IFT-6800, Automne 2016

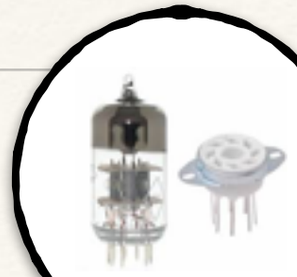
Cours #2–Architecture des ordinateurs contemporains

Louis Salvail
André-Aisenstadt, #3369
salvail@iro.umontreal.ca

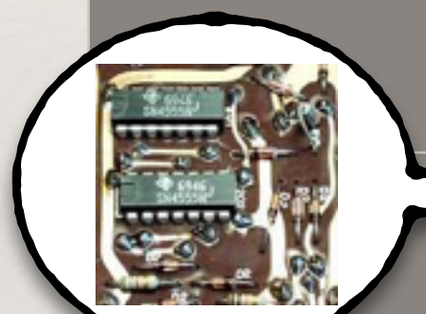
Plan

- Historique
- Architecture, le modèle de Von Neumann
- Codage et opérations de base
- Introduction à la programmation en langage machine:
    - L'assembleur du microprocesseur 6502 (rapidement),
    - Le langage de la machine X-TOY.

Historique I



Génération	Période approximative	Technologie	Vitesse opérations/sec
1	1946-1957	tubes à vide	40 000
2	1958-1964	transistors	200 000
3	1965-1971	SSI/MSI	1 000 000
4	1972-1977	LSI	10 000 000
5	1978-	VLSI	100 000 000

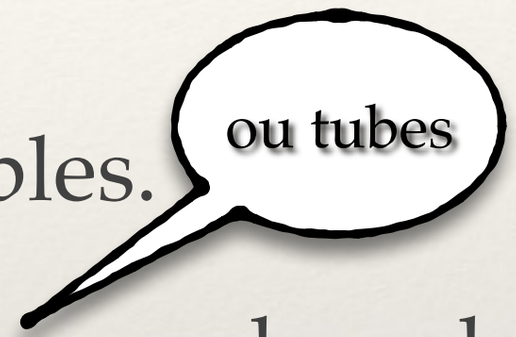


SSI/MSI: Small and Medium Scale Integration (100 composantes)
LSI: Large Scale Integration (500-300000 composantes)
VLSI: Very Large Scale Integration (300000+composantes)

VVLSI: Very Vary Large Scale Integration (1500000+composantes)

Historique II (1946-1957)

- Il s'agit de machines dédiées qui existaient en exemplaires uniques.
- Machines volumineuses et peu fiables.
- Utilisent comme technologie les lampes, les relais, les résistances.
- 10^4 éléments logiques.
- Programmation par cartes perforées.
- La programmation est la plupart du temps limitées aux applications pour lesquelles la machine a été conçue.



Historique III (1958-1964)

- Usage général, machine fiable.
- Technologies à transistors.
- 10^5 éléments logiques.
- Ces machines pouvaient être programmées comme celles d'aujourd'hui.
- Apparition des premiers langages de programmation évolués:
  - FORTRAN (1954)
  - LISP (1958)
  - ALGOL (1958)
  - COBOL (1959)

Historique IV (1965-1971)

- Technologie des circuits intégrés (SSI/MSI).
- 10^6 éléments logiques.
- Avènement du système d'exploitation complexe, des mini-ordinateurs:
  - *Digital Equipment Corporation*, le *PDP-11*, avec le système d'exploitation *TOPS-10*(1967). Populaire dans les universités et sur l'*ARPANET*(1969) entre UCLA et Stanford. Produira les OS *RT-11*, *RSTS*, *RSX-11* et *VMS* permettants la programmation multi-usagers.
  - Les ordinateurs *Sigma(1966)* de *Scientific Data Systems/Xerox Data Systems* avaient des OS qui permettaient le partage du temps de calcul: *BCM*, *BPM*, *BTM*, *UTS* (Universal Time-Sharing System), *CP-V*.
  - L'ordinateur *MODCOMP I* de *Modular Computer Systems, Inc*. et son OS MAX I. Utilisé pour contrôler des sondes spatiales de la NASA.
  - La gamme d'ordinateurs *IBM 360* (1965-1978). Ce sont des mainframes qui ont été très populaires. Conçus pour que les changements de machine soient transparents dans toute la gamme.
  - l'OS *UNIX*(1969), un système multitâche et multi-utilisateur développé sur le PDP-7 par Ken Thomson. Pour avoir un système portable, il fût réécrit dans un langage inventé pour la tâche. Il s'agit du langage C (qui prenait comme départ le langage B) .

Arithmétique de Boole

En arithmétique de Boole les variables sont des variables binaires: X=0 ou X=1

Un registre de *m* bits est composé de *m* valeurs booléennes. Un tel registre peut être vu encodant un entier naturel entre *0* et 2^m*-1*:

7 6 5 4 3 2 1 0

0 1 0 1 1 0 0 1 $=0*2^7+1*2^6+0*2^5+1*2^4+1*2^3+0*2^2+0*2^1+1*2^0=89$

\+

0 1 1 0 0 1 0 0 $=0*2^7+1*2^6+1*2^5+0*2^4+0*2^3+1*2^2+0*2^1+0*2^0=100$

---

1 0 1 1 1 1 0 1 $=1*2^7+0*2^6+1*2^5+1*2^4+1*2^3+1*2^2+0*2^1+1*2^0=189$

Les ordinateurs contemporains ne savent manipuler que les bits. La base de représentation des nombres dans un ordinateur est donc la base 2.

Les opérations ET, OU, NON et OUX

ET	0	1
0	0	0
1	0	1

OU	0	1
0	0	1
1	1	1

OUX	0	1
0	0	1
1	1	0

Toutes les fonctions arithmétiques des entiers naturels dans les entiers naturels peuvent être réalisées en applicant des ET,OU, NON et OUX sur des registres suffisamment grands.

NON	0	1
	1	0

Le ET, le OU et OUX entre deux registres de m bits est le ET, le OU et le OUX bit à bit. Le NON d'un registre de m bits est le NON de chacun de ses bits:

$$0\,1\,0\,1\,1\,0\,0\,1 \wedge 0\,1\,1\,0\,0\,1\,0\,0 = 0\,1\,0\,0\,0\,0\,0\,0$$

$$0\,1\,0\,1\,1\,0\,0\,1 \vee 0\,1\,1\,0\,0\,1\,0\,0 = 0\,1\,1\,1\,1\,1\,0\,1$$

$$0\,1\,0\,1\,1\,0\,0\,1 \oplus 0\,1\,1\,0\,0\,1\,0\,0 = 0\,0\,1\,1\,1\,1\,0\,1$$

$$\neg\, 0\,1\,0\,1\,1\,0\,0\,1 = 1\,0\,1\,0\,0\,1\,1\,0$$

Les entiers négatifs en binaire

- Nous pouvons représenter les entiers négatifs en binaire de *m* bits de trois façons *naturelles*:
  - (représentation signe+magnitude) Utiliser le bit en position *m-1* (la dernière position en commençant par la position 0) comme un bit de signe: 1 veut dire négatif tandis que 0 veut dire positif: $10000011=-3_{10}$ et $00000011=3_{10}$. Notez que $10000011+00000011=10000110=-6_{10}$! De plus, $00000000=10000000=0_{10}$. Un registre de *m* bits peut donc coder des nombres entre $-2^{m-1}+1$...$2^{m-1}-1$ positifs et négatifs. Le nombre de nombres différents représe

utilisée sur les premières machines: UNIVAC, PDP-1,...

  - (complément à 1) Un registre de *m* bits est complémenté bit à bit pour en changer le signe. Les nombres positifs sont $0..2^{m-1}-1$: 00000000...01111111. Les nombres négatifs sont $-2^{m-1}+1$...-1: 11111110..10000000. Comme pour la représentation signe+magnitude: 00000000=11111111. L'addition fonctionne lorsque nous l'effectuons bit à bit: $7_{10}-14_{10}$ = 00000111+NON(00001110)=00000111+11110001=11111000=-00000111=-7_{10}. S'il y a une retenue en sortie alors il faut l'additionner à la fin: $-1_{10}+2_{10}$=11111110+00000010=0000000+retenue=00000001=1_{10}.

Le complément à deux est la représentation la plus utilisée aujourd'hui, presque toujours!

  - (complément à 2) Cette façon de coder élimine le problème des deux zéros et permet d'additionner bit à bit comme en décimal. Le signe d'un registre est changé en complémentant ses bits et en additionnant 1 ensuite: 5_{10}=00000101 et -5_{10}=11111011: 00000101+11111011=00000000. Il est possible de coder sur *m* bits les entiers -2^{m-1}...$2^{m-1}-1$.

Addition de registres de *n* bits en complément à deux

Nous avons deux registres de 8 bits nommés R1 et R2 (les registres de 8 bits sont appelés *octets*):

- Supposons que le premier registre R1 contient la valeur 17 en complément à deux.
- Supposons que le second registre R2 contient la valeur -21 en complément à deux.

	7	6	5	4	3	2	1	0
R1:	0	0	0	1	0	0	0	1

- Maintenant, plaçons -21 dans R2 en trouvant d'abord la représentation de +21:

	0	0	0	1	0	1	0	1

  - Nous complémentons ensuite les bits:

	1	1	1	0	1	0	1	0

  - Finalement, nous additionnons 1 pour obtenir -21 en complément à deux sur 8 bits:

R2:	1	1	1	0	1	0	1	1

- Nous voulons maintenant additionner le contenu de R1 à celui de R2.

Addition de registres de *n* bits en complément à deux (II)

	7	6	5	4	3	2	1	0
						1	1	
R1:	0	0	0	1	0	0	0	1
+								
R2:	1	1	1	0	1	0	1	1
	1	1	1	1	1	1	0	0

Un 1 en dernière position indique un nombre négatif.

ATTENTION: Cette façon de déterminer la valeur négative d'un registre encodée en complément à deux fonctionne toujours sauf pour une seule valeur, puisqu'il y a une valeur négative de plus que les valeurs positives sur un registre de taille fixe. (*Laquelle?????*)

Pour trouver lequel, il suffit de complémenter les bits:

0	0	0	0	0	0	1	1

Et d'additionner 1:

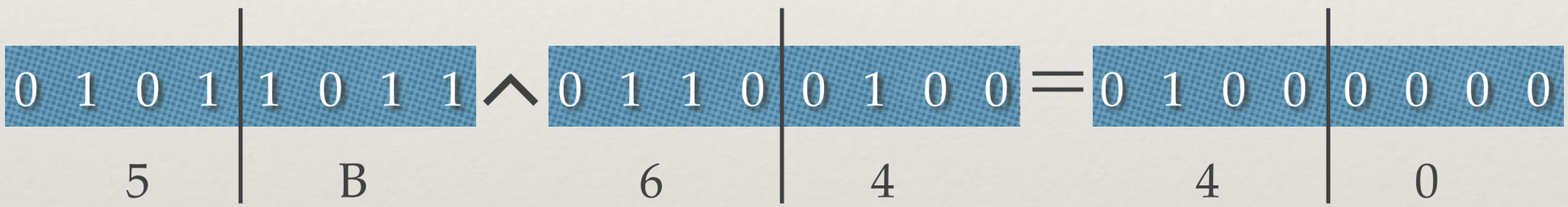
0	0	0	0	0	1	0	0

Et le nombre est "moins" la valeur décimale non-signée correspondante:

-4 = 17-21

L'hexadécimal

Les registres des ordinateurs sont habituellement de taille multiple de 8. Dans ce cas, la base hexadécimale permet de représenter la valeur des registres d'une façon plus compacte. On groupe les bits par 4. Chaque groupe de 4 bits peut prendre les 16 valeurs entre 0..15. La base hexadécimale est justement une base de 16 éléments: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. Exemple: $4A7_{16} = 7+10*16+4*16^2= 7+160+4*256=1191_{10}$.



L'addition maintenant:

$$\begin{array}{r} 5B \\ +\ 64 \\ \hline BF \end{array} \qquad \begin{array}{r} 01011011 \\ +\ 01100100 \\ \hline 10111111 \end{array}$$

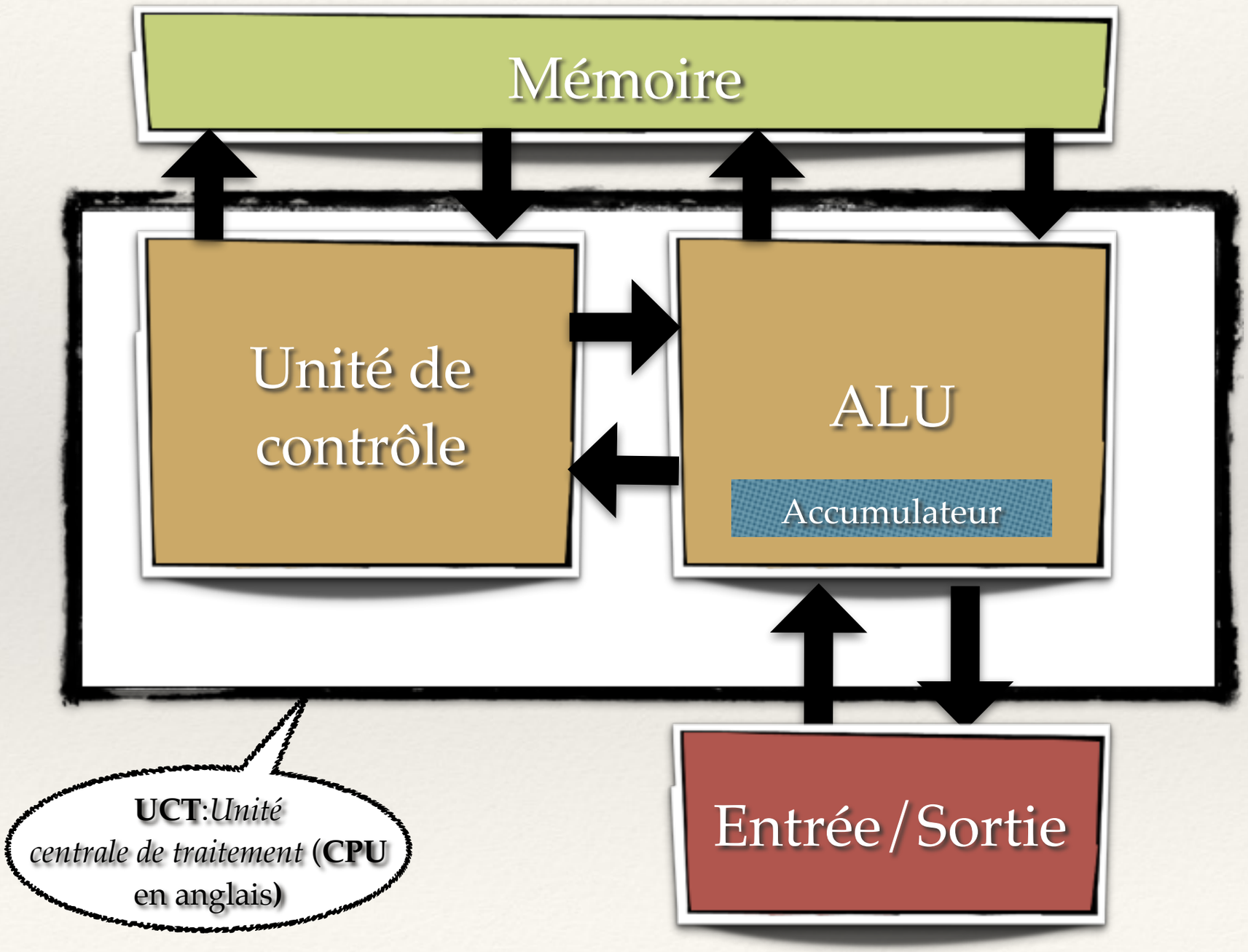
La machine de Von Neumann

- Une architecture qui fait référence au mathématicien John Von Neumann qui a décrit une machine dont le programme est stocké en mémoire. La machine a été développée à l'*Institute for Advanced Study* à Princeton. John W. Mauchly et John Eckert avait déjà utilisé un tel modèle pour leurs travaux sur la machine ENIAC. Cette architecture est appelée *architecture de Von Neumann*.

- L'architecture décrit l'ordinateur par 4 composantes:

1. L'unité arithmétique et logique (UAL ou ALU), son rôle est d'effectuer les opérations de base,
2. L'unité de contrôle/commande qui séquence les opérations de l'ALU,
3. La mémoire qui contient les données et les programmes. Les programmes indiquent à l'unité de contrôle les calculs à faire sur quelles données, et
4. Les dispositifs d'entrée-sortie qui permettent la communication avec le monde extérieur.

La machine de Von Neumann (II)

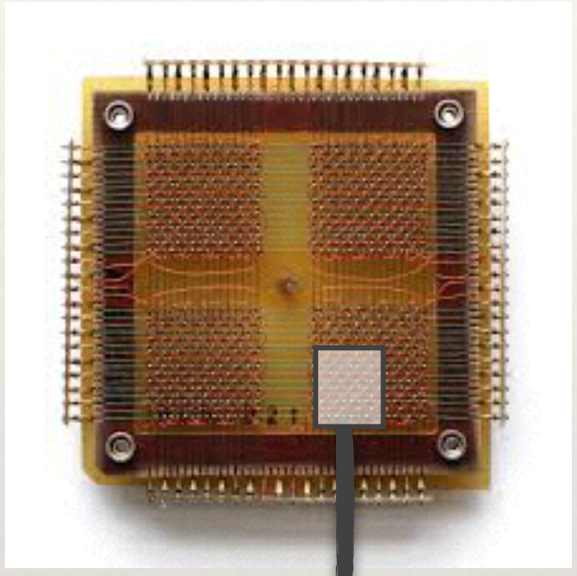


Mémoire

- L'élément de base de la mémoire est le bit (c.a.d. 0 ou 1). Les bits de mémoire sont rangés en paquets de m bits que l'on nomme mot.
  - Un mot de longueur 8 est appelé *octet*.
  - La longueur des mots est habituellement une puissance de 2: 2^k pour un entier k. De nos jours les mots sont habituellement de 32 ou 64 bits.
- La mémoire est une liste de mots. Le mot est la quantité de mémoire minimale et maximale qui peut être lue ou écrite en une seule opération de l'UC/ALU. Chaque mot de la mémoire est logé à une adresse unique.
- Il est possible d'aller chercher des mots en mémoire pour les traiter par l'UC/ALU.
- Il est possible de ranger en mémoire une valeur calculée par l'UC/ALU.
- Puisque les programmes sont en mémoire, ils peuvent être modifiés par l'UC/ALU pendant leur exécution.
- La mémoire décrite précédemment est appelée *mémoire vive (RAM)*. Il y a également la mémoire qui peut seulement être consultée (sans modification possible). Ce type de mémoire est appelé *mémoire morte (ROM)*.

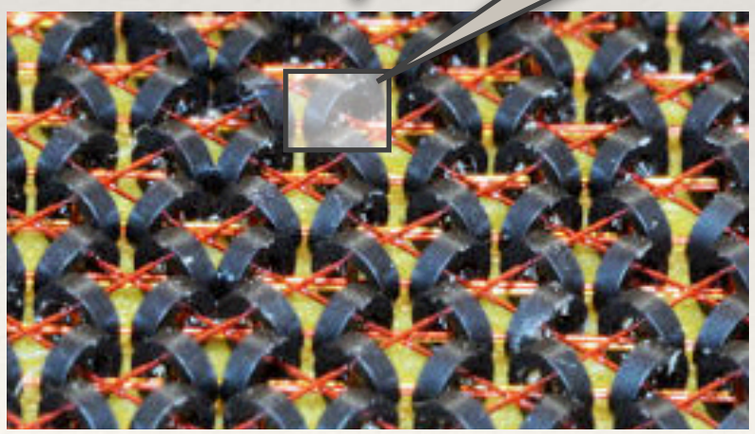
programmes immuables définis par les constructeurs

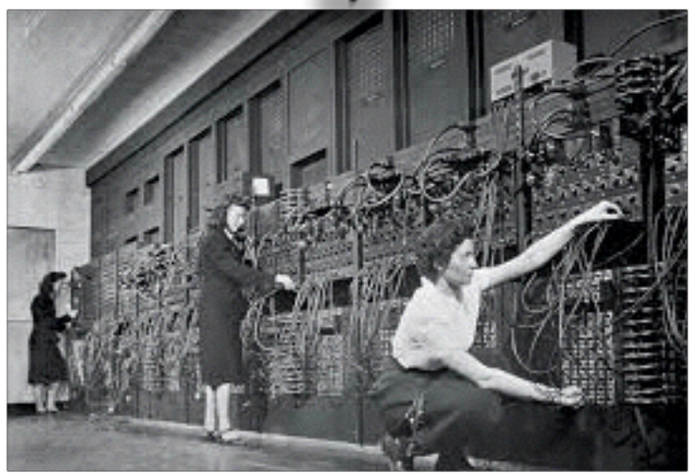
Mémoire (II)



Mémoire centrale du système de guidage du programme *Apollo* de la NASA, 1966–1975 (1024 bits=1Kbits).

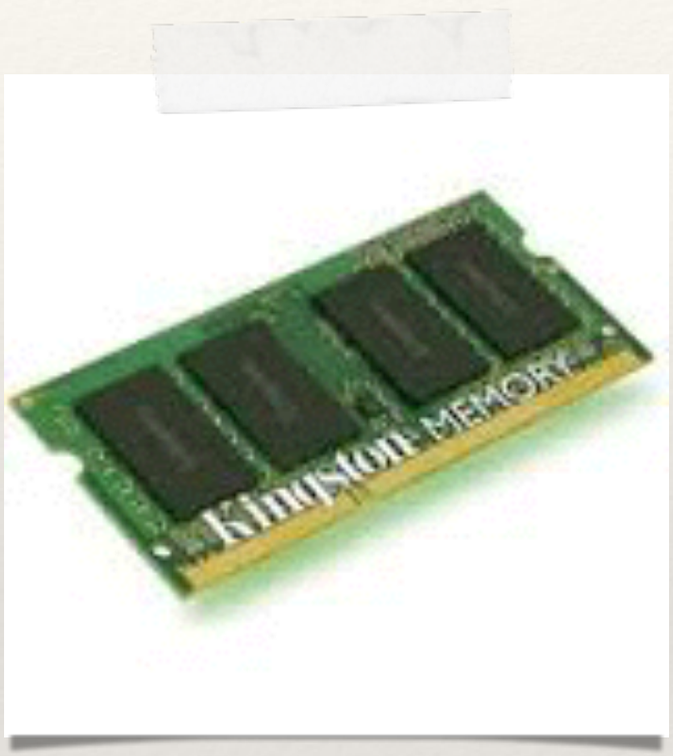
Un bit





Un bit d'ENIAC

Mémoire (III)



4 Gigaoctets de RAM = 32 Gigabits de RAM

10^3 bits

32 millions de fois plus

$32*10^9$ bits

L'information dans une mémoire

Adresses	Contenu binaire	Contenu hexadécimal
0:	01011000	58
1:	01110010	72
2:	01000000	40
3:	11111000	F8
4:	10000000	00
5:	11111111	FF
.	.	
.	.	
.	.	
255:	10101010	AA

Le mémoire de cet ordinateur est organisée en octets

$2^6=64_{10}$

Chaque octet de mémoire est logé à une adresse unique. (256 adresses ici)

La valeur décimale non signée de cet octet est: $2^7+2^6+2^5+2^4+2^3= 128_{10}+64_{10}+32_{10}+16_{10}+8_{10}= 248_{10}$, en complément à deux la valeur est -8_{10}, car complémenter et ajouter 1 donne 8_{10}.

La valeur décimale en complément à deux ne peut pas être obtenue par la méthode précédente. Ces la seule valeur qui ne peut pas être déterminée en complémentant et en ajoutant 1, car nous aurions toujours 10000000_2, qui est négatif!

Cette valeur décimale est -128_{10}

La valeur décimale en complément à deux d'un registre tout à 1 est toujours -1_{10}, car complémenter et ajouter 1 résultera en 1_{10}.

Architecture de Von Neumann:
Communication avec la mémoire(Cycles Saisir & Écrire)

MAR est un registre de taille suffisante pour adresser chaque mot en mémoire (8 bits ici).

MAR: Memory access register

00000000

bus d'adressage

Saisir

Écrire

MDR est un registre de taille suffisante pour ranger un mot (8 bits ici).

MDR: Memory data register

00000000

bus de données

Adresses	Contenu
0:	01011000
1:	01110010
2:	01000000
3:	11111000
4:	00000000
5:	11111111
.	.
.	.
.	.
255:	10101010

Caractéristiques de la mémoire moderne

- Le temps nécessaire pour accéder à un mot est de 700 nanosecondes ou moins.
- Elle peut stocker des milliers de GO (Giga-Octet):
  - 1 octet = 8 bits
  - 1 Kilo-octet = 1 KO = 1024 $\approx 10^3$ octets (1 page de roman)
  - 1 Mega-octet =1 MO = 1000 KO $\approx 10^6$ octets
  - 1 Giga-octet = 1 GO = 1000 MO $\approx 10^9$ octets
  - 1 Tera-octet = 1 TO = 1000 GO $\approx 10^{12}$ octets.
- La mémoire utilise la technologies des circuits intégrés.

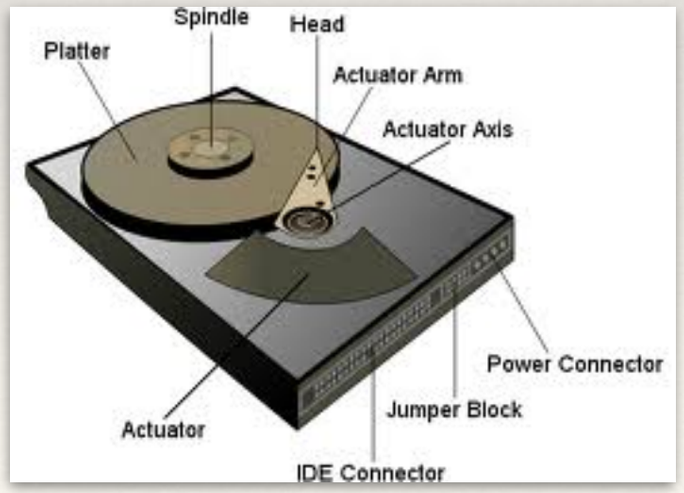
Amélioration des performances

- **Pagination de la mémoire:** est un schéma de gestion de mémoire par lequel l'ordinateur range et récupère des données dans la zone de mémoire secondaire pour son utilisation en mémoire centrale. Les données sont récupérées en blocs de même taille appelés *pages*. Il s'agit d'une mémoire virtuelle. Elle permet d'augmenter virtuellement la taille de la mémoire. L'espace des adresses physiques d'un processus n'a pas a être contigu.
- **Segmentation de la mémoire:** technique de découpage de la mémoire gérée par l'unité de segmentation de l'unité de gestion de mémoire. Un segment est définie par son adresse de début et sa taille. Elle permet la séparation des données et du programme dans des espaces indépendants. Permet d'écrire/de lire en même temps sur des segments différents. Augmente la vitesse des opérations sur la mémoire.
- **Mémoire cache:** petite portion de mémoire à grande vitesse. Utilisée par le CPU pour réduire le temps d'accès moyen à la mémoire centrale. Elle range des copies des données fréquemment utilisées en mémoire centrale.Les CPUs modernes ont plusieurs caches indépendantes incluant une pour les instructions et une pour les données.

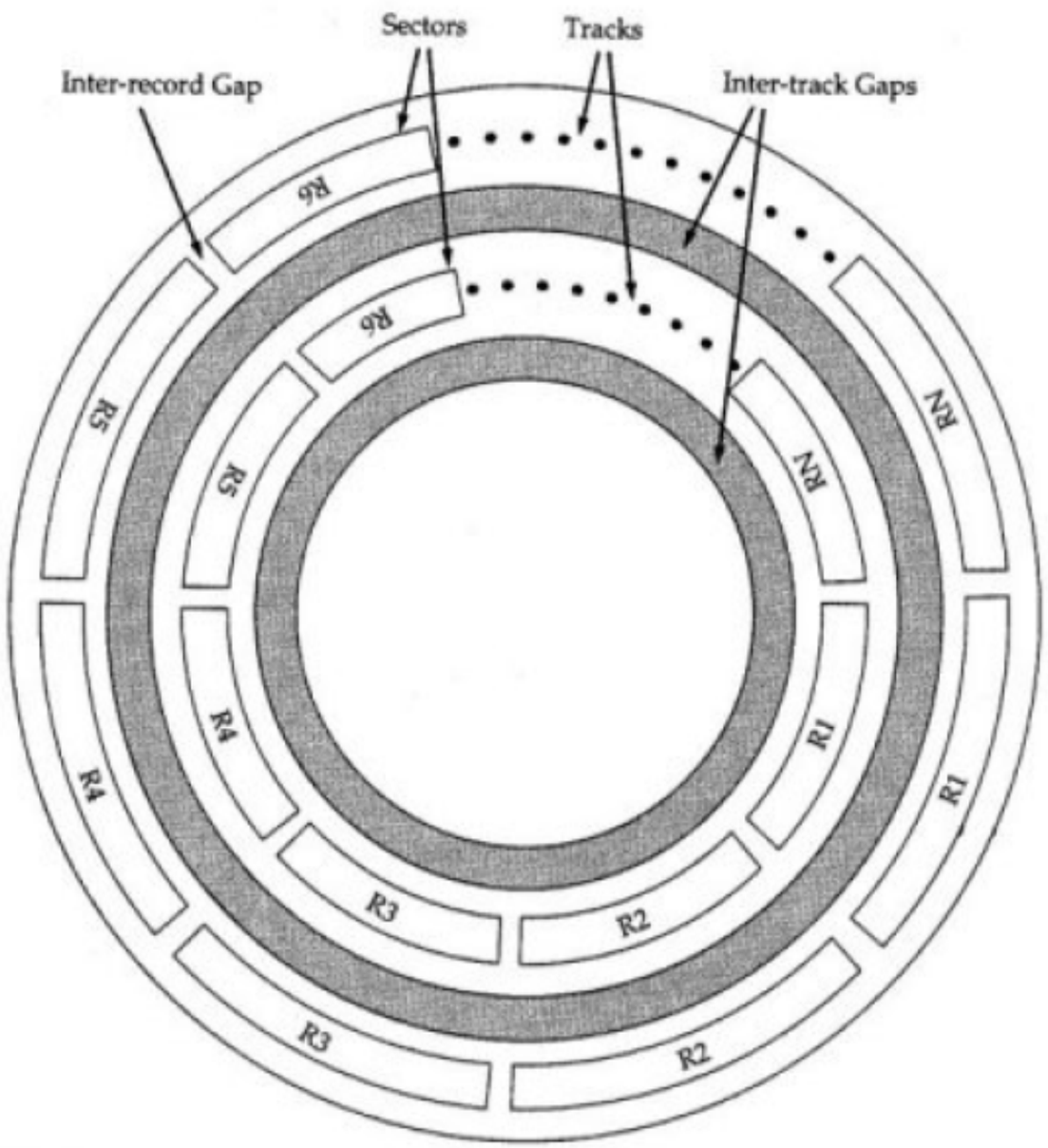
Mémoires auxiliaires

- Bandes magnétiques:
  - accès séquentiel,
  - lecture/écriture.

- Disques durs, disques ZIP, disquettes,…:
  - technologie magnétique,
  - taille de 1 To pour les disques durs et 1.4Mo,
  - lecture/écriture.



Organisation du disque dur



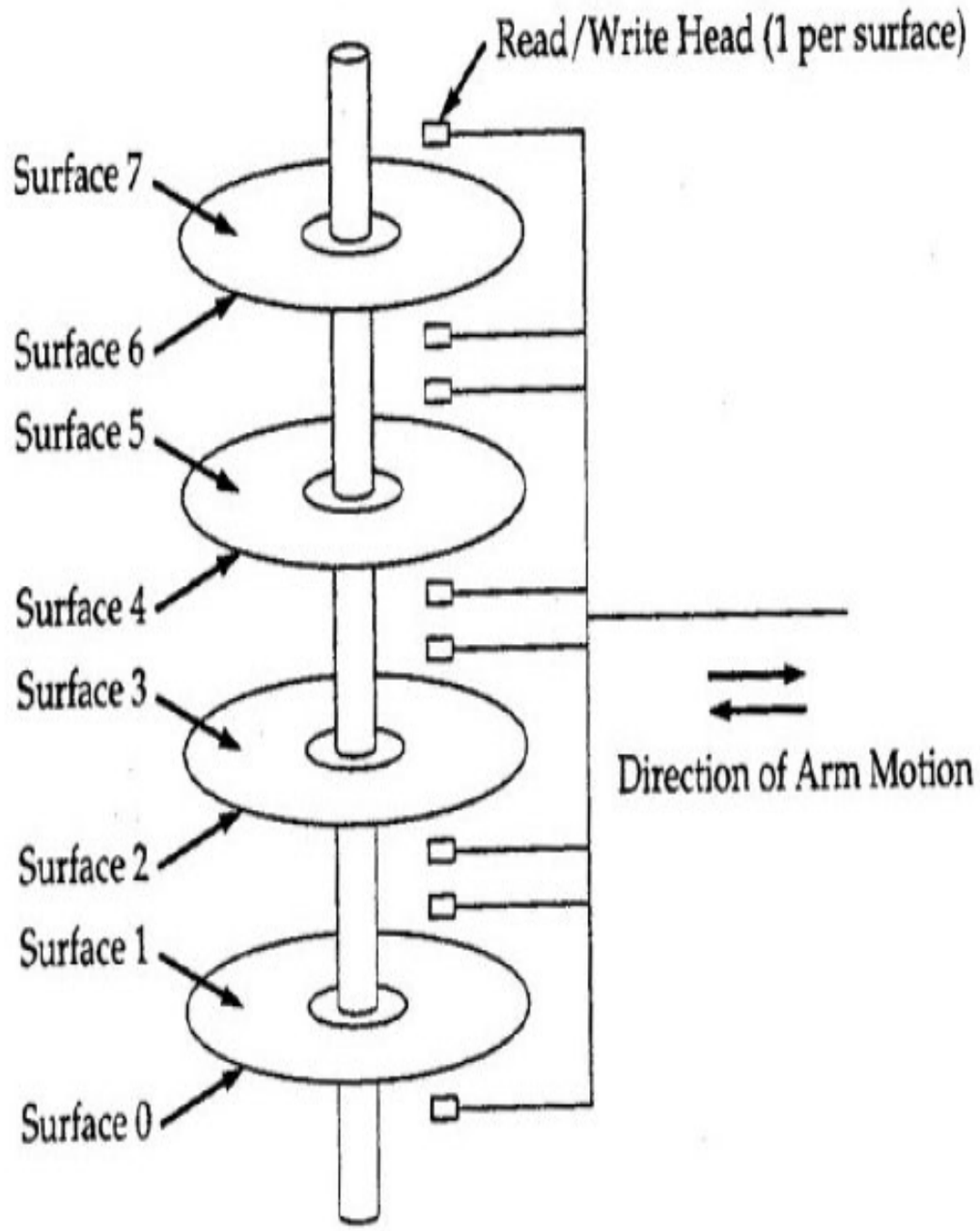


FIGURE 5.1. Disk data layout

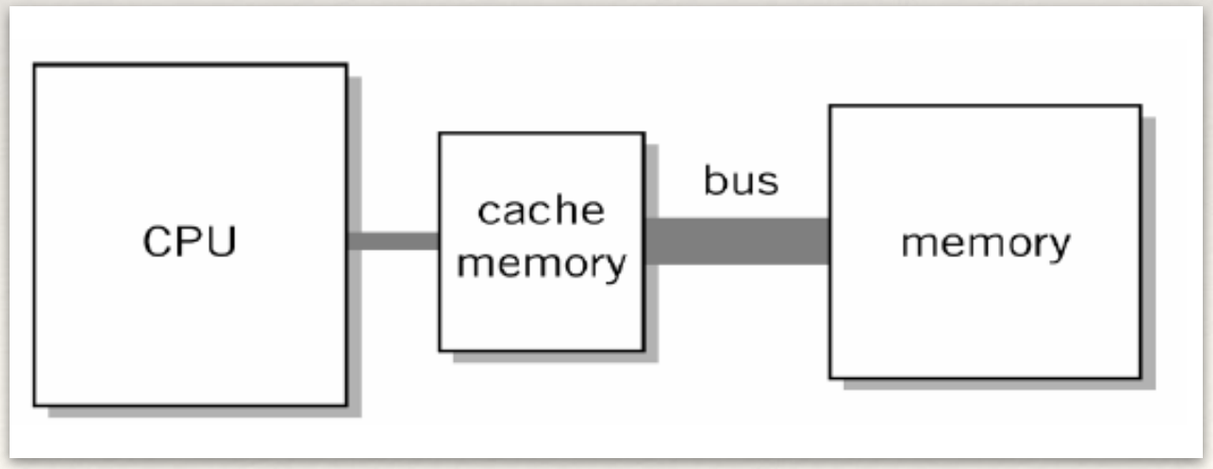
Mémoires auxiliaires (II)

- Disques optiques ou magnéto-optiques: CD-ROM et DVD.
  - Technologie optique,
  - 780 Mo pour les CD-ROMs et 4Go pour les DVDs.
  - Les CD-ROMs sont en lecture seulement tandis que les DVDs sont gravables une ou plusieurs fois.

- SSD (Solid state drive): utilise la mémoire flash.
- Mémoire flash: est une mémoire de masse à semi-conducteurs ré-inscriptible.

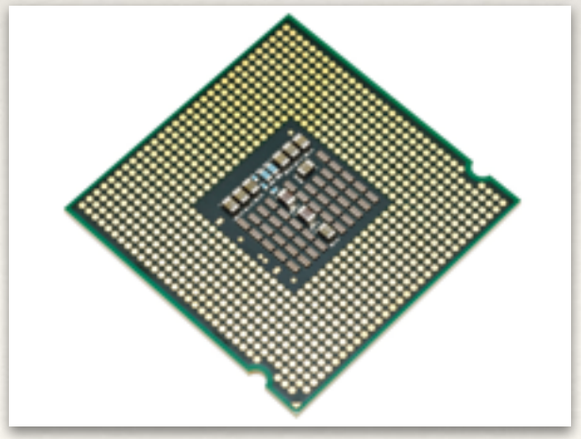
Mémoire cache

- La vitesse du CPU est beaucoup plus rapide que celle de la mémoire RAM. La mémoire cache permet d'accéder aux données en cache d'une façon très efficace.
- Invisible pour tout ce qui est à l'extérieur du CPU.
- Augmente la vitesse d'accès moyenne en contenant des copies de données souvent accédées.



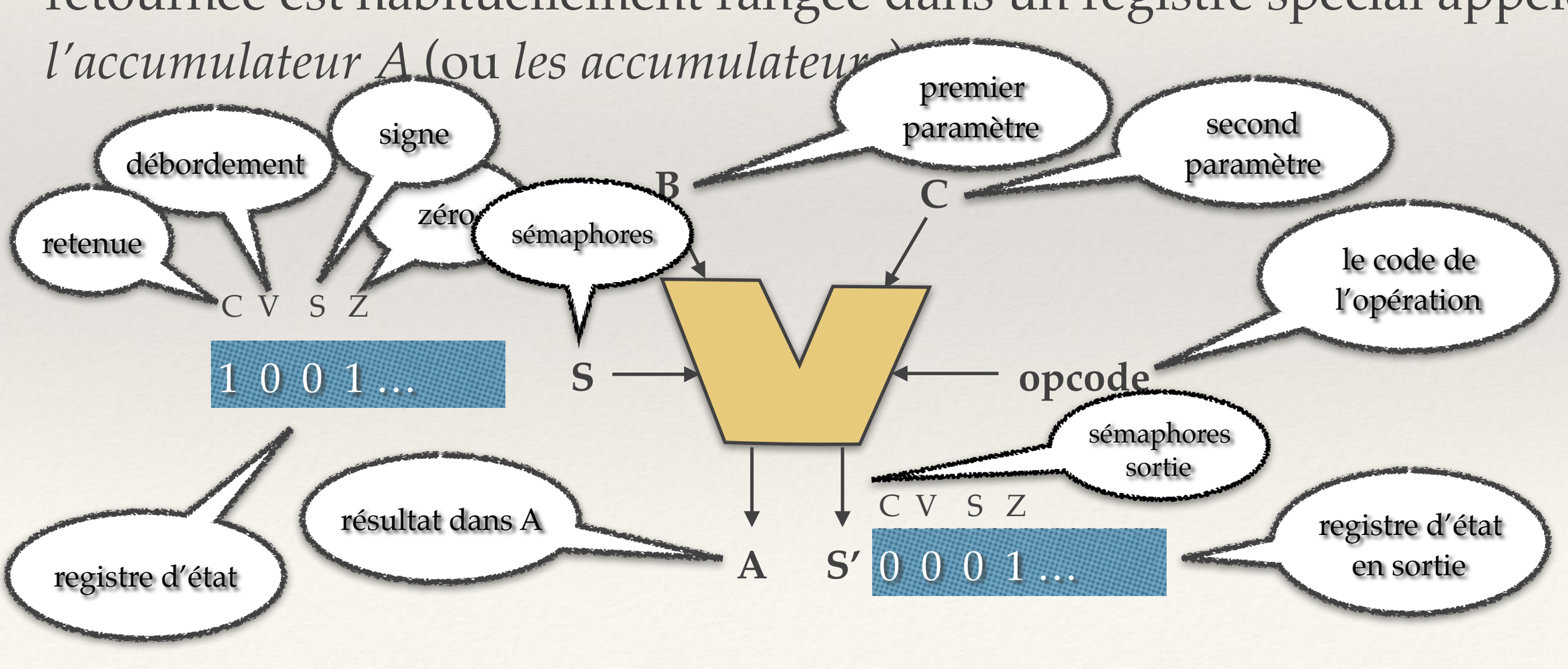
CPU: Unité centrale de traitement

- Un CPU contient habituellement:
  - une unité arithmétique et logique (ALU) et
  - une unité de contrôle/commande.
    - Il contient des registres internes pour ranger l'information avant de la transférer en mémoire. Un registre range l'instruction courante tandis qu'un autre range en ensemble de sémaphores représentant l'état de la dernière instruction complétée.
- L'unité de commande permet d'exécuter les commandes machines du processeur en invoquant l'ALU sur les valeurs des registres et de la mémoire.



l'ALU

- Peut être vu comme une fonction qui prend en entrée 2 entiers (de la taille des mots du CPU: B & C), un registre d'état (qui contient des sémaphores S) et le code d'une opération (opcode) pour retourner la valeur calculée (A) et le nouveau status (S'). La valeur retournée est habituellement rangée dans un registre special appelé *l'accumulateur A* (ou *les accumulateurs*)

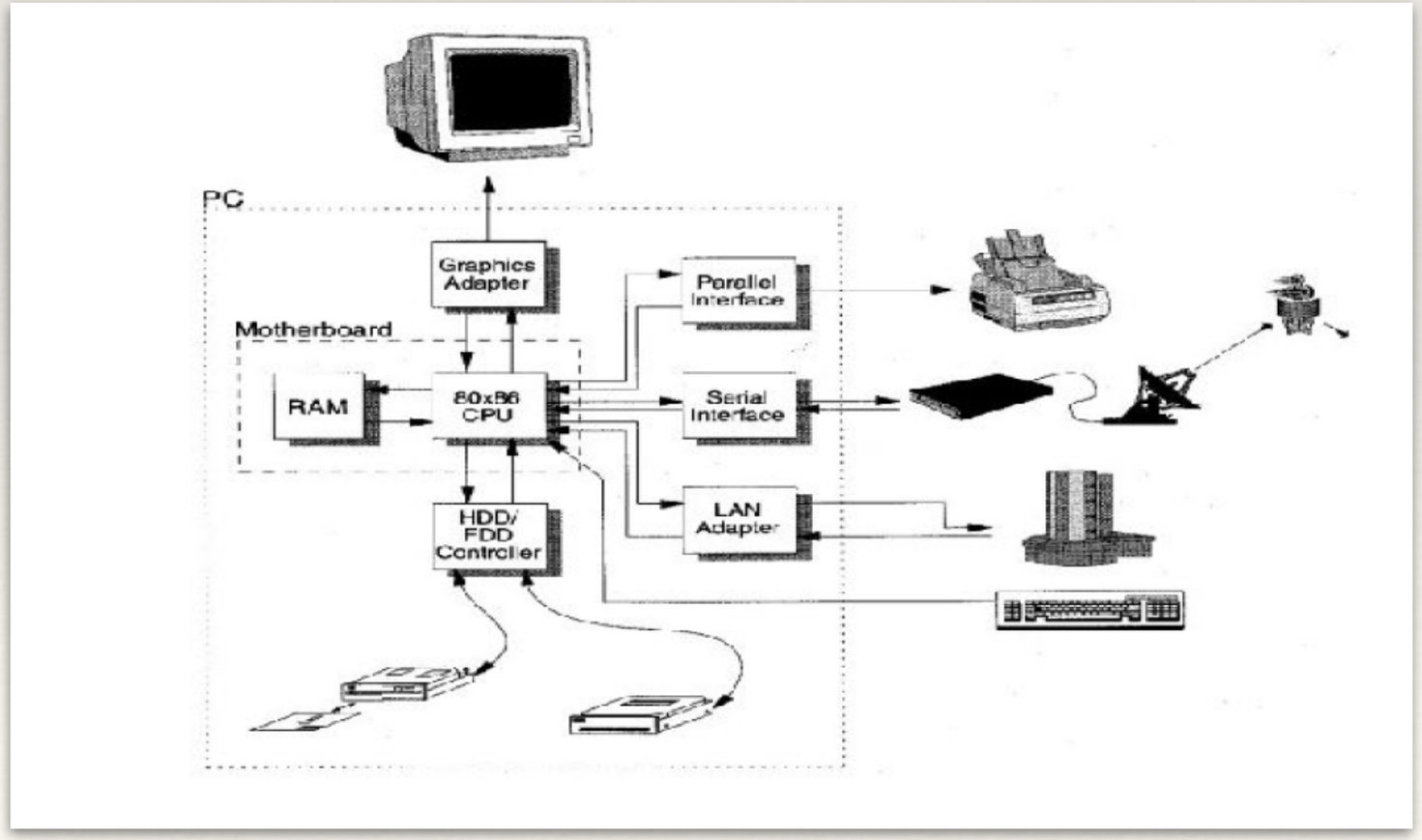


Unité de commande

- Elle fait partie du CPU.
- Elle contrôle le transfert des instructions à être exécutées et des données sur lesquelles l'instruction courante est appliquée. Elle transfert les valeurs calculées en mémoire.
    - L'instruction courante est rangée dans un registre spécial appelé le registre d'instruction (IR:instruction register),
    - L'adresse de la prochaine instruction en mémoire est rangée dans un autre registre spécial appelé compteur ordinal (PC:program counter),
    - Possède également une horloge pour ordonnancer tout ça.
- Transmet à l'ALU l'opération à exécuter ainsi que ses paramètres.
- Coordonne le fonctionnement des autres composants comme la mémoire centrale (via les cycles *saisir & écrire*).

Entrées/Sorties

- Servent d'interface avec les périphériques.
- Les opérations associées (lecture et/ou écriture) sont fonctions du périphériques.



Composants d'Entrées/Sorties

- Clavier: périphérique de saisie.
  - type: nombre de touches(QWERTY, AZERTY, …)
  - connexion: port standard, port USB, sans fil (IR ou radio).
- Souris: périphérique permettant le pointage rapide.
  - type: nombre de boutons, optique, mécanique, trackball,…
  - connexion: port série, USB, sans fil.
- Écran: périphérique de visualisation.
  - technologie: écran plat, rétina, …
  - résolution maximale
- Carte vidéo: offre une zone mémoire à accès multiple.
  - taille de mémoire et résolution,
  - type de connecteur (le bus),
  - instructions spécialisées de dessin 2D et 3D.
- Imprimante:
  - protocole de communication (postcript ou langage propriétaire),
  - technologie, couleur ou noir/blanc (matricielle, jet d'encre, laser, …)
  - résolution (entre 300 et 2400 DPI).

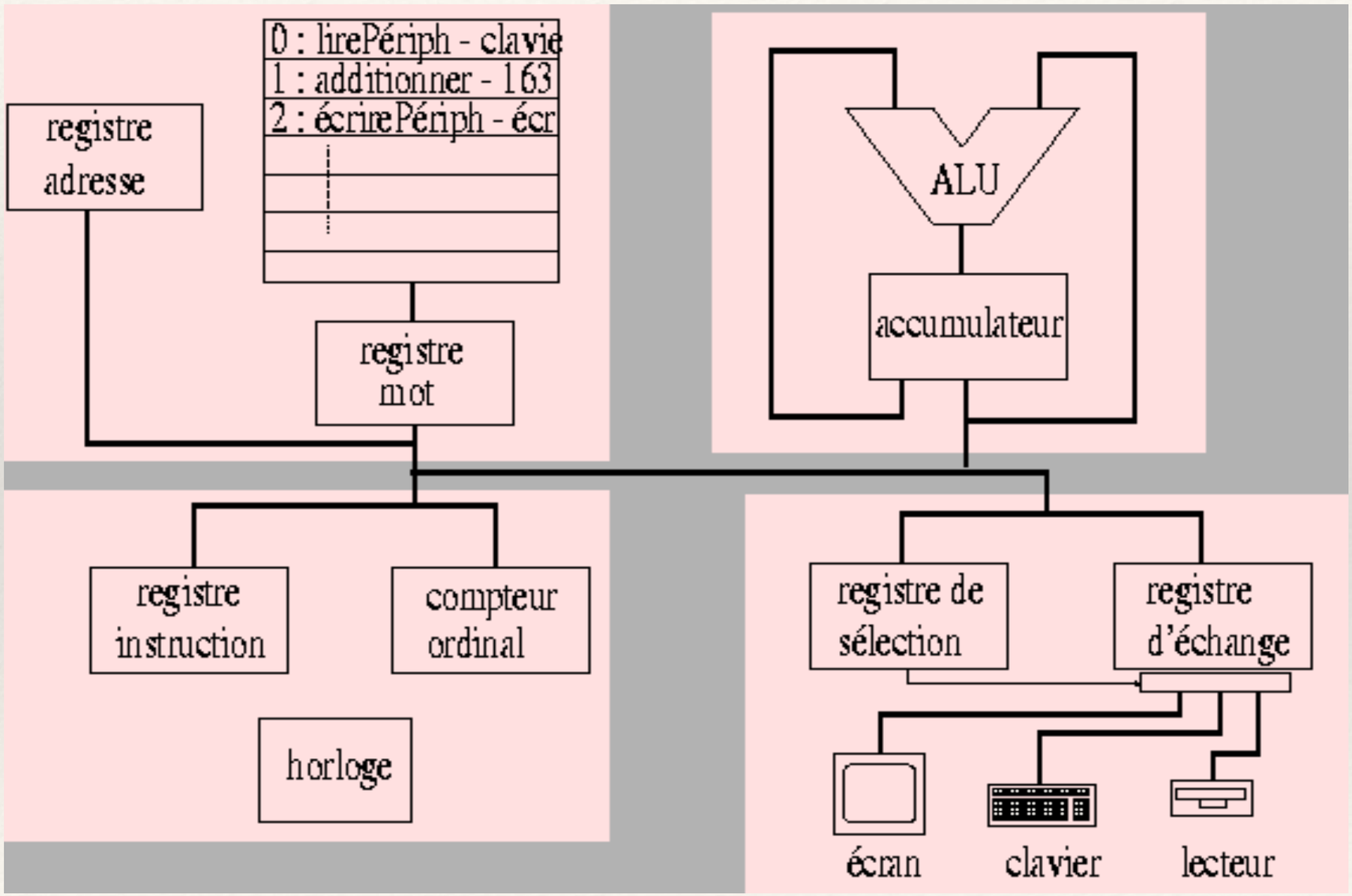
Composants d'Entrées/Sorties(II)

- Carte audio: permet l'exploitation des données audios.
  - nombre et nature des E/S (audio, midi,…)
  - stéréophonie (totale, sur certains canaux, certaines fréquences,…)
  - fréquence d'échantillonnage et espace de codage (de 8Khz 8 bits à 192Kh 16 bits).
- émetteur/reçeveur WI-FI: permet de transmettre et de recevoir des données sans fil.
- etc, etc, ….

Machine complète

Mémoire

ALU



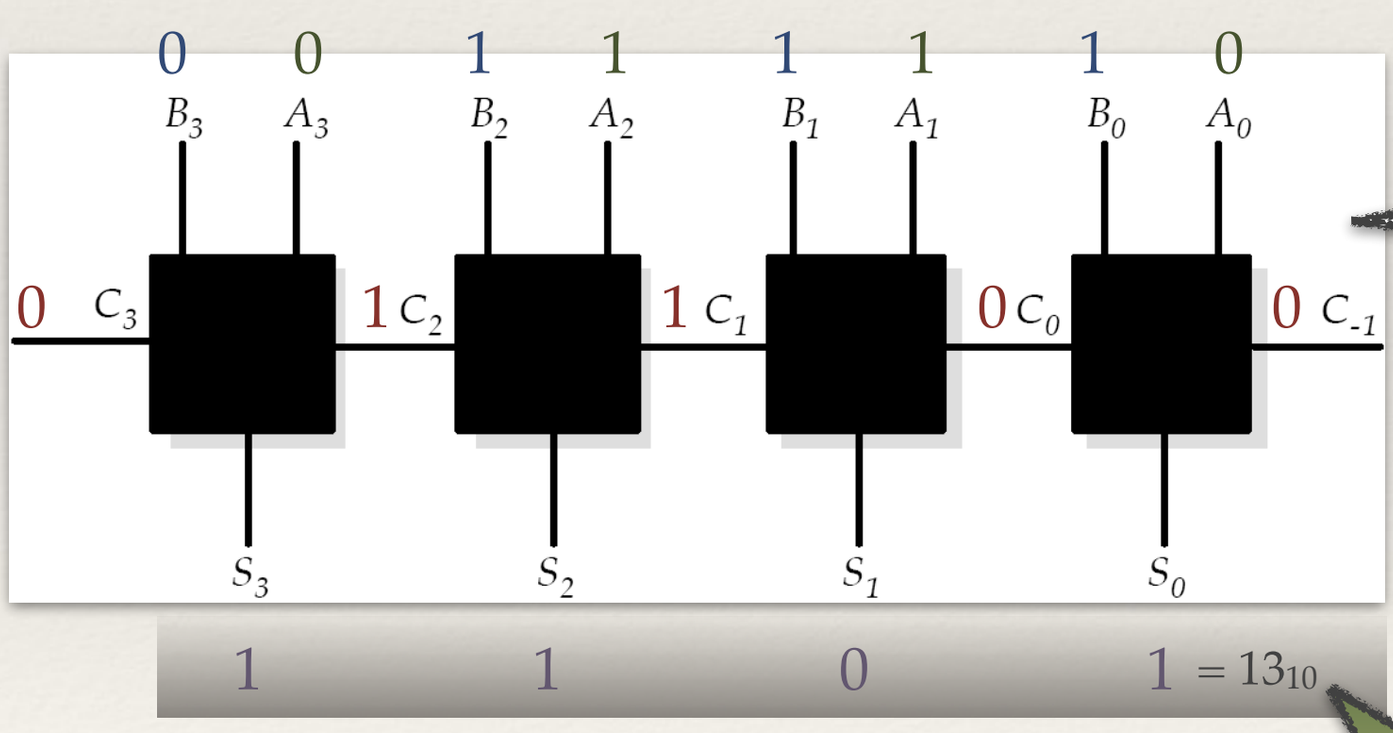
Unité de commande

E/S

Circuits de l'ALU

Additionneur en complément à 2: Soient $A=A_3,A_2,A_1,A_0$ et $B=B_3,B_2,B_1,B_0$ deux registres de quatre bits. Soit C le sémaphore de retenue (C_{-1} indique la retenue au départ du calcul).

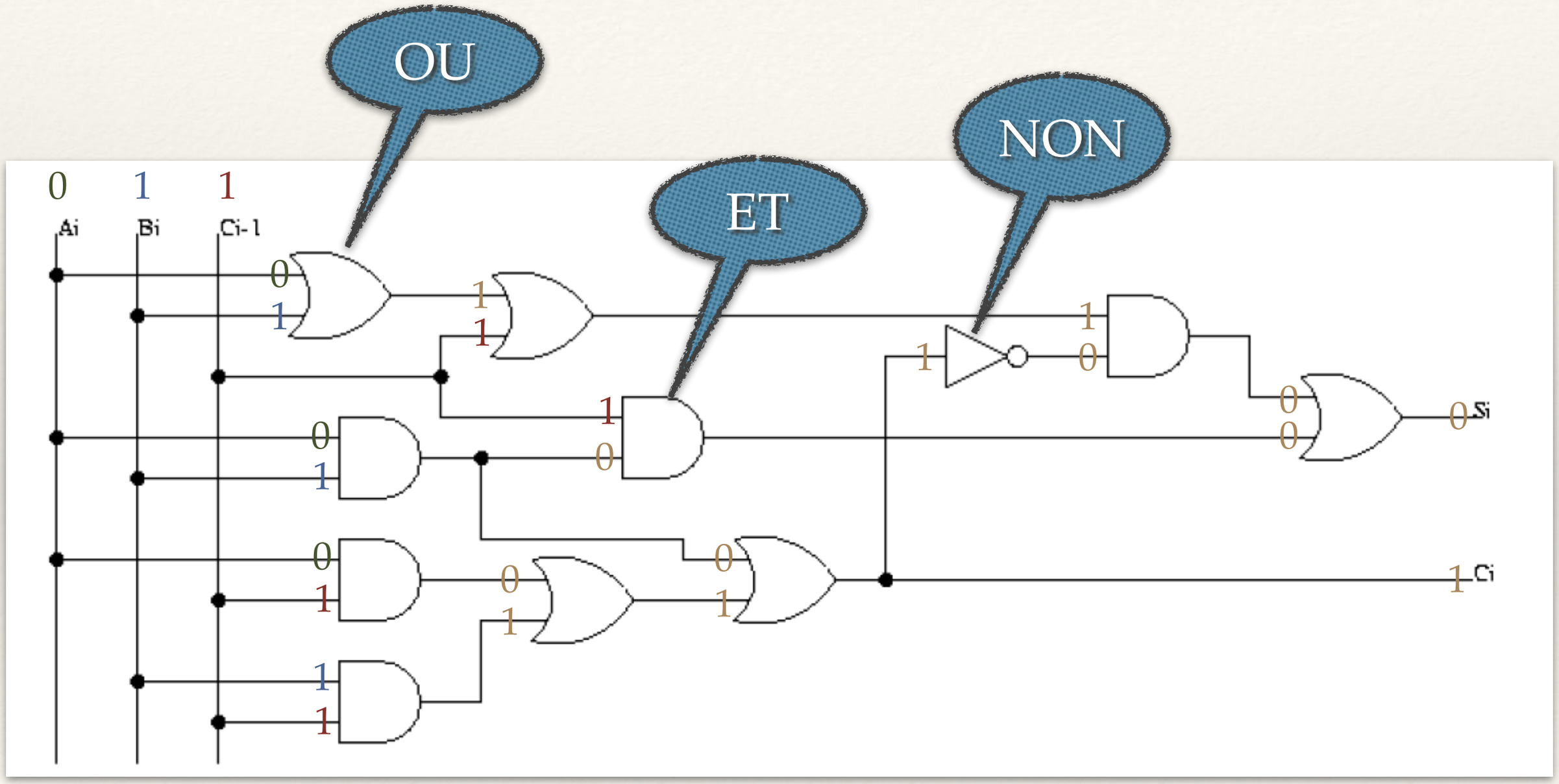
$A= \underset{A_3\,A_2\,A_1\,A_0}{0\,1\,1\,0} = 6_{10}$, $B= \underset{B_3\,B_2\,B_1\,B_0}{0\,1\,1\,1} = 7_{10}$



On additionne bit à bit avec la retenue héritée

A+B=1101=13_{10}

Le circuit d'addition de bit avec retenue



L'utilisation du complément à deux permet ici d'avoir le même circuit pour l'addition et la multiplication. Ceci simplifie l'ALU.

Attention aux débordements

- Un débordement (overflow) de capacité survient lorsque les opérandes ont le même signe mais le résultat a un signe différent.
- Un sémaphore (le sémaphore V) du CPU indique un débordement lors de la dernière opération si V=1 (V=0 autrement).

C V N Z ...

1 0 0 1 ...

registre d'état initial

	0 1 0 1 1 1 0 0	92_{10}
+	0 0 1 0 1 0 1 1	43_{10}
	1 0 0 0 0 1 1 1	-121_{10}

C V N Z

0 1 1 0 ...

registre d'état après l'addition

Instructions du langage machine (exemple: le 6502)

- Les instructions du langage machine sont spécifiées par deux composantes:
  - Le code d'opération: indique l'opération à exécuter,
  - Les paramètres pour l'opération (adresse mémoire ou valeur).
- Exemple: ADC #$AB
  - Additionne au contenu de l'accumulateur la valeur AB_{16}.

mode immédiat

indique l'hexadécimal

Le code machine de cette instruction est donc 09AB en hexadécimal.

code d'opération pour ADC en mode immédiat

opcode 8 bits et 8 bits pour la valeur de X

00001001 10101011

Instructions du langage machine (exemple: le 6502) (II)

- Exemple: ADC $302A

  indique l'hexadécimal

  mode absolue

  - Additionne au contenu de l'accumulateur la valeur de l'octet rangée à l'adresse $302A_{16}$.

opcode 8 bits et 8 bits pour l'adresse cible

00001101 00101010 00110000

En hexadécimal l'instruction machine est:

0 D 2 A 3 0

Cette opération est encodée par 3 octets.

L'ensemble d'instructions

- L'ensemble des instructions d'un micro-processeur peut répondre à l'une des deux approches suivantes:
  - **RISC:** Ordinateurs avec un ensemble réduit d'instructions. Ces instructions sont d'un design simple et très efficace. Des programmes assez complexes doivent cependant être écrits pour faire des choses plus sophistiquées.
  - **CISC:** Ordinateurs avec un ensemble complexe d'instructions. Contient beaucoup d'instructions et certaines très complexes (p.e. un ensemble d'instruction pour faire des calculs compliqué sur des très grands nombres)

Instructions typiques

(le 6502)

- Nous supposons que le micro-processeur possède trois registres de calcul, l'accumulateur A, le registre X et le registre Y.
- Nous dénotons par W et Z des adresses de mémoire RAM.
- Les instructions de transfert de données:
  - LDA W (range dans A la valeur CONT(W))
  - STA W (range le contenu de A à l'adresse W)
  - LDX W, LDY W (range dans X et Y la valeur CONT(W))
  - STX W, STY W (range le contenu de X et Y à l'adresse W)

CONT(W) désigne le contenu de l'adresse W en mémoire.

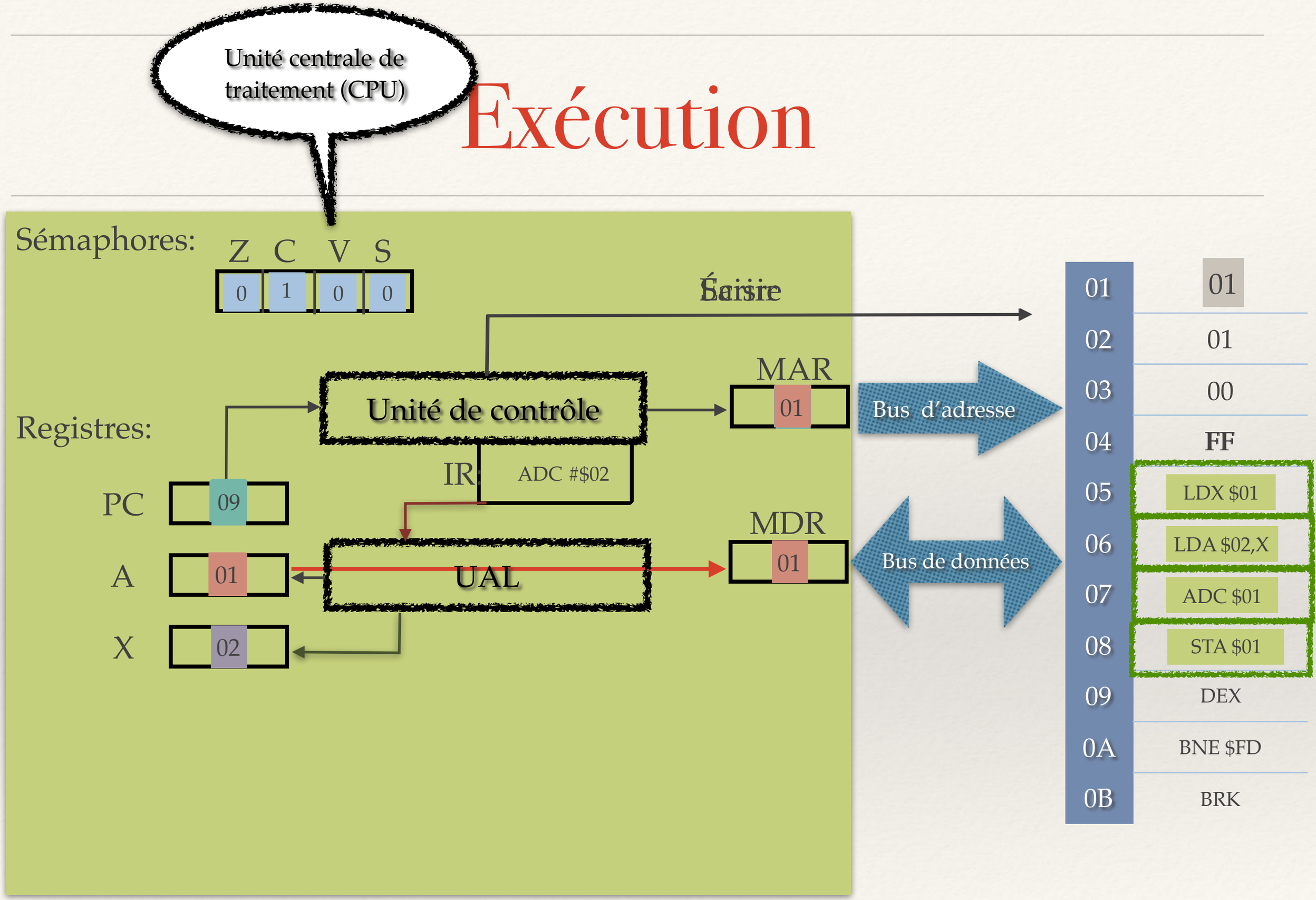
Instructions typiques (II)

- Les instructions arithmétiques:
  - ADC W, SBC W (additionne ou soustrait à A la valeur CONT(W)). S'il y a retenue alors le sémaphore C=1 (i.e. la retenue). Si la valeur obtenue en complément à deux est trop petite ou trop grande alors le sémaphore V=1 (voir la démo à ce sujet).
  - ASL W, ASL A(rotation arithmétique à gauche de CONT(W) ou de A). Le sémaphore C reçoit le bit 7. Le bit 7 reçoit le bit 6 qui reçoit le bit 5,…., qui reçoit le bit 1. Le bit 0 reçoit 0.
  - ROR W, ROL W, ROR A, ROL A (rotation à droite, à gauche de CONT(W) ou de A). Le sémaphore C entre et la valeur en sortie est rangée dans C.
  - AND W, ORA W (calcul le ET et le OU bit-à-bit de A avec le contenu de l'adresse W). A contient le résultat final.
  - INX, INY, DEX, DEY (incrémente et décrémente les registres X et Y). S'il y a débordement alors le sémaphore C=1.
  - INC W, DEC W (incrémente et décrémente CONT(W)). S'il y a débordement alors le sémaphore C=1.

Instructions typiques (III)

- Instructions de comparaison:
  - CMP W (affecte les sémaphores comme pour SBC W).
  - CPX W, CPY W (comme pour CMP en remplaçant A par X et Y).
  - BIT W (affecte les sémaphores comme pour AND W).
- Instructions de branchement:
  - JMP W, JSR W (saute à l'adresse W sans et avec retour).
  - BNE W, BPL W (branche à l'adresse W si Z=0 et N=0 resp.)
  - BCC W, BCS W, BMI W (branche à l'adresse W si C=0/C=1 et N=1 resp.).
  - BVC W, BVS W (branche à l'adresse W si V=0/V=1 reso,).

Exécution


Unité centrale de traitement (CPU)
Sémaphores:
Z C V S
0 1 0 0
Registres:
Unité de contrôle
IR
ADC #$02
PC
09
A
01
X
02
UAL
MAR
01
MDR
01
Saisie
Bus d'adresse
Bus de données
01 01
02 01
03 00
04 FF
05 LDX $01
06 LDA $02,X
07 ADC $01
08 STA $01
09 DEX
0A BNE $FD
0B BRK

L'assembleur

- L'assembleur est un langage très près du langage machine du CPU. Il remplace les opcodes par des mnémoniques. Pour chaque mnémonique, plusieurs modes d'adressage peuvent être utilisés. Ces modes sont indiqués en utilisant des variantes des mnémoniques. Le programme précédent était décrit en *assembleur*.
- On peut associer à une adresse physique une étiquette. Les instructions qui accèdent à l'adresse physiques peuvent être écrites comme si elles accédaient à l'étiquette. Ceci permet de faire abstraction de l'endroit où le programme sera rangé pour son exécution. Aide le programmeur à écrire ses programmes.
- Les instructions sont celles du langage machine comme données précédemment…

Des modes d'adressage du 6502

Modes	Type	Fonction	Note
LDA #$3A	Immédiat	Range $3A dans A	
LDA $3A	Relatif	Range le contenu de $3A dans A	
LDA $3A,X	Relatif indexé	Range le contenu de $3A+val(X) dans A	Sur le 6502 l'adresse dans ce mode doit tenir sur 8 bits: la page 0 de la RAM
LDA ($3A)	Indirect	Range dans A le contenu de l'adresse rangée en $3A (bas) et $3B (haut)	
LDA ($3A),Y	Indirect indexé (I)	Range dans A comme pour ($3A)+val(Y) (i.e. page 0 pour $3A)	
LDA ($3A,X)	Indirect indexé (II)	Range dans A comme pour ($3A+val(X)) (i.e. page 0 pour $3A)	

Exemples de code assembleur pour le 6502

```
        LDX #122
        STX $00
        LDX #16
        STX $01
        LDA $00
        ADC $01
        BCS retenue
fin:    STA $00
        BRK
retenue: LDX #255
        STX $01
        JMP fin
```

```
Address  Hexdump    Dissassembly
-------------------------------
$0600    a2 7a      LDX #$7a
$0602    86 00      STX $00
$0604    a2 10      LDX #$10
$0606    86 01      STX $01
$0608    a5 00      LDA $00
$060a    65 01      ADC $01
$060c    b0 03      BCS $0611
$060e    85 00      STA $00
$0610    00         BRK
$0611    a2 ff      LDX #$ff
$0613    86 01      STX $01
$0615    4c 0e 06   JMP $060e
```

retourne
le maximum des contenus de
mémoire 00,01,02
le maximum est retourne

```
    LDX #2
    LDA $00,X
    TAY
ici: TYA
    DEX
    BMI fin
    TAY
    SBC $00,X
    BPL ici
    LDA $00,X
    TAY
    JMP ici
fin: BRK
```

```
Address  Hexdump    Dissassembly
-------------------------------
$0600    a2 02      LDX #$02
$0602    b5 00      LDA $00,X
$0604    a8         TAY
$0605    98         TYA
$0606    ca         DEX
$0607    30 0b      BMI $0614
$0609    a8         TAY
$060a    f5 00      SBC $00,X
$060c    10 f7      BPL $0605
$060e    b5 00      LDA $00,X
$0610    a8         TAY
$0611    4c 05 06   JMP $0605
$0614    00         BRK
```

Le X-TOY

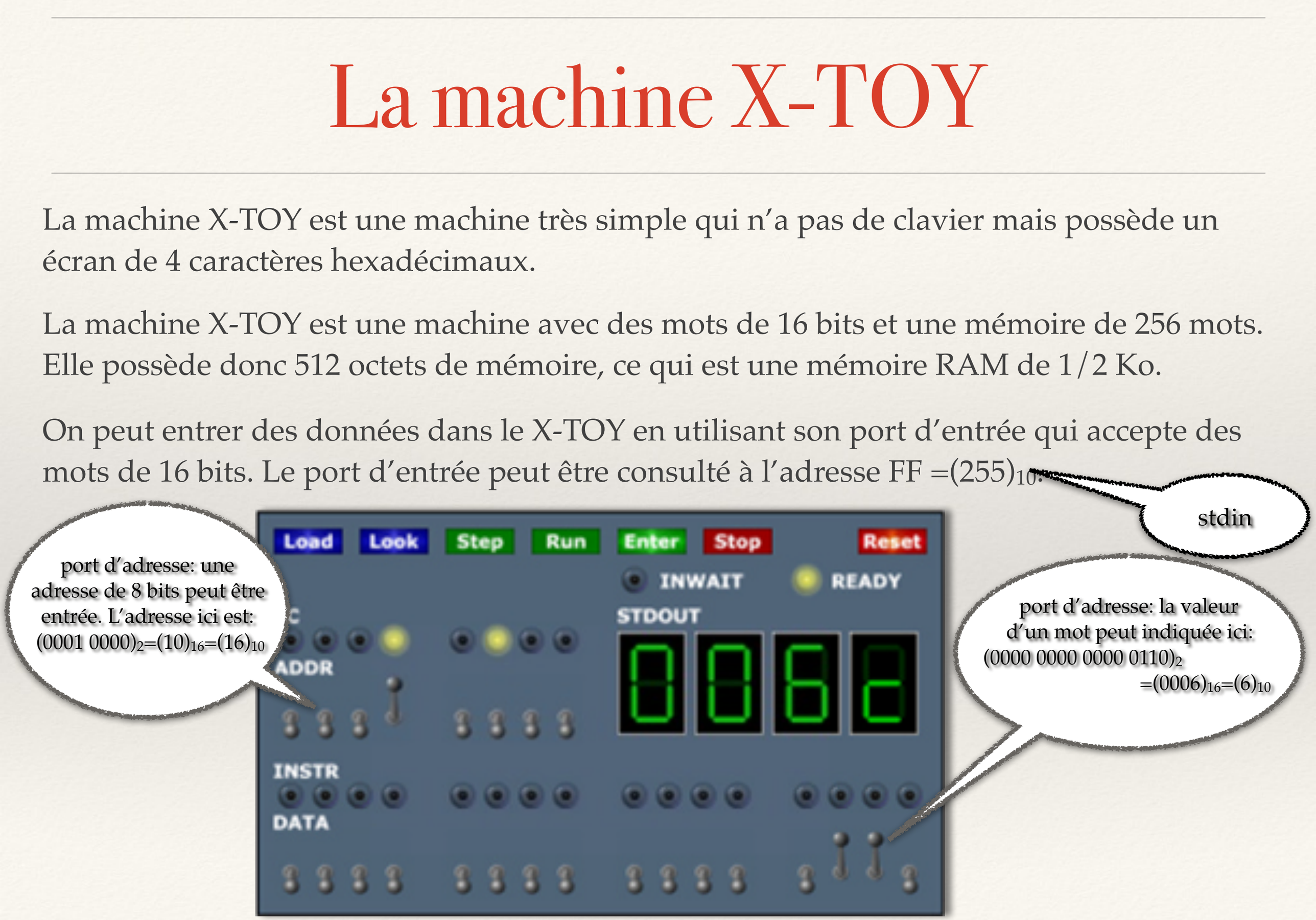
- Le X-TOY est une machine virtuelle qui ressemble à l'ALTAIR-8800. Elle a été définie à l'Université Princeton.
- Il peut être programmé en langage machine par une application java. Son langage machine est inspiré des mini-ordinateurs de la série PDP (*Digital Equipment Corporation*, 1959-1990).
- Le langage machine du X-TOY est extrêmement simple mais possède toute la puissance d'autres langages comme Java, C, C++, Javascript, etc…La langage du X-TOY est même plus puissant que SQL!
- Votre premier TP vous demandera de le programmer pour résoudre un problème simple.

La machine X-TOY

La machine X-TOY est une machine très simple qui n'a pas de clavier mais possède un écran de 4 caractères hexadécimaux.

La machine X-TOY est une machine avec des mots de 16 bits et une mémoire de 256 mots. Elle possède donc 512 octets de mémoire, ce qui est une mémoire RAM de 1/2 Ko.

On peut entrer des données dans le X-TOY en utilisant son port d'entrée qui accepte des mots de 16 bits. Le port d'entrée peut être consulté à l'adresse FF $=(255)_{10}$.



et l'Altair 8800

La machine X-TOY (II)

- X-TOY peut être simulée sur nos ordinateurs par une application Java.
- L'application nous permet de la programmer directement en langage machine et d'exécuter des programmes sur celle-ci.
- So langage machine ne possède que 16 instructions, c'est l'approche **RISC** extrême!!!
- Le fait d'avoir si peu d'instructions rend la programmation un peu plus compliquée. Sa puissance intrinsèque est cependant la même que les machines que vous possédez.

Éléments de la machine X-TOY

- RAM:
  - Une mémoire RAM de 256 mots de 16 bits aux adresses hexadécimales 0 à FF.
  - L'adresse FF_{16} est utilisée pour communiquer avec `stdin` et `stdout`. Ce qui est à l'adresse FF_{16} apparaît sur l'écran: `stdout`. Ce qui est récupéré à l'adresse FF_{16} vient de `stdin`.
- CPU:
  - 16 registres de 16 bits sont dénotés par R[0],R[1],R[2],…,R[9],R[A],R[B],R[C],R[D],R[E] et R[F].
  - R[0] vaut toujours 0000_{16}, c'est une constante.
  - Un registre supplémentaire pour ranger le compteur ordinal PC.
  - 16 instructions séparées en trois groupes:
    - **Chargement/Rangement:** Charge les registres avec des éléments en mémoire et range la valeurs de registre en mémoire. Les adresses consultées peuvent être indiquées directement (adressage immédiat) ou par l'adresse contenue dans un registre (adressage indirect).
    - **Arithmétique:** Additionne et soustrait des registres, applique le ET, le OU, le OUX entre deux registres.
    - **Branchement:** Permet de modifier le PC en fonction
      - de l'adresse contenu dans un registre,
      - d'une condition sur la valeur d'un certain registre (les deux conditions possibles sont R[x]>0 et R[x]=0).

OPCODE	DESCRIPTION	FORMAT	PSEUDOCODE
0	*halt*	-	`exit`
1	*add*	1	`R[d] <- R[s] + R[t]`
2	*subtract*	1	`R[d] <- R[s] - R[t]`
3	*and*	1	`R[d] <- R[s] & R[t]`
4	*xor*	1	`R[d] <- R[s] ^ R[t]`
5	*left shift*	1	`R[d] <- R[s] << R[t]`
6	*right shift*	1	`R[d] <- R[s] >> R[t]`
7	*load address*	2	`R[d] <- addr`
8	*load*	2	`R[d] <- mem[addr]`
9	*store*	2	`mem[addr] <- R[d]`
A	*load indirect*	1	`R[d] <- mem[R[t]]`
B	*store indirect*	1	`mem[R[t]] <- R[d]`
C	*branch zero*	2	`if (R[d] == 0) pc <- addr`
D	*branch positive*	2	`if (R[d] > 0) pc <- addr`
E	*jump register*	-	`pc <- R[d]`
F	*jump and link*	2	`R[d] <- pc; pc <- addr`

Nombre de paramètres pour l'instruction

Autre façon de noter l'instruction. Plus facile à comprendre

Du pseudocode à l'hexadé

Les instructions sont encodées à l'aide des deux formats suivants:

Format 1: Une instruction dont les arguments sont des registres,

Format 1: Une instruction dont un argument est une adresse.

Un numéro de registre entre 0..F

Un numéro de registre entre 0..F

E

A

0

4

	15	12	11	8	7	4	3	0
	1	0	1	1	1	0	1	0	0	0	0	0	0	1	0	0
Format 1	opcode		dest d		source s		source t	
Format 2	opcode		dest d		addr			

d'après la liste d'opcodes donnée précédemment.

Un numéro de registre entre 0..F

Les instructions avec opcodes A et B ignorent les bits 4 à 7: A80B=A81B=A82B=…

Une adresse sur 8 bits

- **E** permet de modifier le compteur ordinal PC en le posant PC<-R[d] ou d est un registre {0..F}. L'exécution sautera alors à l'adresse contenue dans R[d].
- Cette instruction ignore les bits aux positions 0..7 (l'octet le plus à droite du mot):EA00 est la même instruction que EA0A (tout comme EABB, EA12,…).

Exemples de programme

Un programme qui additionne les contenus des adresses 00 et 01 et range le résultat à l'adresse 02. Regardons le pseudo-code en premier lieu:

```
00: 0005    5
01: 0008    8

10: 8A00    R[A] <- mem[00]
11: 8B01    R[B] <- mem[01]
12: 1CAB    R[C] <- R[A] + R[B]
13: 9C02    mem[02] <- R[C]
14: 0000    halt
```

Y a-t-il des valeurs pour `mem[00]` et `mem[01]` qui feront en sorte que `mem[02]` ne contiendra pas la bonne réponse à la fin?

Exemples de programme (II)

Voici un programme qui accepte deux entiers de `stdin`, calcule le OUX des arguments et imprime le résultat sur `stdout`:

```
10:   8AFF     READ R[A]
11:   8BFF     READ R[B]
12:   4CAB     R[C] <- R[A] ^ R[B]
13:   9CFF     WRITE(R[C])
```

Exemples de programme (III)

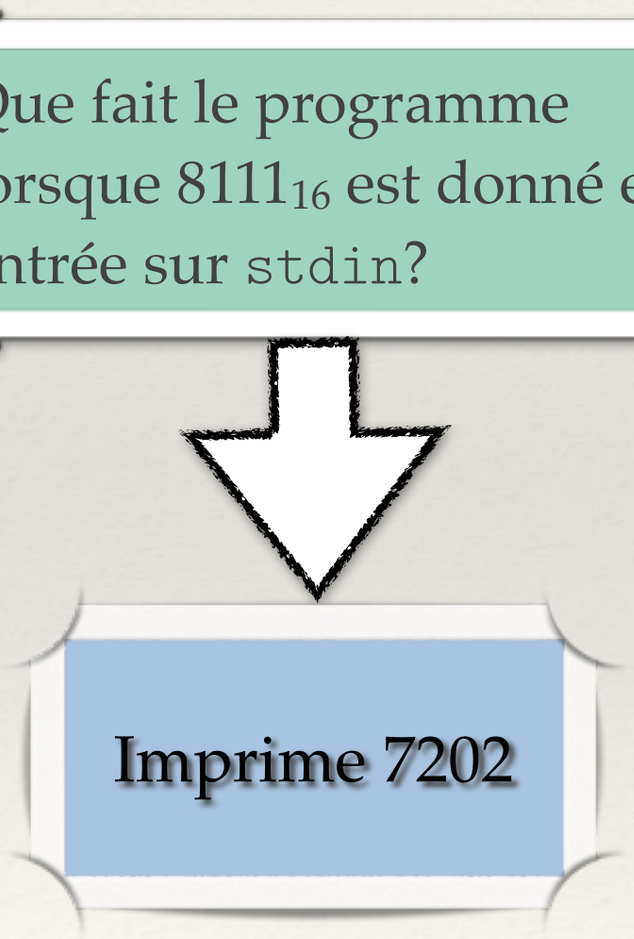
- Voici le comble du programme inutile, que fait-il?
- Rappelez-vous que R[0] vaut toujours 0000_{16}.

```
10:  9010    mem[10] <- R[0]
11:  9011    mem[11] <- R[0]
12:  9012    mem[12] <- R[0]
13:  0000    HALT
```

Exemples de programme (IV)

Ce programme range 0,2,3 dans R[1],R[2] et R[3] respectivement. Charge l'entrée sur `stdin` comme instruction à la ligne 15. La ligne 15 est alors exécutée avant d'imprimer la valeur du registre R[1] sur `stdout`.

Que fait le programme lorsque 8111_{16} est donné en entrée sur `stdin`?




```
program re-program
// Input:    une instruction sur stdin
// Output:   imprime R[1]
// charge les registre R1,R2,R3 avec 0,2,3
//
10: 7100    R[1] <- 0
11: 7202    R[2] <- 2
12: 7303    R[3] <- 3
13: 8AFF    read(R[A])
14: 9A15    mem[15] <- R[A]
15: 1000    nop
16: 91FF    write(R[1])
```

Exemples de programmes (V)

Voici comment utiliser les boucles et les tests pour calculer des trucs plus complexes.

Le programme suivant multiplie deux mots du `stdin` et écrit le résultat sur `stdout`:

```
10: 8AFF      read R[A]
11: 8BFF      read R[B]

12: 7C00      R[C] <- 0000
13: 7101      R[1] <- 0001

14: CA18      if (R[A] == 0) goto 18
15: 1CCB        R[C] <- R[C] + R[B]
16: 2AA1        R[A] <- R[A] - R[1]
17: C014      goto 14
18: 9CFF      write R[C]
19: 0000      halt
```

constante 1

Indenter code pour indiquer l'intérieur d'une boucle aide la compréhension.

`goto 14`
est la même chose que
`if(R[0]==0) goto 14`

Exécutées R[A] fois.
(à chaque fois on ajoute R[B] à R[C])

Tri à bulles (Bubblesort)

Le tri à bulles est une méthode pour trier une liste d'entiers possiblement négatifs dans un ordre quelconque pour produire une liste de ces valeurs triée en ordre croissant (ou décroissant):

Les mots à trier sont donnés sur le `stdin`.

Les mots triés sont donnés sur le `stdout`.

Les mots trier sont aussi rangées en mémoire à partir de l'adresse 00_{16}.

```
program Bubble
// Input:   N<10 nombres donnes par stdin termines par un nombre qui
//          produit un debordement lorsque multiplie par 2: {-2^14..2^14 -1}
//          Aucun element a l'exterieur de cet ensemble n'est accepte.
//          Un nombre a l'exterieur de l'ensemble indique la fin de la liste.
// Output:  les nombre tries en memoire a partir de la memoire 00, les
//          valeurs en ordre croissant sont donnees sur le stdout.
//   ATTENTION: lorsqu'aucun element n'est donne, l'entier utilise
//              pour terminer la liste vide est retourne. Joue le role
//              d'un message d'erreur.
// ------------------------------------------------------------------
10: 7101                    1E: 1FF1
11: 8BFF                    1F: 2DAF
12: BB0A                    20: DD19
13: 1AA1                    21: 0000
14: 1DBB                    22: 2EC1
15: 4DDB                    23: A70C
16: DD11                    24: A60E
17: CD11                    25: 2876
18: 2AA1                    26: D829
19: 2CA1                    27: B60C
1A: 2DCF                    28: B70E
1B: DD22                    29: 2CC1
1C: AD0F                    2A: C01A
1D: 9DFF
```

Documentation et simulateur

- Vous pouvez avoir plus d'information sur la machine virtuelle X-TOY à:
  - http://introcs.cs.princeton.edu/java/50machine/
- Vous pouvez télécharger le simulateur X-TOY à:
  - http://introcs.cs.princeton.edu/xtoy/