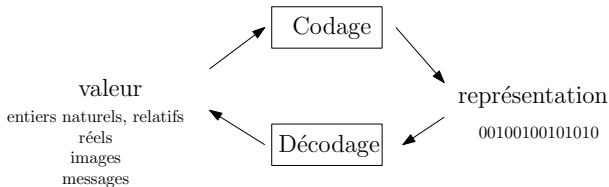
o
oo
oooooooooooo
oooo
oooooooooo
oooooooooooooooooooo

Space Invaders (1978)

Codage et opérations binaires

Représenté et représentant

Une valeur, quelle que soit sa nature, doit être représenté en binaire :



Bonne représentation ?

- facile à coder/décoder/manipuler, avec ou sans pertes
- robuste aux perturbations, compact

Représentation des entiers naturels

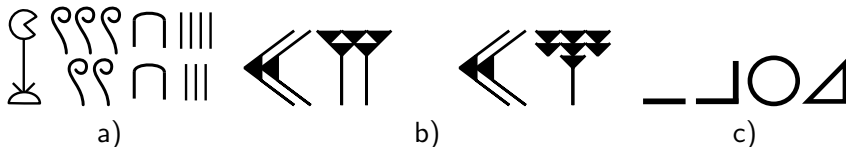


Figure: a) Système additif des égyptiens, un symbole par puissance de 10; 1527; b) système mixte additif/positionnel des mésopotamiens en base 60 : $1527 = (20 + 5) * 60^1 + (20 + 7) * 60^0$; c) système positionnel des Shadoks : Bu-Zo-Ga-Mu $99 = 1.4^3 + 2.4^2 + 0.4 + 3$

```

○○●○○○
○○○○○○○○

```

```

○○
○○
○○○○

```

```

○
○○
○○○○○○○○○○
○○○
○○○○○○○○
○○○○○○○○○○○○○○○○

```

Représentation des entiers naturels

En base 10 :

$$34 = 3.10^1 + 4.10^0$$

En base p :

$$(a_{k-1}a_{k-2} \cdots a_1a_0)_p = \sum_{i=0}^{k-1} a_i p^i$$

avec $\forall i, a_i \in [0, p-1], a_{k-1} \neq 0$

Représentation des entiers naturels

Comment passer de la **représentation** base p à la **valeur** en base 10 ?

$$(a_{k-1}a_{k-2} \cdots a_1a_0)_p = \sum_{i=0}^{k-1} a_i p^i$$

avec $\forall i, a_i \in [0, p-1], a_{k-1} \neq 0$.

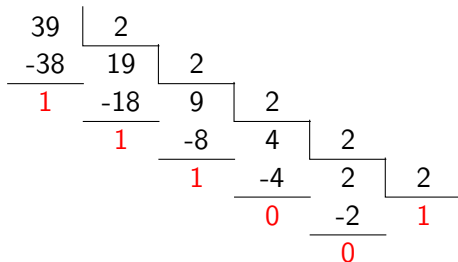
Comment passer de **la valeur** n en base 10 à **la représentation** en base p ?

$$\sum_{i=0}^{k-1} a_i p^i = a_0 + p \cdot \left(\sum_{i=0}^{k-2} a_{i+1} p^i \right)$$

\Rightarrow division euclidienne par p

Représentation binaire, $p=2$: $39 = 100111_2$

$$\sum_{i=0}^{k-1} a_i p^i = a_0 + p \cdot \left(\sum_{i=0}^{k-2} a_{i+1} p^i \right)$$



Puissance de 2	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
39_{10}	1	0	0	1	1	1

```

oooo●o
oooooooo

```

```

oo
oo
oooo

```

```

o
oo
oooooooooooo
oooo
oooooooooooo
oooooooooooooooooooo

```

Représentation hexadécimale, $p=16$: $39 = 27_{16}$

$$\begin{array}{r|l}
 39 & 16 \\
 \hline
 -32 & \textcolor{red}{2} \\
 \hline
 \textcolor{red}{7} &
 \end{array}$$

←

Ou, à partir de la représentation binaire :

$$39_{10} = \overbrace{(0010)}^{(2)} \overbrace{0111}^{(7)}$$

Décimal codé binaire et code de gray

Codage décimal codé binaire :

$$421 = \overbrace{0100}^4 \overbrace{0010}^2 \overbrace{0001}^1$$

Codage de Gray :

valeur	représentation binaire	représentation de Gray
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Addition

Exemple : $001 + 011$;

<i>a</i>	<i>b</i>	Retenue	Reste	<i>a</i>	<i>b</i>	<i>r</i>	Retenue	Reste
0	0	0	0	0	0	0	0	0
0	1	0	1	0	1	0	0	1
1	0	0	1	1	0	0	0	1
1	1	1	0	1	1	0	1	0
				0	0	1	0	1
				1	0	1	1	0
				1	1	1	1	1

- Algorithme d'addition de représentations non signées
- Précision finie $\Rightarrow n \in [0, 2^k - 1]$ et bit de carry

```

○○○○○○○
●○○○○○○○

```

```

○○
○○
○○○○

```

```

○
○○
○○○○○○○○○○
○○○
○○○○○○○○
○○○○○○○○○○○○○○○○

```

Autres opérations

Soustraction :

- $a - b$, $a \geq b$
- posée avec emprunt de retenue (e.g. $100_2 - 001_2$)

Multiplication :

- addition et décalage (e.g. 39×5)

Division :

- posée (e.g. $39/5$)

Entiers négatifs ?!


```

○○○○○○○
○○●○○○○○○○

```

```

○○
○○
○○○○

```

```

○
○○
○○○○○○○○○○○○
○○○○
○○○○○○○○○
○○○○○○○○○○○○○○○○○○

```

Représentation des entiers relatifs

Codage par décalage (excess-K)

- $n \in [-2^{k-1}, 2^{k-1} - 1], K = 2^{k-1}$
- Codage : $x = EncUB_k(n + K), n + K \in [0, 2^k - 1]$
- Décodage : $n = DecUB(x) - K = \sum_{i=0}^{k-1} x_i 2^i - 2^{k-1}$

n	-4	-3	-2	-1	0	1	2	3
x	000_{2K}	001_{2K}	010_{2K}	011_{2K}	100_{2K}	101_{2K}	110_{2K}	111_{2K}

- addition particulière $(-4 + 1, k=3)$. comparaison facile.

Représentation des entiers relatifs

Codage par valeur signée (sign-magnitude)

- $n \in [-2^{k-1} + 1, 2^{k-1} - 1]$
- un bit de signe, k-1 bits de valeur

n	-3	-2	-1	0	1	2	3
x	111_{2s}	110_{2s}	101_{2s}	100_{2s} ou 000_{2s}	001_{2s}	010_{2s}	011_{2s}

- 2 représentations de 0. Addition particulière ($1 + (-1)$, $k=3$).

○○○○○○○
○○○○●○○○○

○○
○○
○○○○

○
○○
○○○○○○○○○○
○○○○
○○○○○○○○
○○○○○○○○○○○○○○○○

Représentation des entiers relatifs

Codage par complément à deux (2's complement)

- $n \in [-2^{k-1}, 2^{k-1} - 1]$

n	-4	-3	-2	-1	0	1	2	3
1C	(011)	100	101	110	111 ou 000	001	010	011
2C	100	101	110	111	000	001	010	011

- les opérations sont les mêmes qu'avec les représentations non signées
- comparaison : 1) tester le signe 2) sinon la valeur
- une seule représentation du 0
- débordement (*overflow*); e.g. $k=3$: $3 + 3$; $(-3) + (-3)$, vérifiable avec les bits de retenue (01 ou 10)

○○○○○○○
○○○○○●○○○

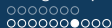
○○
○○
○○○○○

○
○○
○○○○○○○○○○○
○○○○
○○○○○○○○○
○○○○○○○○○○○○○○○○○

Représentation des nombres réels

Virgule fixe (fixed-point)

- $26.5 = 2 \times 10^1 + 6 \times 10^0 + 5 \times 10^{-1}$
- Codage du "." ? $26.5 = 11010.1_2$; Convention $Q < n_e > < n_f >$
- Codage : représentation non signée de $n \times 2^{n_f}$
- Décodage : $n = \sum_{i=0}^{k-1} a_i 2^{i-n_f} = 2^{-n_f} \sum_{i=0}^{k-1} a_i 2^i$
- complément à deux (e.g. $5.25 - 3.5$; $-5.25 + 3.5$ en Q4.2)
- même opérations arithmétiques que pour les entiers (DSP)
- mais précision uniforme, non représentation de valeurs particulières (e.g. ∞ , NaN, ..)



Représentation des nombres réels

Virgule flottante IEEE 754-2008 (floating-point)

- Notation scientifique en base p :

$$x = \pm m \times p^e, \quad m \in [0, p[, \quad e \in \mathbb{Z}$$
- $1245 = 1.245 \times 10^3 = 0.1245 \times 10^4 = 0.01245 \times 10^5$
- représentations dénormalisées ($m = 0$); représentation normalisée ($m \neq 0$)
- En binaire, $m \in [0, 2[$; Étendu $\mathbb{R} \cup \{-\infty, \infty\} \cup \{\text{sNaN}, \text{qNaN}\}$

signe S (1 bit)	exposant E (n_e bits)	mantisse M (n_m bits)
-------------------	----------------------------	----------------------------

M code uniquement la partie fractionnaire. E en excess- K .
 Plusieurs conventions : binary-16, binary-32, binary-64

Représentation des nombres réels

Binary-16

Représentation sur 16 bits : 1 bit de signe, $n_e = 5$ bits pour l'exposant, $n_m = 10$ bits pour la mantisse

Représentation binary-16			Valeur représentée	Note
Signe	Exposant	Mantisse		
0	00000	0...0	+0	
1	00000	0...0	-0	
s	00000	0...01	$(-1)^s 2^{-14} 1.2^{-10} = (-1)^s 2^{-24}$	Plus petit réel dé
s	00000	0...10	$(-1)^s 2^{-14} 2.2^{-10} = (-1)^s 2^{-23}$	Second plus pe
s	00000	1...11	$(-1)^s (2^{-14} - 2^{-24})$	Plus grand réel dé
s	00001	0...00	$(-1)^s 2^{1-15} = (-1)^s 2^{-14}$	Plus petit réel n
s	11110	1...11	$(-1)^s 2^{15} (2 - 2^{-10})$	Plus grand réel n
s	11111	0...00	$(-1)^s \infty$	
x	11111	M, $M \neq 0$	NaN (sNaN ou qNaN)	Exception, e.g

```

○○○○○○○
○○○○○○○○●○

```

```

○○
○○
○○○○

```

```

○
○○
○○○○○○○○○○
○○○○
○○○○○○○○
○○○○○○○○○○○○○○○○○○

```

Opérations arithmétiques sur les représentations non signées

Représentation des caractères (ASCII, ISO-8859, UTF-8)

ASCII CONTROL CODE CHART

BIT 7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
hex		DEC		DEC		DEC		hex		DEC		DEC		DEC	
BITS		CONTROL		SYMBOLS		UPPER CASE		LOWER CASE							
				NUMBERS											
0 0 0 0	NUL	0	DLE	SP	0	0	0	P	'	0	0	0	0	0	0
0 0 0 1	SOH	1	DC1	!	1	A	Q	a	q	1	1	1	1	1	1
0 0 1 0	STX	2	DC2	"	2	B	R	b	r	2	2	2	2	2	2
0 0 1 1	ETX	3	DC3	#	3	C	S	c	s	3	3	3	3	3	3
0 1 0 0	EOT	4	DC4	\$	4	D	T	d	t	4	4	4	4	4	4
0 1 0 1	ENQ	5	NAK	%	5	E	U	e	u	5	5	5	5	5	5
0 1 1 0	ACK	6	SYN	&	6	F	V	f	v	6	6	6	6	6	6
0 1 1 1	BEL	7	ETB	'	7	G	W	g	w	7	7	7	7	7	7
1 0 0 0	BS	8	CAN	(8	H	X	h	x	8	8	8	8	8	8
1 0 0 1	HT	9	EM)	9	I	Y	i	y	9	9	9	9	9	9
1 0 1 0	LF	10	SUB	*	10	J	Z	j	z	10	10	10	10	10	10
1 0 1 1	VT	11	ESC	+	11	K	[k	{	11	11	11	11	11	11
1 1 0 0	FF	12	FS	,	12	L	\	l		12	12	12	12	12	12
1 1 0 1	CR	13	GS	-	13	M]	m	}	13	13	13	13	13	13
1 1 1 0	SO	14	RS	.	14	N	^	n	~	14	14	14	14	14	14
1 1 1 1	SI	15	US	/	15	O	_	o	DEL	15	15	15	15	15	15

LEGEND: CHAR

Order: 01/01/01
Dept. of Comp. Sci.
University of Tennessee
Knoxville TN 37906, USA

Norme trop spécifique à l'Américain.

Multiplicité des normes ISO-8859-x sur 8 bits; Norme UTF-8

Devinette

Quelle est la valeur de $(626F6E6A6F757221)_{16}$?

○○○○○○○
○○○○○○○○●

○○
○○
○○○○

○
○○
○○○○○○○○○○
○○○
○○○○○○○○
○○○○○○○○○○○○○○○○

Devinette

Quelle est la valeur de $(626F6E6A6F757221)_{16}$?

① “bonjour!” en ASCII

```

○○○○○○○
○○○○○○○○●

```

```

○○
○○
○○○○

```

```

○
○○
○○○○○○○○○○
○○○○
○○○○○○○○
○○○○○○○○○○○○○○○○

```

Devinette

Quelle est la valeur de $(626F6E6A6F757221)_{16}$?

- ❶ “bonjour!” en ASCII
- ❷ 7 093 009 341 547 377 185 , entier non signé sur 64 bits

```

○○○○○○○
○○○○○○○○●

```

```

○○
○○
○○○○

```


```

○
○○
○○○○○○○○○○
○○○
○○○○○○○○
○○○○○○○○○○○○○○○○

```

Devinette

Quelle est la valeur de $(626F6E6A6F757221)_{16}$?

- ❶ “bonjour!” en ASCII
- ❷ 7 093 009 341 547 377 185 , entier non signé sur 64 bits
- ❸  , un octet code un niveau de gris (0: noir, 255: blanc)

ooooooo
ooooooooo

oo
oo
oo
ooooo

o
oo
oooooooooooo
oooo
oooooooooo
oooooooooooooooooooo

La couche physique et la couche logique

```

○○○○○○○
○○○○○○○○○

```

```

●○
○○
○○○○

```

```

○
○○
○○○○○○○○○○○
○○○
○○○○○○○○○
○○○○○○○○○○○○○○○○○○

```

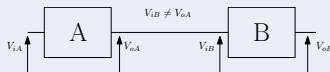
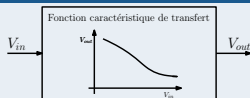
Représentation physique et manipulation d'un bit

Représentation

Un bit (0, 1) va être représenté physiquement par un niveau de tension

Par une valeur ? Par des domaines contigus ? Séparés par des marges ?

Manipulation



Des marges différentes en entrée et en sortie.

Il nous faut un composant avec une VTC non linéaire

Au début, le relais

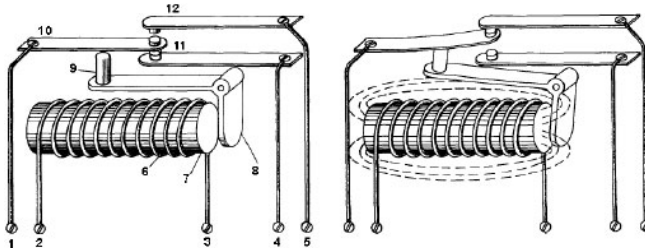


Figure: Relais électromécanique de J. Henry (1930-1940)

Puis vient le transistor (1947)

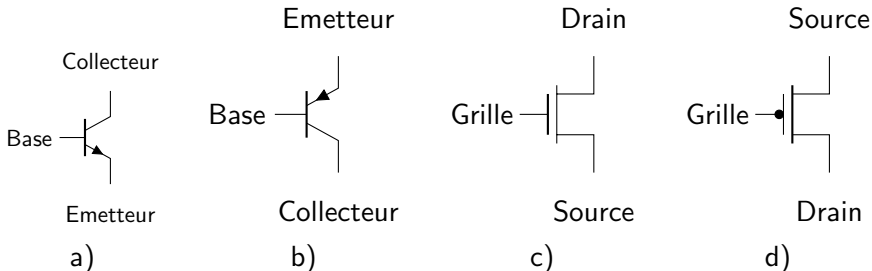
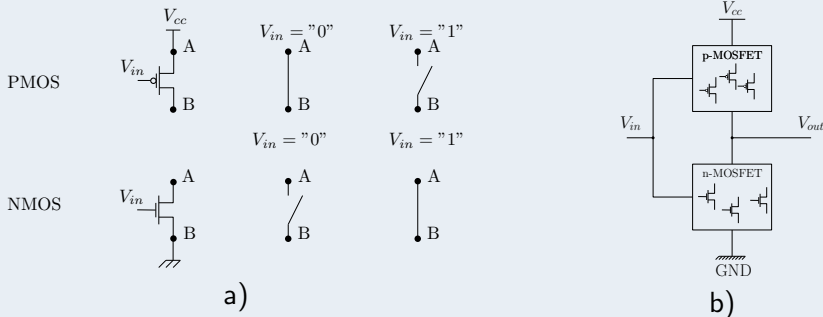


Figure: a) Transistor bipolaire NPN. b) Transistor bipolaire PNP. c) Transistor unipolaire NMOS. d) Transistor unipolaire PMOS

Un interrupteur commandable

TTL, CMOS et circuits



Notre premier circuit

Inverseur (*Not*) : $S = \overline{A}$

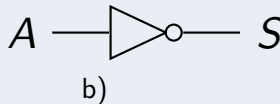
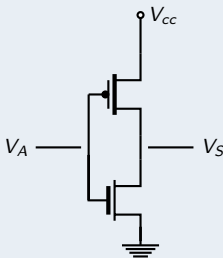


Figure: Une réalisation possible d'une porte NOT en technologie CMOS.
b) Symbole normalisé (norme Américaine).

Circuits à deux entrées

Porte NAND : $S = \overline{A \cdot B}$

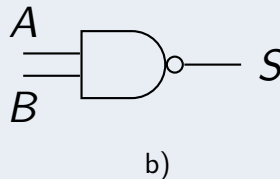
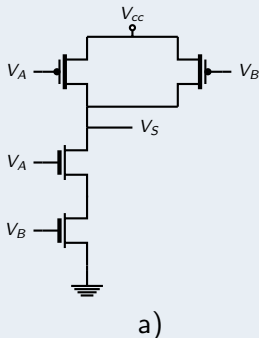


Figure: a) Une réalisation possible d'une porte NAND en technologie CMOS. b) Symbole normalisé

Circuits à deux entrées

Porte NOR : $S = \overline{A + B}$

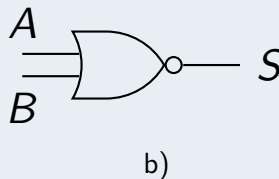
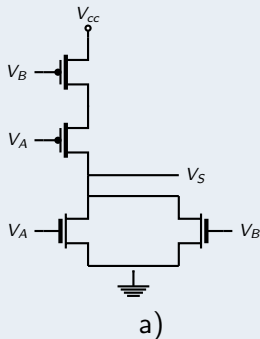


Figure: a) Une réalisation possible d'une porte NOR en technologie CMOS. b) Symbole normalisé.

oooooo
oooooooo

oo
oo
ooo●o

o
oo
oooooooooooo
oooo
oooooooooooo
oooooooooooooooooooo

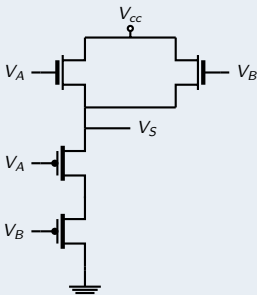
Circuits à deux entrées

On essaye une AND : $S = A.B$

oooooo
oooooooooooo
oo
ooo●oo
oo
oooooooooooo
oooo
oooooooooo
oooooooooooooooooooo

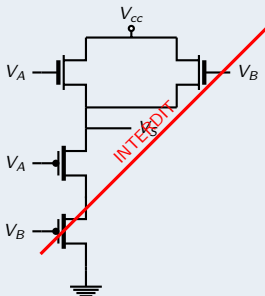
Circuits à deux entrées

On essaye une AND : $S = A.B$



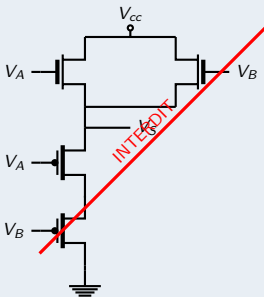
Circuits à deux entrées

On essaye une AND : $S = A.B$



Circuits à deux entrées

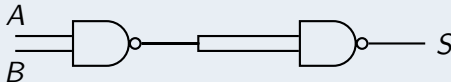
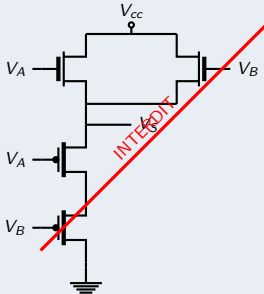
On essaye une AND : $S = A.B$



On fait comment ??

Circuits à deux entrées

On essaye une AND : $S = A.B$




```

ooooooo
ooooooooo

```

```

oo
oo
oo
oooo●

```

```

o
oo
oooooooooooo
oooo
oooooooooooo
oooooooooooo
oooooooooooooooooooo

```

Universalité de la porte NAND

Lois de DeMorgan

- $\overline{A.B} = \overline{A} + \overline{B}$
- $\overline{A + B} = \overline{A}. \overline{B}$

On peut construire les portes AND, OR, NOT à partir de NAND.
La porte NAND est dite **universelle**.

```

○○○○○○○
○○○○○○○○○

```

```

○○
○○
○○○○

```

```

●
○○
○○○○○○○○○○○
○○○○
○○○○○○○○○
○○○○○○○○○○○○○○○○○○

```

Portes logiques

Portes à 1 entrée

Nom	Table de vérité	Symbole	Notation						
Buffer	<table><tr><th>A</th><th>S</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	A	S	0	0	1	1	$A \longrightarrow \triangle \longrightarrow S$	$S = A$
A	S								
0	0								
1	1								
Inverseur	<table><tr><th>A</th><th>S</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	S	0	1	1	0	$A \longrightarrow \triangle \bigcirc \longrightarrow S$	$S = \overline{A}$
A	S								
0	1								
1	0								

Portes à 2 entrées

A	B	16 sorties possibles															
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Op :		0	$A.B$	$A > B$	A	$A < B$	B	$A \oplus B$	$A + B$	$\bar{A} + \bar{B}$	$A == B$	\bar{B}	$A \geq B$	\bar{A}	$B \geq A$	$\bar{A}.\bar{B}$	1

oooooo
oooooooo

oo
oo
oooo

o
●o
oooooooooo
oooo
oooooooooo
oooooooooooooooooooo

Synthèse de circuits logiques

Les outils

- Spécification fonctionnelle : table de vérité (e.g. $A + B$)
- Equation logique et simplification (Tableau de Karnaugh) (e.g. $A + B$)
- Disjonction de conjonctions
- peut être réalisé
 - exclusivement avec des NAND,
 - ou exclusivement avec des NOR
 - exclusivement avec des NOT, AND, OR

oooooo
oooooooo

oo
oo
oooo

o
o
o●
oooooooooo
oooo
oooooooo
oooooooooooooooo

Circuits à $n > 2$ entrées

Cascade/Arbre et temps de propagation

Exemple d'une porte ET à 4 entrées.

oooooo
oooooooo

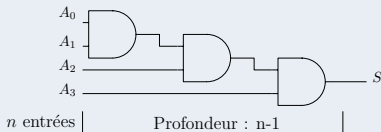
oo
oo
oooo

o
o●
oooooooooooo
oooo
oooooooooooo
oooooooooooooooooooo

Circuits à $n > 2$ entrées

Cascade/Arbre et temps de propagation

Exemple d'une porte ET à 4 entrées.



```

ooooooo
ooooooooo

```

```

oo
oo
oo
oooo

```

```

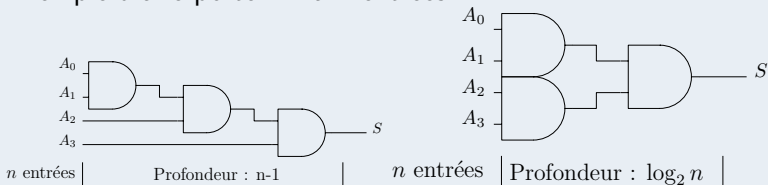
o
o
o●
oooooooooooo
oooo
oooooooooooo
oooooooooooooooooooo

```

Circuits à $n > 2$ entrées

Cascade/Arbre et temps de propagation

Exemple d'une porte ET à 4 entrées.



```

ooooooo
ooooooooo

```

```

oo
oo
oooo

```

```

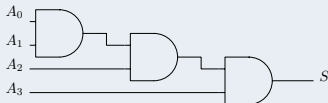
o
o
o●
oooooooooooo
oooo
oooooooooooo
oooooooooooo

```

Circuits à $n > 2$ entrées

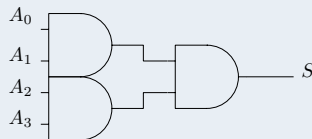
Cascade/Arbre et temps de propagation

Exemple d'une porte ET à 4 entrées.



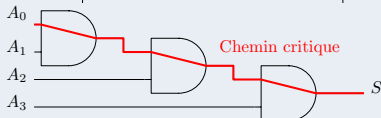
n entrées |

Profondeur : $n-1$



n entrées |

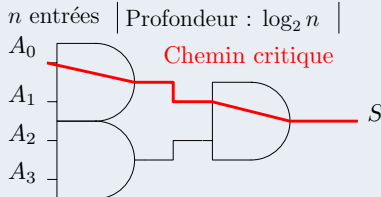
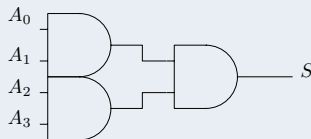
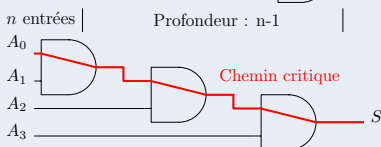
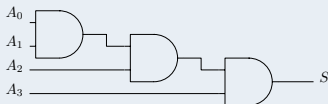
Profondeur : $\log_2 n$



Circuits à $n > 2$ entrées

Cascade/Arbre et temps de propagation

Exemple d'une porte ET à 4 entrées.



oooooo
oooooooo

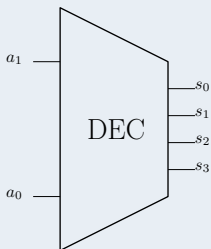
oo
oo
oooo

o
oo
●ooooooooo
oooo
oooooooooo
oooooooooooooooo

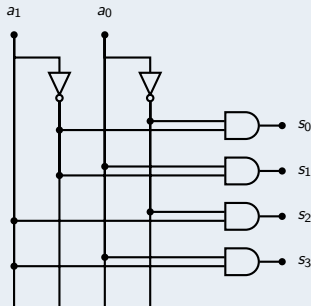
Circuits de logique combinatoire

Décodeur

⇒ Spécification fonctionnelle



a)

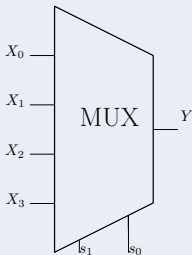


b)

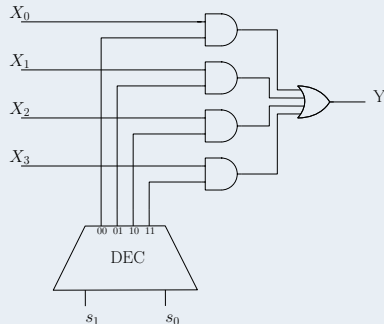
Circuits de logique combinatoire

Multiplexeur

⇒ Spécification fonctionnelle



a)

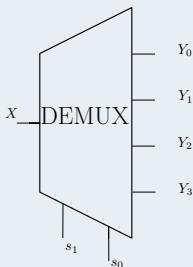


b)

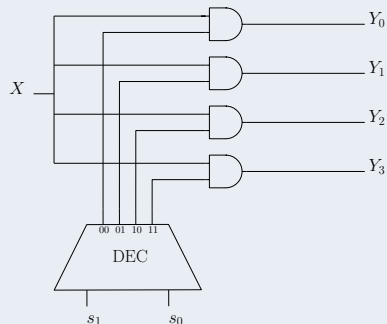
Circuits de logique combinatoire

Démultiplexeur

⇒ Spécification fonctionnelle

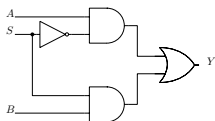


a)

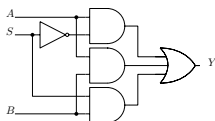
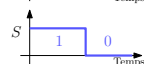
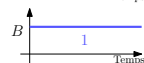
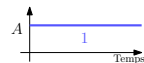


b)

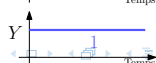
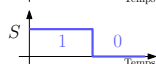
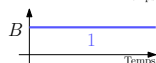
Aléa statique



A \ B	0	0	1	1
	0	1	1	0
0	0	0	1	1
1	0	1	1	0



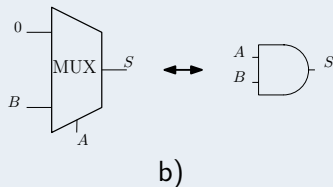
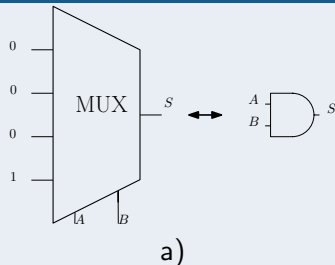
A \ B	0	0	1	1
	0	1	1	0
0	0	0	1	1
1	0	1	1	0



○○○○○○○
○○○○○○○○○○○○
○○
○○○○○
○○
○○○○●○○○○○
○○○○
○○○○○○○○
○○○○○○○○○○○○○○○○

Universalité du multiplexeur et ROM

Universalité du multiplexeur



⇒ Read Only Memory (ROM)

```

○○○○○○○
○○○○○○○○○

```

```

○○
○○
○○○○

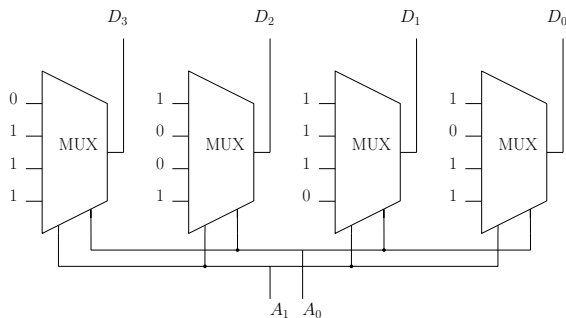
```

```

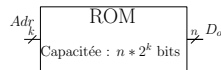
○
○○
○○
○○○○○●○○○○○
○○○
○○○○○○○○○
○○○○○○○○○○○○○○○○○○○○

```

Mémoire en lecture seule (Read Only Memory - ROM)



Adresses	Données
0x0	0111
0x1	1010
0x2	1011
0x3	1101



```

ooooooo
ooooooooo

```

```

oo
oo
oooo

```

```

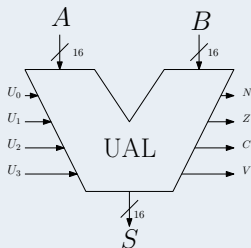
o
oo
ooooooo●oooo
oooo
ooooooooo
oooooooooooooooooooo

```

Unité arithmétique et logique (UAL)

Cahier des charges

- composant logique avec :
 - 2 entrées A, B sur n bits
 - 1 sortie S sur n bits
 - des bits de sélection d'opération, e.g. $2^4 = 16$ op: $U_3 U_2 U_1 U_0$



```

ooooooo
ooooooooo

```

```

oo
oo
oo
oooo

```

```

o
oo
oo
oooooooo●ooo
oooo
ooooooooo
oooooooooooooooooooo

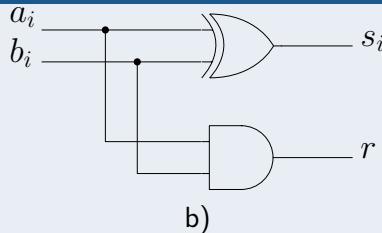
```

Unité arithmétique et logique (UAL) - Additionneur

Demi-additionneur 1 bit

a_i	b_i	s_i	r
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

a)




```

oooooo
oooooooo

```

```

oo
oo
oo
oooo

```

```

o
oo
ooooooooo●oo
oooo
ooooooooo
ooooooooooooo

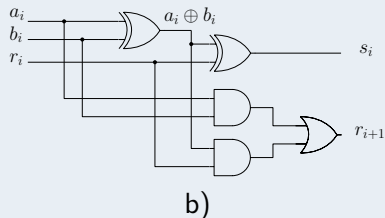
```

Unité arithmétique et logique (UAL) - Additionneur

Additionneur 1 bit

a_i	b_i	r_i	s_i	r_{i+1}
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

a)



```

○○○○○○○
○○○○○○○○○

```

```

○○
○○
○○○○

```

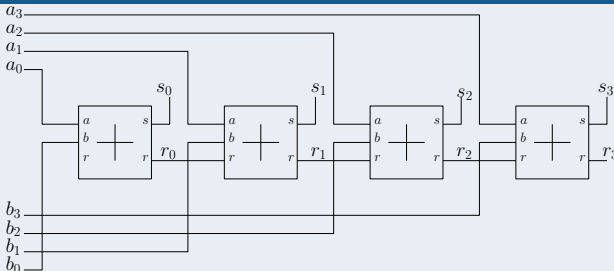
```

○
○○
○○○○○○○○○●○○
○○○○
○○○○○○○○
○○○○○○○○○○○○○○○○○○

```

Unité arithmétique et logique (UAL) - Additionneur

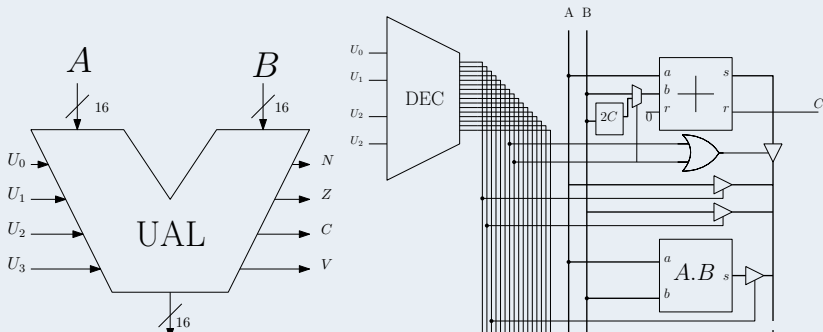
Additionneur n bits



Unité arithmétique et logique (UAL)

Synthèse de l'UAL

- on réalise chacun des circuits opératoires
- on les combine et sélectionne grâce à un décodeur



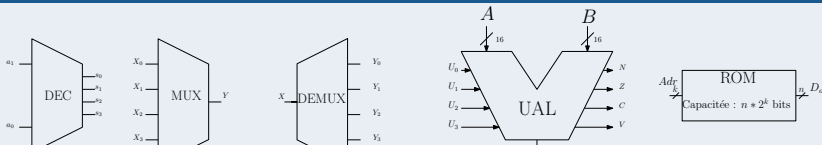
Résumons

Logique combinatoire

Pour réaliser un circuit de logique combinatoire

- spécification fonctionnelle : table de vérité, équation logique
- simplification de l'équation (e.g. tableaux de karnaugh)
- circuit avec les portes logiques ET, OU, NOT (ou juste NAND, NOR, MUX)

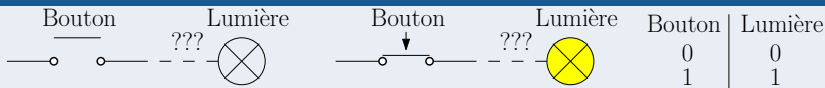
Par exemple



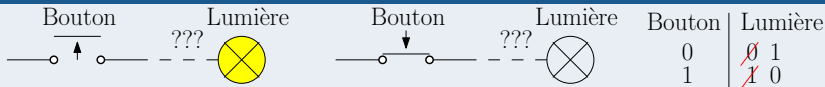
C'est tout ?

Problème : un bouton (0/1) - une lumière (0/1)

Allumons



Eteignons

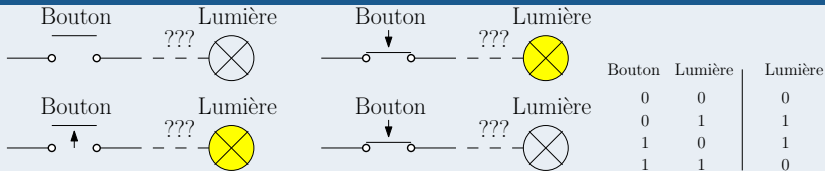


⇒ mince !?!?

C'est tout ?

Problème : un bouton (0/1) - une lumière (0/1)

Solution



Le **nouvel état** lumière dépends de l'**entrée bouton** et de l'**ancien état** lumière

oooooo
oooooooo

oo
oo
oooo

o
oo
oooooooooooo
ooo●
oooooooo
oooooooooooooooooooo

Plus généralement

On reboucle simplement la sortie sur l'entrée ?

Ce n'est pas forcément la sortie qui est utilisée en entrée puisque je peux très bien avoir besoin de produire la même sortie dans des états différents

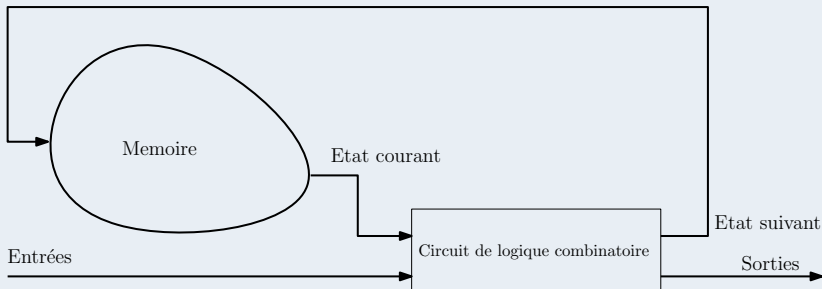
oooooo
oooooooo

oo
oo
oooo

o
oo
oooooooooooo
oooo
●oooooooo
oooooooooooooooo

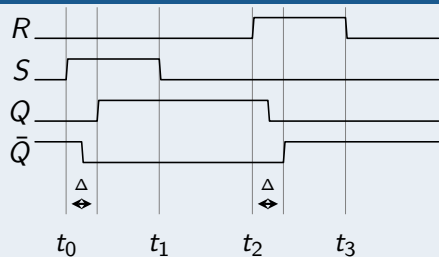
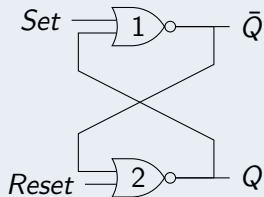
Logique séquentielle

Transducteur fini



Verrou Reset-Set (RS)

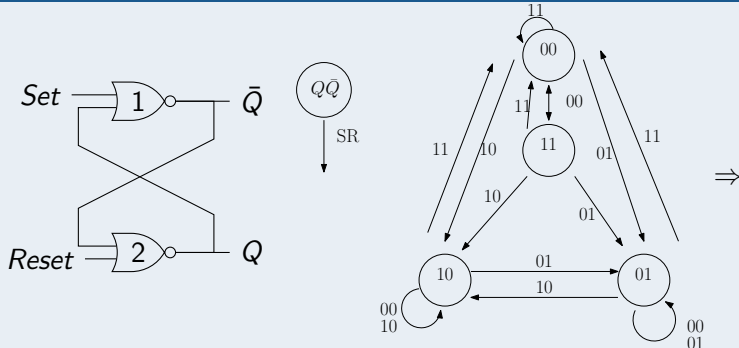
Schéma et Chronogramme



⇒ effet mémoire si $R=S=0$ ($t_1 \leq t \leq t_2$, $t \geq t_3$)

En fait ... c'est plus compliqué

Machine à états finis pour un circuit symétrique



logique synchrone et logique asynchrone

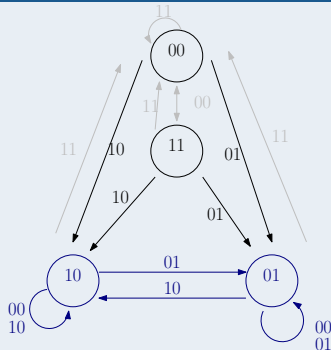
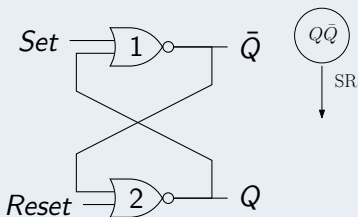
oooooo
oooooooo

oo
oo
oooo

o
oo
oooooooooooo
ooo
ooo
ooo●oooo
oooooooooooooooo

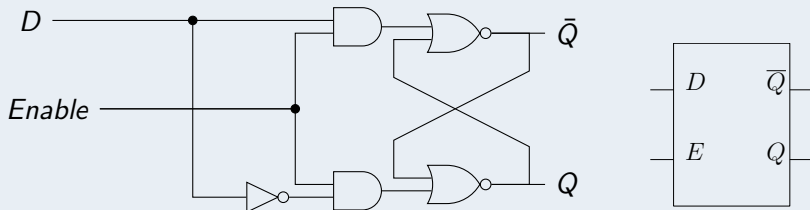
Supprimer les états instables

Et si $R.S = 0$?



Mémoriser un bit sur niveau haut : Verrou D

Schéma et chronogramme



- $Set = D.Enable$
- $Reset = \bar{D}.Enable$
- $Q(t) = Enable.D + \overline{Enable}.Q(t-1)$

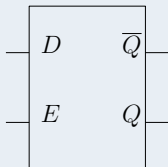
oooooo
oooooooo

oo
oo
oooo

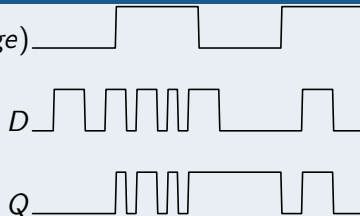
o
oo
oooooooooooo
oooo
oooo●ooo
oooooooooooooooo

Mémoriser un bit sur niveau haut : Verrou D

Schéma et chronogramme



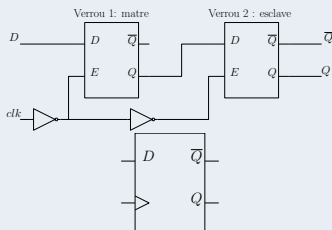
Enable(Horloge)



- $E = 1 \Rightarrow Q = D$: le verrou est transparent,
- $E = 0 \Rightarrow$ le verrou est dans un état mémoire, sa sortie ne change pas même si D change

Mémoire sur front : Bascule D synchrone sur front montant

Bascule D : maître-esclave; Schéma et chronogramme



Horloge

$D = D_1$

E_1

$Q_1 = D_2$

E_2

$Q = Q_2$

$Q = D$ au front montant d'horloge

```

ooooooo
ooooooooo

```

```

oo
oo
oooo

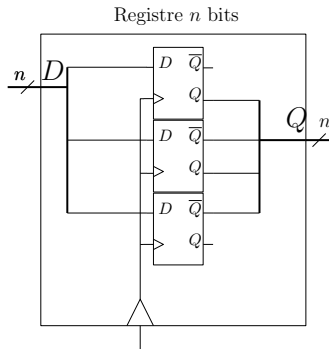
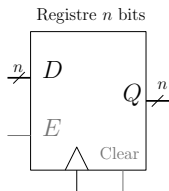
```

```

o
oo
oooooooooooo
oooo
ooooooooo●o
oooooooooooooooooooo

```

Registre à n bits

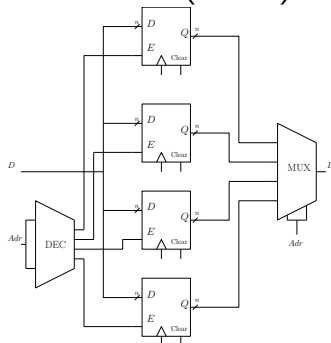
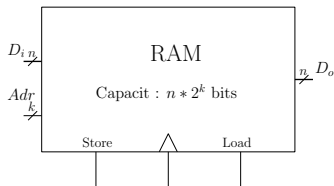


○○○○○○○
○○○○○○○○○

○○
○○
○○○○○

○
○○
○○○○○○○○○○○
○○○○
○○○○○○○○●
○○○○○○○○○○○○○○○

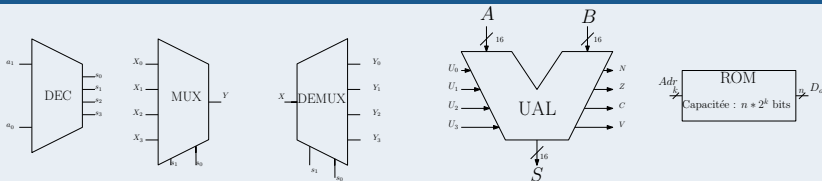
Mémoire en lecture/écriture à accès aléatoire (RAM)



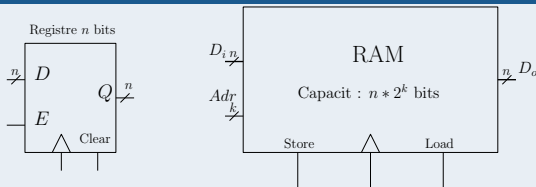
- Lecture : on place Adr et $Load=1 \Rightarrow D_o = RAM[Adr]$
- Écriture : on place Adr , D_i et $Store=1$ + **front d'horloge** $\Rightarrow RAM[Adr] = D_i$

Résumons

Logique combinatoire



Logique séquentielle



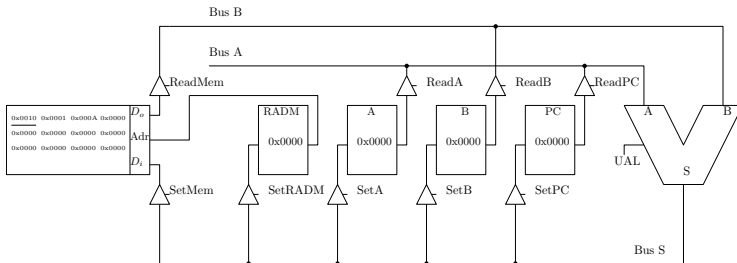
ooooooo
ooooooooo

oo
oo
oooo

o
oo
ooooooooooo
oooo
ooooooooo
o●oooooooooooooooo

Notre premier chemin de données

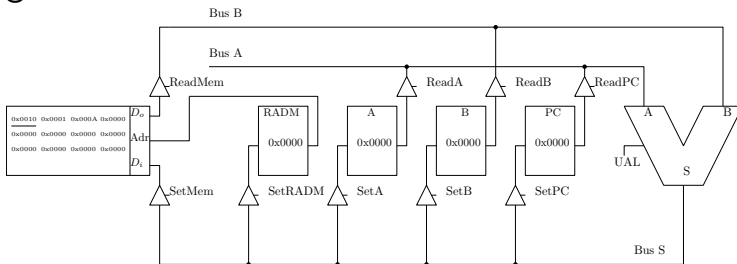
Notre premier chemin de données



Spécifications

- données et adresses sur 16 bits; RAM : 2^{16} mots de 16 bits = 128 ko
- des registres génériques : A, B ; des registres particuliers PC et RADM
- RAM pour stocker (pour le moment) les données
- des signaux de contrôle : Read<A,B,PC,Mem>, Set<A,B,PC,RADM,Mem>, UAL
- Architecture Load/Store : opérations avec des opérandes en registre

Les registres



- A, B : registres d'opérandes pour effectuer des opérations
- PC (*Program counter*, ou CO : compteur ordinal): index la position de la donnée en cours d'utilisation
- RADM : Registre d'Adresse Mémoire : quel mot est adressé en mémoire (\neq PC)

```

oooooo
oooooooo

```

```

oo
oo
oooo

```

```

o
oo
oooooooooooo
oooo
oooooooooooo
oooo●oooooooooooo

```

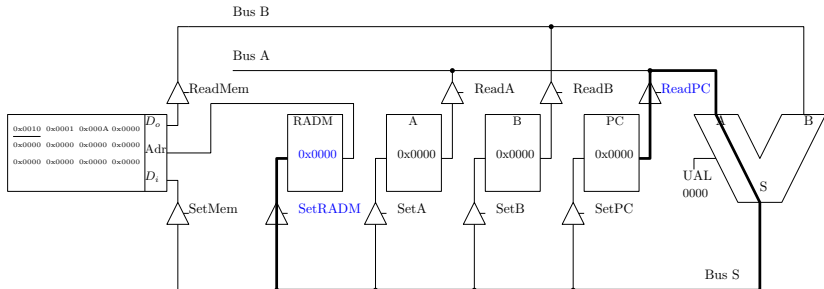
Exemple de séquençement manuel

Problème

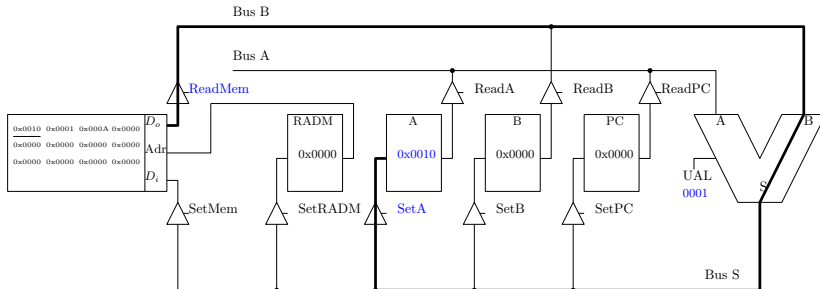
Additionner $16=0x0010$ et $1=0x0001$ et stocker le résultat à l'adresse $0x000A$

Adresses	Contenu			
0000	0010	0001	000A	0000
0004	0000	0000	0000	0000
0008	0000	0000	0000	0000

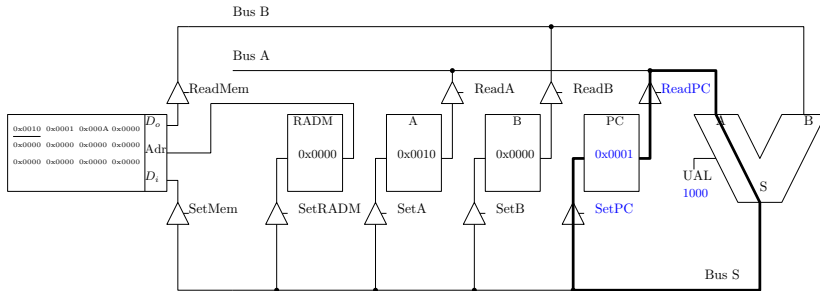
Chargement immédiat dans A (1/3)



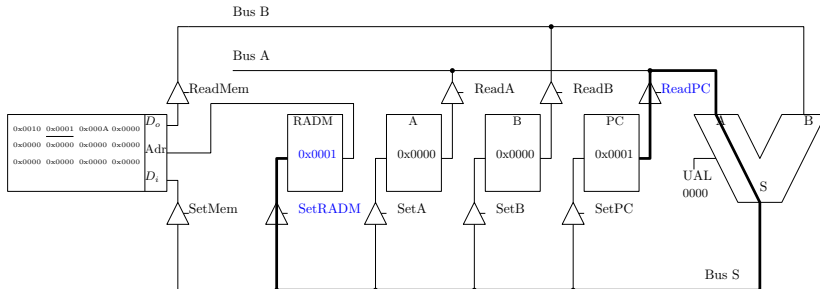
Chargement immédiat dans A (2/3)



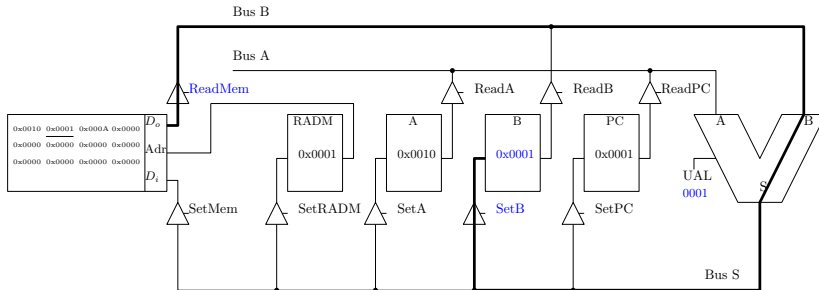
Chargement immédiat dans A (3/3)



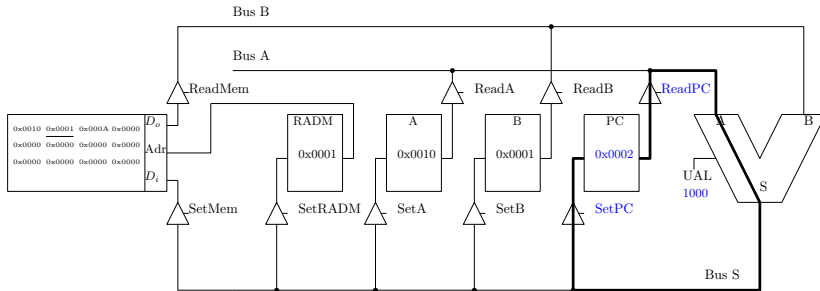
Chargement immédiat dans B (1/3)



Chargement immédiat dans B (2/3)



Chargement immédiat dans B (3/3)



```

○○○○○○○
○○○○○○○○○

```

```

○○
○○
○○○○

```

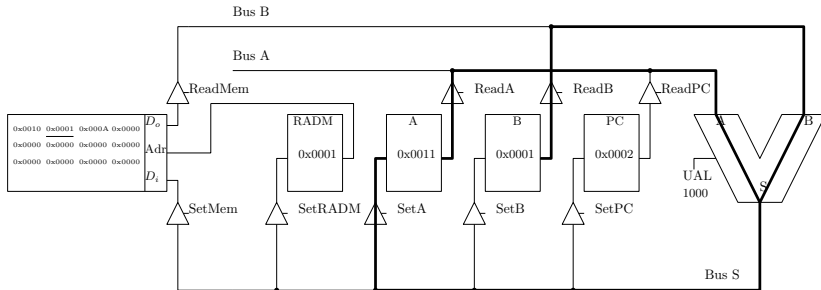
```

○
○○
○○○○○○○○○○○
○○○○
○○○○○○○○
○○○○○○○○○○●○○○○○

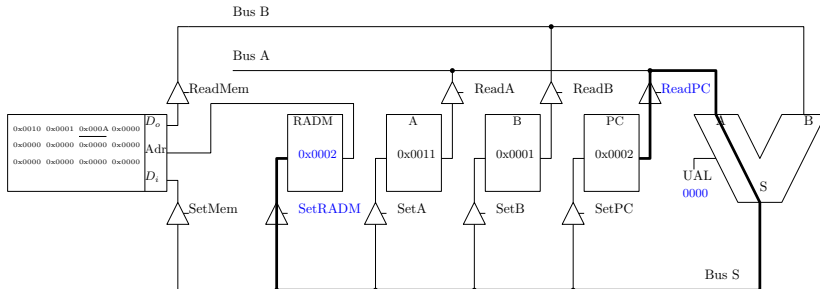
```

Synthèse : notre premier chemin de données

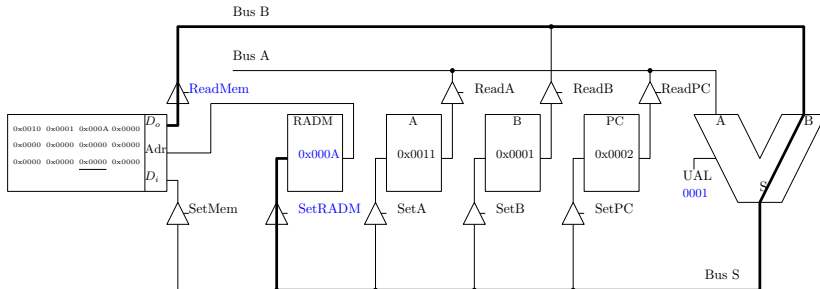
Addition : $A := A + B$



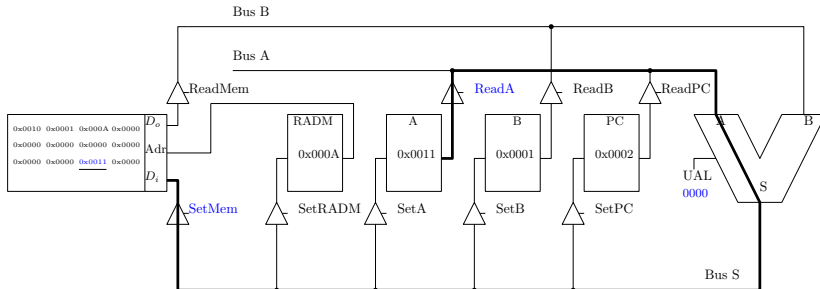
Sauvegarde du contenu du registre A en mémoire (1/4)



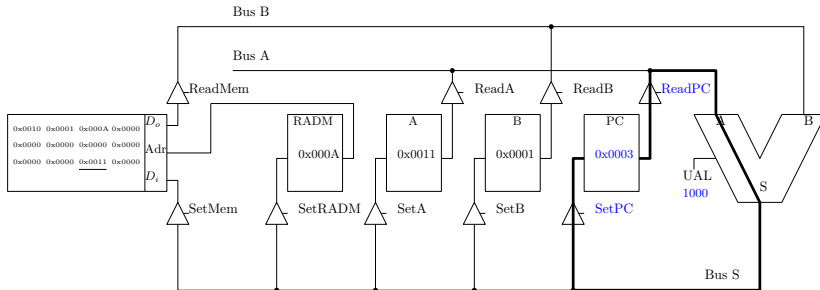
Sauvegarde du contenu du registre A en mémoire (2/4)



Sauvegarde du contenu du registre A en mémoire (3/4)



Sauvegarde du contenu du registre A en mémoire (4/4)



oooooo
oooooooooo

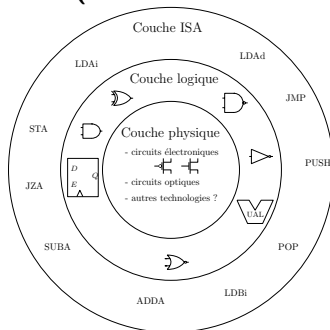
oo
oo
ooooo

o
oo
oooooooooooo
oooo
oooooooooo
ooooooooooooooo●

Chargement et Mode d'adressage immédiat / direct

- Adressage immédiat : le **mot mémoire est la valeur** à charger
- Adressage direct : le **mot mémoire est l'adresse** en mémoire de la valeur à charger, e.g. incrémenter un compteur dont la valeur courante est stockée à une adresse donnée en mémoire
- Adressage indirect : le mot mémoire est l'adresse à laquelle trouver l'adresse de la valeur à charger
- Adressage relatif : le mot mémoire contient l'adresse et un décalage, e.g. accéder aux éléments d'un tableau $A[i]$

La couche ISA (Instruction Set Architecture)



```

0000000
000000000

```

```

00
00
00000

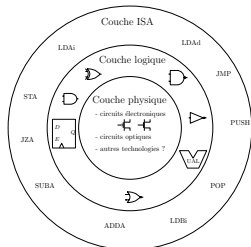
```

```

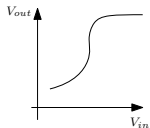
0
00
000000000000
0000
000000000
000000000000000000

```

Approche par couches d'abstractions successives



Couche physique



Couche logique

A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

Couche ISA

Registres : A, B, PC

```

LDA : 0x1000
STA : 0x1c00
JMP : 0x7000

```

Couche d'assemblage

```

main: LDAi 0x0003
      STA 0x1000
      JMP 0x0020

```

Langage de haut niveau

```

Python
x = 3
print(x)
for i in range(3):
    x = x - 1

```

```

C
int x = 3;
printf("%i", x);
while(x != 0) {
    x = x - 1;
}

```

```

ooooooo
ooooooooo

```

```

oo
oo
ooooo

```

```

o
oo
oooooooooooo
oooo
oooooooooo
oooooooooooooooooooo

```

En plus des données, définissons le programme

Architecture de von Neumann : mémoire (data+prog) ↔ processeur

Les instructions

Adresses	Contenu			
0000	0010	0001	000A	0000
0004	0000	0000	0000	0000
0008	0000	0000	0000	0000

```

ooooooo
ooooooooo

```

```

oo
oo
ooooo

```

```

o
oo
oooooooooooo
oooo
oooooooooooo
oooooooooooooooooooo

```

En plus des données, définissons le programme

Architecture de von Neumann : mémoire (data+prog) ↔ processeur

Les instructions

Adresses	Contenu			
0000	LDAi	0010	LDBi	0001
0004	ADDA	STA	000A	0000
0008	0000	0000	0000	0000

```

0000000
0000000000

```

```

00
00
00000

```

```

0
00
00000000000
0000
000000000
000000000000000000

```

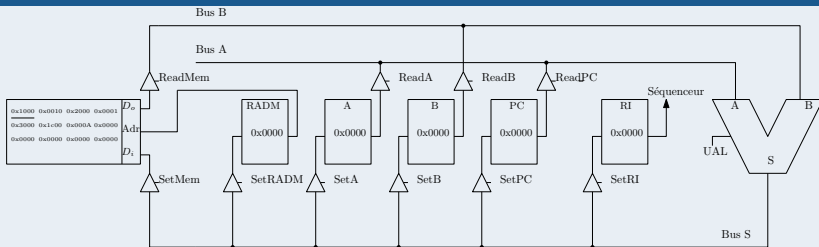
En plus des données, définissons le programme

Codage des instructions

Nom de l'instruction	Code de l'instruction	Adresses	Contenu			
LDAi	0×1000	0000	1000	0010	2000	0001
LDAd	0×1400		3000	1c00	000A	0000
LDBi	0×2000	0004	0000	0000	0000	0000
STA	$0 \times 1c00$	0008	0000	0000	0000	0000
ADDA	0×3000					

⇒ Programme en **langage machine**

Une première possibilité



- récupérée en mémoire
- placée dans le registre d'instruction (RI)
- qui est une entrée du séquenceur

oooooo
oooooooo

oo
oo
oooo

o
oo
oooooooooooo
oooo
oooooooooooo
oooooooooooooooooooo

Séquencement du chemin de données

Séquenceur

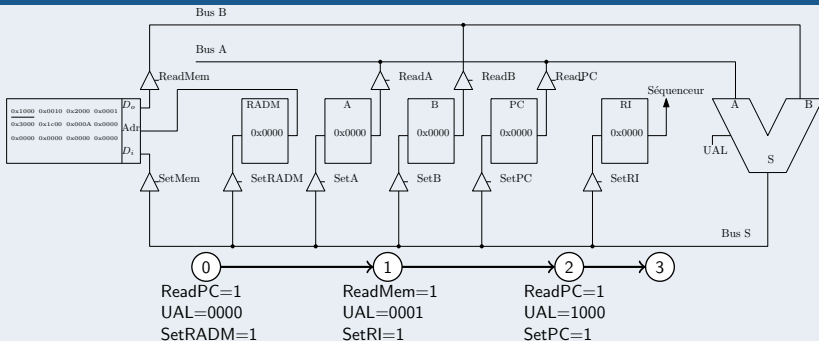
Séquence de signaux de contrôle spécifique :

- ❶ pour récupérer l'instruction
- ❷ en fonction de l'instruction

⇒ machine à états finis

Générer les signaux de contrôle : le séquenceur

Fetch



```

0000000
0000000000

```

```

00
00
00000

```

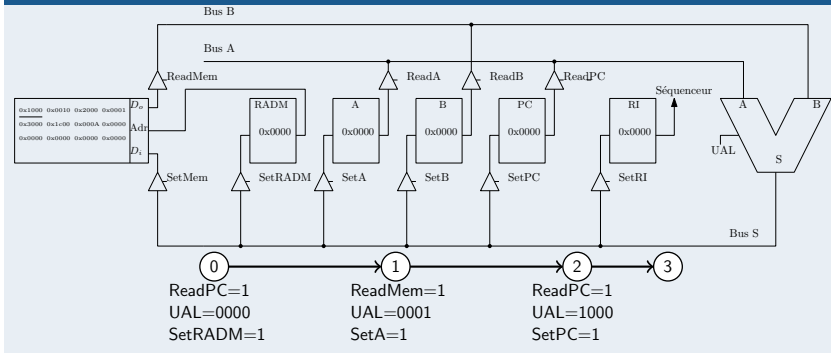
```

0
00
000000000000
0000
00000000
000000000000000000

```

Générer les signaux de contrôle : le séquenceur

Chargement immédiat dans A : LDAi



```

0000000
0000000000

```

```

00
00
00000

```

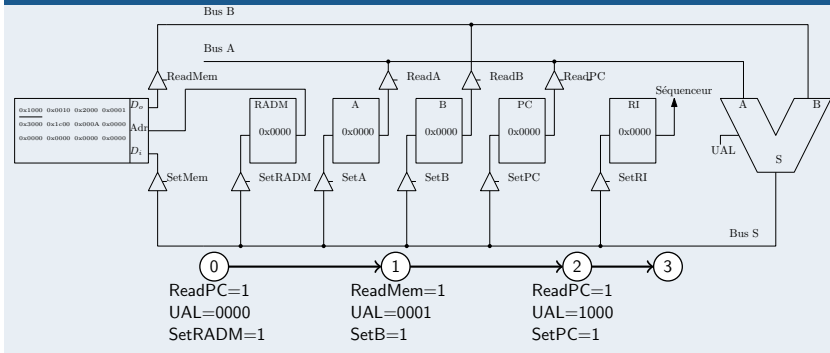
```

0
00
000000000000
0000
00000000
0000000000000000

```

Générer les signaux de contrôle : le séquenceur

Chargement immédiat dans B : LDBi



```

0000000
0000000000

```

```

00
00
00000

```

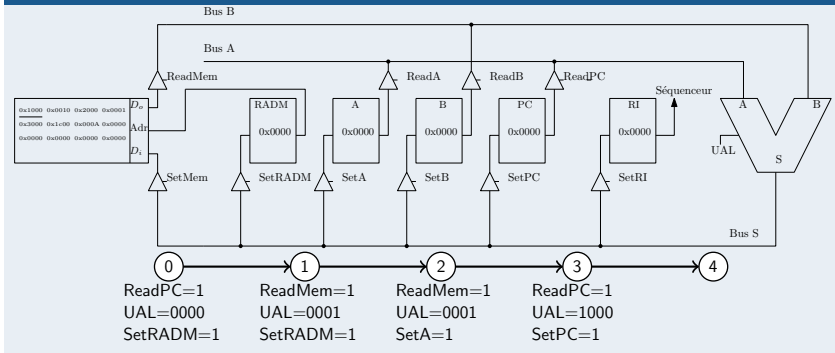
```

0
00
000000000000
0000
00000000
0000000000000000

```

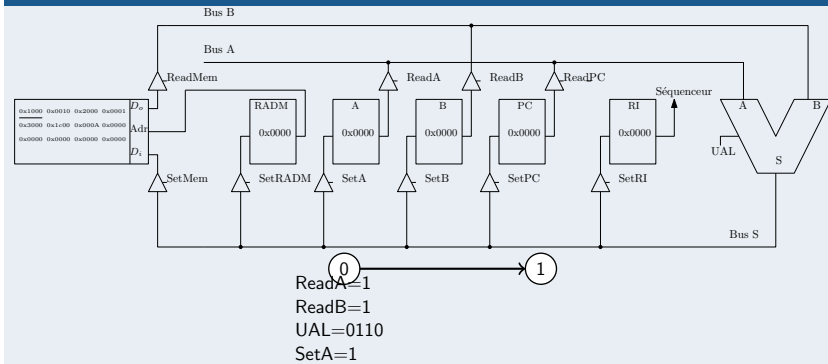
Générer les signaux de contrôle : le séquenceur

Chargement direct dans A : LDAd



Générer les signaux de contrôle : le séquenceur

Addition $A := A + B$: ADDA



```

00000000
0000000000

```

```

00
00
00000

```

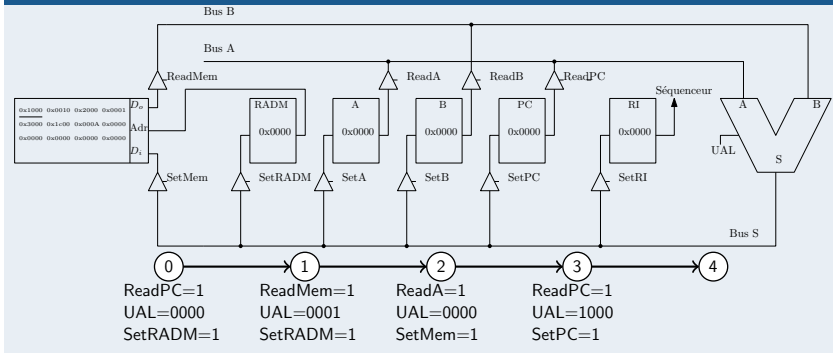
```

0
00
000000000000
0000
000000000
000000000000000000

```

Générer les signaux de contrôle : le séquenceur

Sauvegarde de A en mémoire : STA



```

00000000
0000000000

```

```

00
00
00000

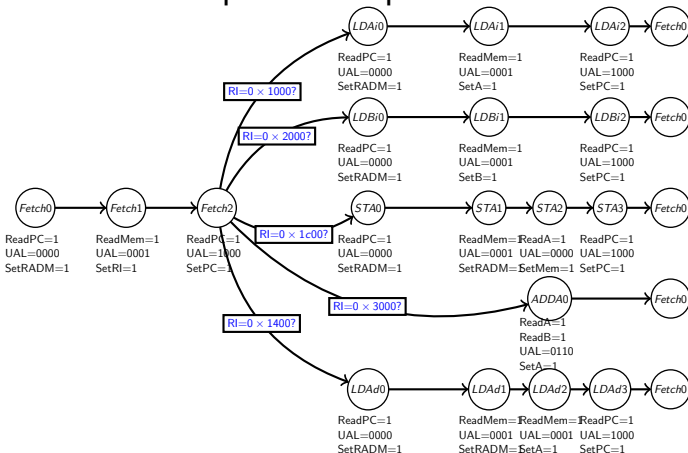
```

```

0
00
000000000000
0000
000000000
0000000000000000

```

Machine à états finis pour le séquenceur



Réalisation matérielle ? Soyons astucieux sur le codage des états

```

0000000
0000000000

```

```

00
00
00000

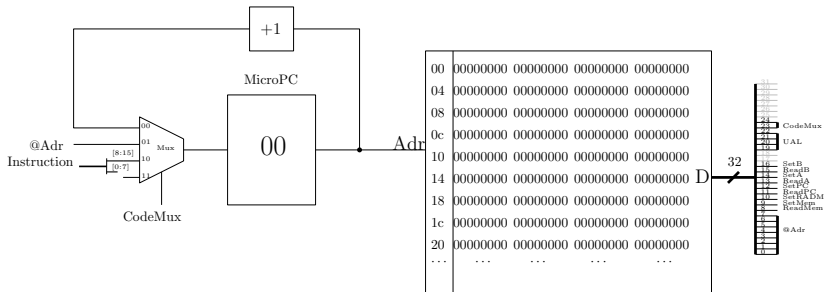
```

```

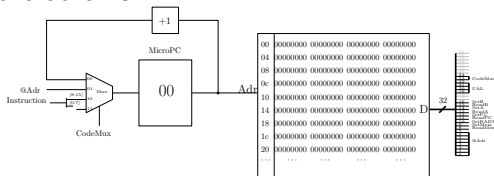
0
00
000000000000
0000
000000000
0000000000000000

```

Séquenceur micro-programmé



Les micro-instructions



- ROM[0x00] : saut à fetch
- ROM[0x08,0x09,0x0A,0x0B] : fetch/decode
- ROM[0x10,0x11,0x12,0x13] : LDAi
- ROM[0x14,0x15,0x16,0x17] : LDAd
- ROM[0x1c,0x1d,0x1e,0x1f] : LDAd
- ROM[0x20,0x21,0x22,0x23] : LDBi
- ROM[0x30,0x31,0x32,0x33] : ADDA

```

ooooooo
ooooooooo

```

```

oo
oo
oooo

```

```

o
oo
oooooooooooo
oooo
oooooooooooo
oooooooooooooooooooo

```

Les branchements

$$fact(n) = \begin{cases} \text{si } n = 0 \text{ alors} & 1 \\ \text{sinon} & n * fact(n - 1) \end{cases}$$

si ... alors ... sinon ? Indicateurs de l'UAL

Instructions de branchement

- JMP (0×7000)

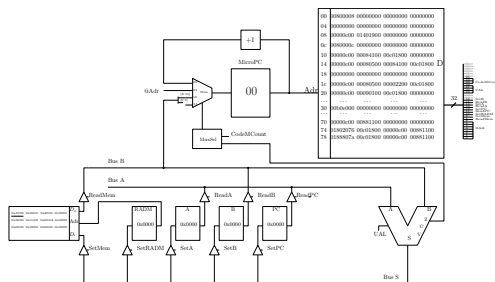
$$\text{JMP } op \Leftrightarrow PC := op$$

- JZA (0×7400)

$$\text{JZA } op \Leftrightarrow \begin{cases} PC := op & \text{si } A == 0 \\ PC := PC + 1 & \text{sinon} \end{cases}$$

◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡

Architecture avec branchement



CodeMCount	Z	$S_1 S_0$	Sémantique
000	0 ou 1	00	MicroPC := MicroPC+1
001	0 ou 1	01	MicroPC := @Adr
010	0 ou 1	10	MicroPC := Instruction
011	0	00	MicroPC := MicroPC+1 si la sortie de l'UAL $\neq 0$
011	1	01	MicroPC := @Adr si la sortie de l'UAL == 0