Architecture et programmation des ordinateurs

PUBLIC CONCERNE : première année DUT Informatique

Jacques LONCHAMP

DATE : 2010/2011

UNIVERSITE NANCY 2
INSTITUT UNIVERSITAIRE DE TECHNOLOGIE
2ter boulevard Charlemagne
CS 55227
54052 NANCY Cedex

Tél : 03.54.50.38.00
Fax : 03.54.50.38.01
http://iut-charlemagne.univ-nancy2.fr

Table des matières

Programme pédagogique national du DUT Informatique (2005)

U.F. ARCHITECTURE DE L'ORDINATEUR : TC-INFO-ASR1
Volume horaire : 30h – Pré-requis : aucun.
Objectifs :

- Comprendre le fonctionnement général d'un microprocesseur et de son environnement matériel.

Compétences minimales :

- Connaître les méthodes de codage et de représentation de l'information, et les traitements associés.
- Connaître le fonctionnement des circuits combinatoires et séquentiels associés au traitement de ces données.

Contenu :

- Codage de l'information : numération, représentation des nombres et codage en machines, codage des caractères, arithmétique et traitement associés.
- Éléments logiques : algèbre de Boole, circuits logiques combinatoires (décodeur, additionneur, unité de calcul), systèmes séquentiels simples (registres, compteurs).
- Microprocesseur : microprogrammation, séquencement, bus, langage machine, interruptions, composants externes (mémoires, contrôleurs, périphériques).

Indications de mise en oeuvre :

- Interactions souhaitables avec l'enseignement des mathématiques (représentation des nombres, algèbre de Boole).
- L'étude du microprocesseur et de son environnement matériel peut faire l'objet de l'examen (voire de l'assemblage) d'un véritable ordinateur et de ses composants.

U.F. ARCHITECTURE ET PROGRAMMATION : TC-INFO-ASR2
Volume horaire : 30h – Pré-requis : U.F. TC-INFO-ASR1.
Objectifs :

- Comprendre l'implémentation bas niveau des mécanismes liés aux langages de haut niveau.

Compétences minimales :

- Manipuler les concepts du langage machine.
- Connaître l'influence des architectures des microprocesseurs modernes sur les performances des programmes.

Contenu :

- Langage machine : pile système, modes d'adressage, jeux d'instructions, langage d'assemblage.
- Mécanismes de haut niveau : gestion des données par le compilateur (données statiques/dynamiques, pile, tas), arbres de calcul, appel de fonctions/procédures.
- Processeurs modernes : mémoire cache, pipeline, instructions SIMD, performance des programmes.

Indications de mise en oeuvre :

- Des séances de travaux pratiques peuvent être mises en place pour visualiser, au travers d'exemples simples, les concepts abordés (écriture de courts programmes en langage d'assemblage, étude du code généré par un compilateur, mesures de performance de programmes).

1 Introduction

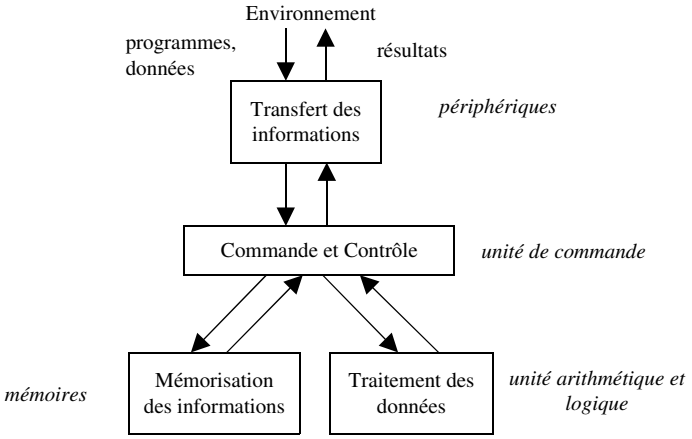
1.1 Définitions

Il n'est pas facile aujourd'hui de décrire ce qu'est un ordinateur en raison de la grande diversité des machines existantes (supercalculateurs, serveurs, postes de travail fixes ou portables, assistants personnels, puces embarquées dans tous les objets de la vie courante) et de la rapidité des évolutions technologiques. Il existe cependant un certains nombre de *principes d'organisation et de fonctionnement* que ce cours cherche à présenter de manière synthétique et dont la connaissance est importante pour tous les informaticiens.

Un ordinateur est une machine électronique programmable capable d'effectuer des traitements automatiques de données et d'interagir avec son environement. Plus précisément, un ordinateur réalise quatre fonctions de base :

- *transfert* des informations depuis et vers son environnement,
- *mémorisation* des informations (données, résultats et programmes),
- *traitement* des données (calculs arithmétiques et logiques),
- *commande et contrôle* des fonctions précédentes à partir des ordres de l'utilisateur, spécifiés au sein des programmes.

Cette vue fonctionnelle de l'ordinateur est résumée par la figure suivante :



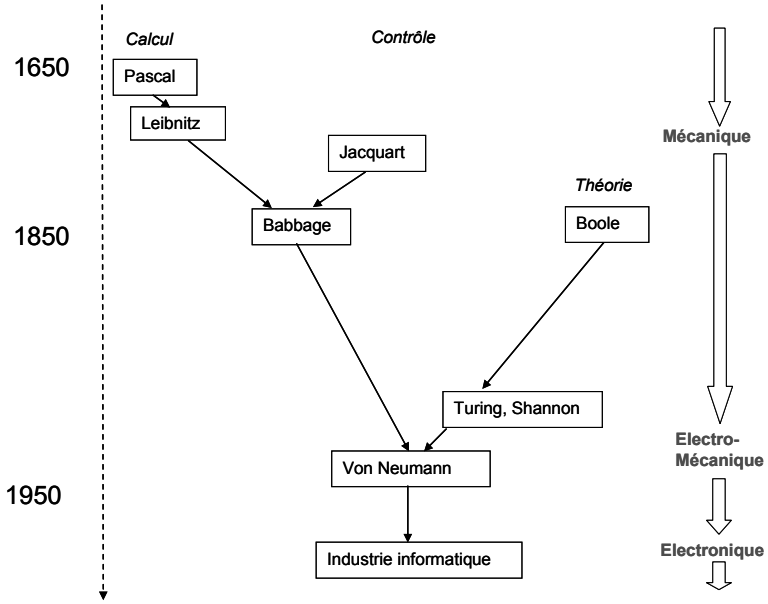
1.2 Éléments historiques

L'ordinateur est l'aboutissement d'une longue histoire. L'informatique est née de la convergence de plusieurs démarches indépendantes, parfois très anciennes :

1. la mécanisation du calcul arithmétique, avec par exemple :
   - la Pascaline de Blaise Pascal (1643), capable d'effectuer additions et soustractions,
   - la machine de Leibnitz (1673) qui ajoute la multiplication, la division et le calcul des racines carrées,
2. la construction de machines (automates) commandées par des « programmes », avec par exemple les métiers à tisser programmables de Falcon et de Jacquard (1804) dont les programmes sont enregistrés sur des cartons perforés,
3. la logique mathématique, avec par exemple :
   - Boole (1854) qui formalise une algèbre avec deux éléments (vrai et faux) et trois opérations (et, ou, non),
   - Shannon (1938) qui lie l'algèbre de Boole, les nombres binaires et les signaux électriques,
   - Turing (1936) qui définit les fonctions calculables par une machine théorique et jette les bases de l'algorithmique moderne.

La convergence du calcul mécanique et de la commande par programme remonte à Charles Babbage, vers 1830. Sa « machine analytique » contient déjà l'essentiel des concepts qui se retrouvent dans les ordinateurs modernes : une unité de calcul (le moulin) commandée par une unité de contrôle, une mémoire (le magasin), une unité d'entrée pour recevoir des cartes perforées (opérations et nombres) et une unité de sortie pour perforer le résultat. Elle n'a pu être construite de son vivant. En effet, c'est l'insuffisance des technologies disponibles (mécaniques puis électro-mécaniques) qui a empêché pendant de nombreuses années la réalisation d'un ordinateur complet.
L'apparition de l'électronique a permis de construire les premiers : l'ENIAC, en 1945, comporte 19000 tubes, pèse 30 tonnes, couvre 72 m^2 et permet 330 multiplications par seconde. La programmation se fait par des fiches à brancher sur un tableau de connections.

En 1948, Von Neumann a proposé de coder le programme en binaire et de le ranger en mémoire. L'architecture de base des ordinateurs, appelée depuis « architecture de Von Neumann », est ainsi acquise.
La même année, le transistor apparaît qui apporte rapidité et fiabilité aux ordinateurs.
L'aventure industrielle peut démarrer (vers 1955), avec en particulier la société IBM.
Le premier langage de programmation évolué, le FORTRAN, est défini par John Backus d'IBM, en 1957.



La révolution suivante est celle des circuits intégrés et des microprocesseurs sur une puce (1971).
Le premier PC (*personal computer*), le Micral de la société Française R2E, apparaît en 1973.
Enfin, les protocoles TCP et IP, à la base d'Internet, sont définis en 1982.

1.3 Organisation du cours

La première partie de ce cours s'intéresse à la fonction de mémorisation des données et plus particulièrement à la manière dont sont représentées les informations dans l'ordinateur (*codification et numérisation*).

La deuxième partie du cours décrit comment sont réalisées les fonctions élémentaires de traitement et de mémorisation, en termes de *circuits logiques*, puis à l'organisation en *composants matériels* de l'ordinateur (mémoires, unité de contrôle, unité arithmétique et logique, unités d'échange, bus, périphériques, ...).

La troisième partie traite de la *programmation en « langage machine »*, c'est à dire dans le langage directement compréhensible par l'unité de commande et reflétant les circuits mis en œuvre dans l'unité arithmétique et logique. Cette étude se fait avec une machine simulée.

2 La représentation des informations

2.1 Le binaire

Le binaire est le système de numération en base 2.
Rappelons que tout système de numération comporte un ensemble B de symboles (ou chiffres) et un ensemble de règles permettant de représenter les éléments d'un ensemble fini ou non de nombres. La taille de l'ensemble B, notée b dans la suite, s'appelle la base du système de numération.
Le binaire comprend donc deux symboles, représentés conventionnellement par 0 et 1.
Le tableau ci-dessous donne les premiers entiers naturels exprimés en binaire :

décimal	0	1	2	3	4	5	6	7	8	9 ...
binaire	0	1	10	11	100	101	110	111	1000	1001 ...

a) Rappel mathématiques sur la notation positionnelle en base b
Dans les systèmes de numération classiques, appelés systèmes de numération pondérés, chaque chiffre a un poids qui dépend de sa position dans le nombre.
Plus précisément, un nombre N est représenté par une suite de chiffres $c_n, c_{n-1}, c_{n-2}, ..., c_1, c_0$ dont la valeur est comprise entre 0 et $b-1$. Le chiffre de rang i a un poids égal à b^i :

$$N = c_n \cdot b^n + c_{n-1} \cdot b^{n-1} + ... + c_1 \cdot b^1 + c_0 \cdot b^0$$

Exemples :
En base 10 : $5908 = 5 \cdot 10^3 + 9 \cdot 10^2 + 0 \cdot 10^1 + 8 \cdot 10^0$
En base 2 : $11001 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 16 + 8 + 0 + 0 + 1 = 25$

Dans le contexte du binaire, il est donc important de bien maîtriser la table des puissances de 2. Cette table est donnée à l'annexe A.

b) Pourquoi le binaire ?
C'est une forme de représentation facile à transmettre (ex : présence/absence de courant électrique dans un fil) et facile à mémoriser par tout dispositif à 2 états stables dont on peut commander le changement d'état (ex : surface aimantée ou non aimantée).

c) Des opérations très simples
Les tables d'addition et de multiplication sont très simples.

+	0	1
0	0	1
1	1	0(1)

×	0	1
0	0	0
1	0	1

Exemple d'addition et de multiplication :

```
   1
  1001
+ 101
-----
 1110
```

```
    1011 (multiplicande)
  x 1001 (multiplicateur)
  ------
    1011
   0000
  0000
 1011
 -------
 1100011
```

On remarque qu'une multiplication se ramène à une suite de recopiages du multiplicande, de décalages et d'additions. La soustraction peut se ramener à l'addition d'un nombre négatif dont nous verrons dans la suite quelle est sa représentation. La division peut se ramener à des soustractions itérées tant

que le résultat est supérieur ou égal au diviseur.

d) Conversion entre base 2 et base 10
Elle découle directement de la définition du binaire.
Exemple : 101001 donne $1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 32 + 8 + 1 = 41$.

e) Conversion entre base 10 et base 2
On peut utiliser la méthode des « restes successifs » qui consiste à diviser le nombre par la base autant de fois que possible, c'est-à-dire tant que le quotient reste positif. Le résultat est constitué des restes des divisions lus du dernier au premier.
Exemple :
41 divisé par 2 donne 20 reste 1 ; c'est le chiffre des unités, à droite du nombre ;
20 divisé par 2 donne 10 reste 0 ;
10 divisé par 2 donne 5 reste 0 ;
5 divisé par 2 donne 2 reste 1 ;
2 divisé par 2 donne 1 reste 0 ;
1 divisé par 2 donne 0 reste 1 ; c'est le chiffre de la plus forte puissance de 2, à gauche du nombre ;
D'où le nombre binaire : 101001

Exercices

1. Additionnez 101101 et 11001.

2. Multipliez 101 et 110.
Comment peut-on facilement multiplier un nombre binaire par 2, 4, 8 ?

3. Vérifiez les opérations précédentes en convertissant en décimal données et résultats.

4. Convertissez 37 en binaire.

2.2 Quelques éléments de vocabulaire

Bit = *binary digit* en anglais = symbole binaire (0 ou 1).
Octet = *byte* en anglais = 8 bits (ex : 01001111).
Mot = nombre entier d'octets, manipulable comme un tout par un ordinateur (pour les fonctions de calcul, de mémorisation, de transfert). Les plus courants sont les mots de 32 bits et les mots de 64 bits.

Unités de capacité (par exemple, taille d'une mémoire en nombre de bits) :
- kilo (K) = $2^{10} = 1024 \approx 1000$,
- méga (M) = $2^{20} \approx 1000000$,
- giga (G) = $2^{30} \approx 1000000000$,
- téra (T) = $2^{40} \approx 1000000000000$,
- péta (P) = $2^{50} \approx 1000000000000000$.

Le bit le plus à gauche d'un nombre binaire est dit bit de « poids fort » (ou MSB pour *Most Significant Bit*). Le bit le plus à droite d'un nombre binaire est dit bit de « poids faible » (ou LSB pour *Less Significant Bit*).

Exercices

5. Quel est le plus grand entier binaire qui tient sur un octet ? sur un mot de 16 bits ? sur un mot de 32 bits ? sur n bits ?

6. Algorithme de la multiplication de deux entiers en binaire.
Le multiplicateur est rangé dans le mot MCAT. Le multiplicande est rangé dans le double mot MCANDE. Le résultat est rangé dans le double mot RES.
Montrez sur un exemple que l'algorithme suivant réalise bien la multiplication :

1. mettre RES à 0,
2. si MCAT est impair (bit de poids faible à 1), additionner MCANDE et RES dans RES,
3. décaler MCAT de 1 bit vers la droite et MCANDE de 1 bit vers la gauche, les bits qui sortent du mot étant perdus,
4. si MCAT vaut 0, arrêter, sinon recommencer en 2.

On pourra par exemple calculer 11×9 en utilisant le tableau suivant avec des mots de 4 bits :

MCAT	MCANDE	RES
1001	00001011	00000000
...	...	...

2.3 Représentation des entiers naturels ($\mathbb{N}$)

Ce sont les entiers positifs et le zéro.

a) En binaire « pur »
Il s'agit du mode de représentation habituel des entiers naturels. Il existe un danger de débordement, dû à la taille limitée des mots utilisés.
Exemple : sur 8 bits, il y a débordement à partir de 255.

La plupart des processeurs (ceux d'Intel en particulier) se contentent d'indiquer le débordement dans un bit d'un registre spécial qui peut être testé par le programme. Quelques rares processeurs déclenchent une exception qui entraîne l'arrêt du programme. C'est donc au programmeur de faire attention aux débordements possibles !

Selon les processeurs, les octets des entiers sont rangés en mémoire de manière classique c'est à dire par poids décroissants (processeurs *big-endian* ou « gros-boutistes », comme les processeurs Motorola ou Sparc) ou de manière plus surprenante par poids croissants (processeurs *little-endian* ou « petits-boutistes », comme les processeurs Intel).

b) En décimal codé binaire (DCB ou *BCD* en anglais)
Chaque chiffre décimal est codé en binaire sur 4 bits, en conservant la structure du nombre décimal.
Exemple : 1428 donne 0001 0100 0010 1000, soit 1 en binaire sur 4 bits, suivi de 4 en binaire sur 4 bits, suivi de 2 en binaire sur 4 bits et de 8 en binaire sur 4 bits.

Ce type de représentation consomme beaucoup de place mémoire. Son utilisation est aujourd'hui peu fréquente.

2.4 Représentation octale et hexadécimale des nombres binaires

Un nombre en binaire est long et difficile à lire par un humain. On préfère souvent le représenter en octal ou en hexadécimal, ce qui facilite la communication entre humains.
Attention ! il s'agit d'une représentation externe pour l'homme et non d'une représentation interne en machine.

a) Octal C'est le système de numération en base 8. Il utilise huit chiffres : 0, 1, 2, 3, 4, 5, 6 et 7. Un chiffre octal correspond à trois chiffres binaires (car $8 = 2^3$). Pour passer du binaire à l'octal il suffit donc de découper des tranches de trois bits. Pour passer de l'octal au binaire, chaque chiffre

octal génère une tranche de trois bits.
Exemples : 1 101 101 010 donne 1552 en octal. 632 en octal, noté $632_{(8)}$, donne 110 011 010.

b) Hexadécimal C'est le système de numération en base 16. Il utilise seize chiffres : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E et F. Un chiffre hexadécimal correspond à quatre chiffres binaires (car $16 = 2^4$). Pour passer du binaire à l'hexadécimal il suffit donc de découper des tranches de quatre bits. Pour passer de l'hexadécimal au binaire, chaque chiffre hexadécimal génère une tranche de quatre bits.
Exemples : 11 0110 1010 donne 36A en hexadécimal. $1A2_{(16)}$ donne 0001 1010 0010.

Exercices

7. Convertissez 10100011101 en hexadécimal, en décimal.

8. Que vaut FFF en binaire, en décimal ?

9. Convertissez 1100110 en octal, en décimal.

2.5 Représentation des entiers relatifs ($\mathbb{Z}$)

Ce sont les entiers positifs, négatifs et le zéro. Il existe trois méthodes possibles de représentation des entiers relatifs : avec bit de signe, en complément à 2 et par excédent.

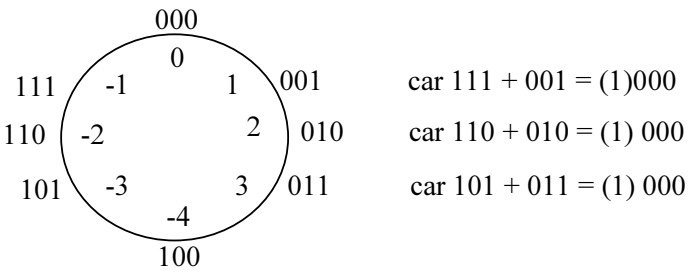
a) Représentation avec bit de signe
Le bit de gauche est réservé pour le signe : 0 pour le signe + (afin que les positifs soient identiques aux entiers naturels) et 1 pour le signe −.

Avec cette représentation :
- il existe deux zéros différents (+0 et −0),
- l'addition d'un positif et d'un négatif ne peut pas se faire par une simple addition des bits : un nombre positif additionné à un nombre négatif donnerait toujours un nombre négatif ! Pour déterminer le signe du résultat il faut savoir quelle est la plus grande valeur absolue. Les circuits de calcul sont donc complexes.

Pour ces raisons, cette forme de représentation n'est plus utilisée.

b) Représentation en complément à 2 sur n bits
On souhaite une représentation telle que :
- les positifs soient identiques aux entiers naturels,
- l'addition des bits d'un nombre u et de son opposé $-u$ donne toujours 0,
- il y ait à peu près autant de positifs que de négatifs.

La meilleure solution consiste à choisir la correspondance binaire–entiers suivante, montrée sur le cercle ci-dessous pour $n = 3$, qui vérifie les trois propriétés souhaitées. Les entiers relatifs sont à l'intérieur du cercle et leurs représentations en binaire sont à l'extérieur du cercle. Les nombres dont le bit de poids fort est à 0 sont utilisés pour représenter les positifs selon la numération binaire. Les nombres dont le bit de poids fort est à 1 sont utilisés pour représenter les négatifs en choisissant un certain ordre qui permet de vérifier les propriétés ci-dessus.



On vérifie que l'addition de u et de $-u$ donne 2^n c'est à dire 0 quand on ignore le bit de retenue qui « sort » à gauche (on peut aussi dire que $u + -u = 0$ modulo 2^n). La valeur médiane (100) est attribuée à -4 plutôt qu'à 4 afin que tous les négatifs commencent par 1 et tous les positifs commencent par 0. On retrouve ainsi un bit de signe !

Plus généralement, on peut dire que le complément à 2 sur n bits d'un nombre binaire A, noté $c2(A)$, est tel que $A + c2(A) = 2^n$ c'est à dire un 1 suivi de n zéros.
Donc : $c2(A) = 2^n - A = (2^n - 1) - A + 1$.
$2^n - 1$ est une suite de n chiffres 1. $(2^n - 1) - A$ revient à soustraire chaque bit de A à un 1. Si ce bit vaut 0 le résultat est 1. Si ce bit vaut 1 le résultat est 0. Autrement dit on change les 0 en 1 et les 1 en 0 dans le nombre A. C'est ce qu'on appelle le « complément à 1 ». Reste finalement à ajouter 1 à ce complément à 1 pour obtenir le complément à 2 (cf. exemple ci-dessous).

D'où la règle pratique pour obtenir le complément à 2 d'un nombre binaire : « **on remplace les 0 par des 1, les 1 par des 0 et on ajoute 1. On ne tient jamais compte du bit de retenue qui déborde à gauche des n bits** (ou *carry bit*) ».
Exemple : ($n = 8$)
2^8 = 1 0000 0000 (1 plus 8 zéros),
$2^8 - 1$ = 1111 1111 (des 1 partout),
soit A = 0000 1011 (c'est à dire 11 en décimal),
$(2^8 - 1) - A$ = 1111 0100 (changement des 0 en 1 et des 1 en 0),
$(2^8 - 1) - A + 1$ = 1111 0101 (c'est le complément à 2 de 11, représentant -11).
On vérifie que $A + c2(A)$ = 0000 1011 + 1111 0101 = (1) 0000 0000.

On note également que si on recommence la même transformation on retrouve le nombre positif : $c2(c2(A)) = c2(2^n - A) = 2^n - (2^n - A) = A$.
Exemple :

```
1111 0101   (-11)
0000 1010   (modification des bits)
+       1   (ajout de 1)
---------
0000 1011   (on retrouve 11).
```

Cette représentation a les avantages suivants :

- on a un seul 0, codé par 0000 0000 ; en effet : $c2(0) = 2^n - 0 = 2^n$ qui « sort » à gauche.

```
     0000  0000
     1111  1111 (modification des bits)
     +        1 (ajout de 1)
     ------------
  (1)0000 0000  (on ne tient pas compte du débordement à gauche donc -0 = +0)
```

- le premier bit est aussi un bit de signe : les nombres strictement négatifs commencent toujours par 1 (en effet en enlevant à 2^n un nombre compris entre 1 et 2^{n-1} on obtient un nombre entre $2^n - 1$ et 2^{n-1} ; ces nombres commencent tous par 1 en binaire sur n bits) ;

– **on peut additionner un nombre positif et un nombre négatif sans se préoccuper du signe ; le résultat est toujours correct**. Considérons par exemple l'addition binaire de x et y avec y < 0. On sait que y est représenté par c2(y)= 2^n - y. Donc x + y = x + 2^n - y = x - y + 2^n. Comme le 2^n est perdu par débordement à gauche le résultat de l'addition est bien égal à x - y.
Exemple : 7 + (-6)

```
      0000 0111  (7)
    + 1111 1010  (-6)
   ------------
   (1)0000 0001  (débordement à gauche perdu et résultat = 1)
```

– l'apparition d'un bit qui sort à gauche n'est pas nécessairement une erreur en complément à 2 ; les calculs incorrects correspondent à un report de l'avant dernier bit dans le dernier bit sans report du dernier bit en dehors du mot ou bien à un report en dehors du mot sans report de l'avant dernier sur le dernier bit (cf. exercice 13).
Exemples : deux exemples de calculs incorrects

```
 0 1                                    1 0      1
   0100 0000  (64)                        1000 0001  (-127)
 + 0100 0001  (65)                      + 1000 0001  (-127)
------------                            -----------
(0)1000 0001  (-127 !)                 (1)0000 0010  (2 !)
```

Le complément à 2 est utilisé dans tous les ordinateurs pour représenter les entiers relatifs. Il existe cependant un cas particulier où une autre représentation est employée. Nous la présentons au paragraphe suivant.

c) Représentation par excédent (ou « biaisée »)
Pour représenter l'intervalle $[-i, +j]$, on ajoute i à toutes les valeurs de l'intervalle : 0 représente donc la plus petite valeur négative $(-i)$ et $i + j$ représente la valeur j. On parle « d'excédent à i » et celà correspond à un décalage de i pour tous les nombres.

Sur n bits, on peut par exemple ajouter 2^{n-1} qui est en gros la valeur centrale de toutes les valeurs possibles.
Exemple : sur 8 bits, $2^{8-1} = 2^7 = 128$.
En « excédent à 128 » :
-128 est codé par 0,
-127 est codé par 1,
-126 est codé par 2,
...
-1 est codé par 127,
0 est codé par 128,
1 est codé par 129,
...
126 est codé par 254,
127 est codé par 255.
On peut également définir un « excédent à 127 » pour l'intervalle $[-127, +128]$.

Comme nous le verrons dans la suite, cette représentation en excédent est utilisée pour coder les exposants des nombres réels .

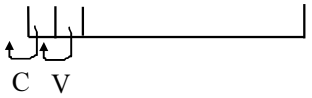
Exercices

10. Calculez $30 + -20$, -20 étant codé en complément à 2 sur 8 bits.

11. Donnez en base 10 les nombres dont le codage en complément à 2 sur 16 bits sont les suivants : 0110 1100 0001 1011 et 1011 0110 1011 0011.

12. Comparez les trois systèmes de représentation des relatifs sur 4 bits en dressant un tableau de toutes les valeurs possibles avec les trois systèmes.

13. Calculez les six opérations ci-dessous, caractéristiques de tous les cas possibles d'additions dans $\mathbb{Z}$, en complément à 2 sur 8 bits.
Positif + positif sans débordement : $6 + 8$.
Positif + positif avec débordement : $127 + 1$.
Positif + négatif, résultat positif : $4 + (-2)$.
Positif + négatif, résultat négatif : $2 + (-4)$.
Négatif + négatif sans débordement : $(-2) + (-4)$.
Négatif + négatif avec débordement : $(-127) + (-2)$.
On note C le bit de *carry* (c'est-à-dire le bit « qui sort ») et V le bit d'*overflow* (c'est à dire le bit qui se reporte de l'avant dernier sur le dernier bit). Au vu des six résultats obtenus, quelle condition sur C et V permet de détecter qu'une addition est correcte ?



14. Multiplication de deux entiers relatifs en complément à 2.
Comment faut-il procéder en supposant que l'on dispose d'un moyen pour multiplier les entiers naturels ? Vous considérerez les trois cas possibles : deux positifs, un positif et un négatif, deux négatifs.

2.6 Représentation des réels ($\mathbb{R}$)

Ce sont tous les nombres « à virgule ». C'est à dire les rationels, qui peuvent s'écrire comme des fractions, et les irrationels, comme $\sqrt{2}$ ou π.

a) Rappels sur la « notation scientifique »

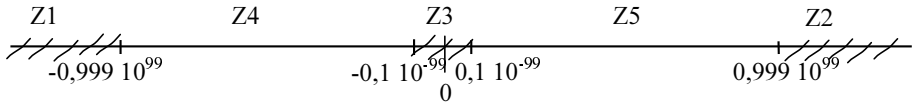
Un nombre réel peut s'écrire avec une mantisse m et un exposant e : $nb = \pm m \cdot 10^{\pm e}$.
Exemples : $123000 = 1,23 \cdot 10^5$; $0,0071 = 7,1 \cdot 10^{-3}$.
L'écriture d'un nombre réel en notation scientifique n'est pas unique.
Exemples : $123 \cdot 10^3$ représente également 123000 ; $0,71 \cdot 10^{-2}$ représente également 0,0071.
On peut fixer une règle qui impose l'unicité de la représentation, ou « normalisation ». Une règle de normalisation possible peut stipuler que la mantisse doit commencer par $0,nccc\ldots$ où n est un chiffre non nul et c un chiffre quelconque.
Exemples : avec cette règle, les valeurs normalisées de 123000 et 0,0071 sont $0,123 \cdot 10^6$ et $0,71 \cdot 10^{-2}$.

b) Propriétés de la « notation scientifique »

Elle comporte un nombre fini de valeurs, alors qu'il y a une infinité de réels. On travaille donc avec des valeurs approchées. Leur densité varie selon la proximité du 0 : plus on s'éloigne de 0 et plus la distance entre les nombres représentés s'accroît.
Exemple : avec une mantisse à 4 chiffres normalisée en 0,ncc et un exposant à 2 chiffres, on peut représenter $900 \cdot 199 \cdot 2 = 358200$ nombres. Des très petits et des très grands !

Dans les zones Z4 et Z5 de la figure ci-après les nombres sont très proches les uns des autres (forte densité) à proximité de 0 et très éloignés les uns des autres (faible densité) du côté des grands nombres. Par exemple, entre $0,998 \cdot 10^0$ et $0,999 \cdot 10^0$ l'écart entre 2 nombres consécutifs est de $0,001$. Au contraire, entre $0,998 \cdot 10^{20}$ et $0,999 \cdot 10^{20}$ l'écart entre 2 nombres consécutifs est de 10^{17}, soit

10000000000000000 !
Dans la zone Z3 les nombres sont trop proches de 0 pour être distingués de 0 (*underflow*).
Dans les zones Z1 et Z2 les nombres sont trop grands pour être représentés (*overflow*).

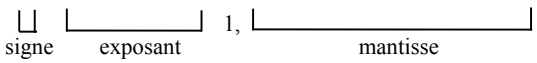


La multiplication des réels en notation scientifique ne pose pas de problème : il faut additionner les exposants et multiplier les mantisses, puis normaliser. Par contre, l'addition exige une dénormalisation pour avoir les mêmes exposants avant addition, puis une re-normalisation après addition.

c) Représentation de la partie fractionnaire en binaire
Il suffit de généraliser la notation positionnelle aux puissances négatives. Après la virgule les valeurs des chiffres binaires correspondent aux puissance négatives de 2 :
$2^{-1} = 1/2 = 0,5$
$2^{-2} = 1/4 = 0,25$
$2^{-3} = 1/8 = 0,125$ ...
Exemple : 101,101 donne $1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 4 + 1 + 0,5 + 0,125 = 5,625$.

Pour trouver la représentation binaire d'une partie fractionnaire on fait une suite de multiplications par 2 en retirant à chaque fois le chiffre avant la virgule jusqu'à obtenir 0. Cette suite de chiffres retirés constitue la partie fractionnaire en binaire.
Exemple :
$0,375 \cdot 2 = 0,750$; on retire 0 ; on obtient 0,750
$0,750 \cdot 2 = 1,5$; on retire 1 ; on obtient 0,5
$0,5 \cdot 2 = 1$; on retire 1 ; on obtient 0 ; donc 0,375 donne 0,011 en binaire.

d) Représentation des réels en virgule flottante
La norme IEEE 754 (*Institute of Electrical and Electronics Engineers*) est une norme quasi universelle pour la représentation des réels.
Pour les mots de 32 bits (« réel en simple précision ») :
- le signe est représenté dans le premier bit (0 pour +, 1 pour −),
- l'exposant est représenté sur 8 bits en excédent à 127,
- la mantisse est normalisée sous la forme $1,bbb$. Le 1, n'est pas représenté (on parle de « bit caché » ou *hidden bit*). La suite de la mantisse est représentée sur 23 bits.



Le zéro est représenté par 32 bits à 0. Par convention :
- quand l'exposant est rempli de 1 et la mantisse est à zéro, il s'agit de l'infini,
- quand l'exposant est rempli de 1 et la mantisse est différente de 0, il s'agit d'un code d'erreur (un *NaN* pour *Not a Number*).

Exemple : 1,5 en décimal donne 1,1 en binaire soit $+1,1 \cdot 2^0$.
L'exposant vaut 127 en excédent à 127. Le nombre réel donne donc :
0 01111111 (1,) 1000... soit 3FC00000 en hexadécimal.

Sur 64 bits (« réel en double précision »), l'exposant est en excédent à 1023 sur 11 bits et la mantisse occupe 52 bits. Il existe également une « précision étendue » sur 80 bits.

Comme la taille de la mantisse est finie, on peut avoir des erreurs d'arrondi. La norme IEEE 754 définit quatre modes de calcul pour l'arrondi : au plus proche (mode par défaut), vers moins l'infini,

vers plus l'infini et par troncature.

Exercices

15. Donnez la représentation hexadécimale de -12,625 en flottant IEEE 754 sur 32 bits.

16. Quelle est la valeur décimale du réel flottant IEEE 754 : 42E4 8000 ?

17. Quelle est la valeur décimale du réel flottant IEEE 754 : 3E94 0000 ?

18. Représentez 7/8 en format IEEE 754 sur 32 bits. Donnez le résultat en hexadécimal.

19. On appelle précision la différence entre 1 et le plus petit nombre qui le suit. Que vaut-elle en simple précision ? Exprimée comme une puissance de 10 ? Cela correspond à combien de chiffres significatifs après la virgule ?

20. Addition de deux réels.
Pour faire une addition, il faut ramener les deux opérandes au même exposant en dénormalisant celui qui a l'exposant le plus petit. Ce faisant, on doit décaler sa mantisse vers la droite avec le risque de perdre des chiffres. Le résultat peut donc être entaché d'erreur. Cette erreur est d'autant plus grande que la différence entre les exposants est plus grande.
On suppose que la mantisse est représentée sur 6 chiffres binaires.
On veut additionner $1.001100\ 2^4$ et $1.011011\ 2^2$.
Quelle est l'erreur commise ?

Remarque : une applet pour réaliser des conversions du décimal vers l'IEEE 754 et réciproquement est disponible sur `www.loria.fr/~jloncham` rubrique `Enseignement`.

e) Représentation des réels en « virgule fixe »
La virgule flottante convient parfaitement aux applications de calcul scientifique dans lesquelles la place de la virgule peut varier énormément. Au contraire, dans les applications de gestion la virgule est souvent placée à une position donnée. En comptabilité, par exemple, les nombres ont 2 chiffres après la virgule pour les montants en euros et centimes d'euros. On peut utiliser dans ces cas une représentation en « virgule fixe ».

Dans une représentation en virgule fixe, la virgule n'est pas représentée dans le nombre. Sa position est soit fixée à un certain emplacement, soit définie par le programmeur au moment de l'utilisation de la valeur (cf. la clause V, comme virgule, dans les réels en COBOL définis par des déclarations telles que `SOMME PIC S9999V99`).

2.7 Les caractères alphanumériques

a) Le code ASCII
Dans cette norme (*American Standard Code for Information Interchange*) chaque caractère est codé sur sept bits. On peut donc représenter 128 caractères différents ce qui suffit pour la langue anglaise dépourvue d'accents. Les codes sont en général stockés et transmis sur un octet. Pour un caractère stocké en mémoire le 8^e bit est en général à 0. Lors des transmissions, ce 8^e bit peut être utilisé pour faire un contrôle de parité (cf. exercices récapitulatifs).

On peut souligner plusieurs points importants à propos du code ASCII :
- Les codes compris entre 0 et 31 ne représentent pas des caractères affichables. Ces codes, souvent nommés « caractères de contrôle » sont utilisés pour indiquer des actions aux terminaux infor-

matiques. Par exemple, dans un fichier texte, la fin d'une ligne est représentée par un caractère de contrôle (ou une paire de caractères de contrôle). Plusieurs conventions coexistent : sous les systèmes Unix ou « type Unix » (Linux, AIX, Mac OS X, etc.), la fin de ligne est indiquée par un saut de ligne (LF) ; sous Mac OS jusqu'à la version 9, la fin de ligne est indiquée par un retour chariot (CR) ; sous les systèmes Microsoft Windows, la fin de ligne est indiquée par un retour chariot suivi d'un saut de ligne (CR+LF, 2 octets). Ainsi, quand on ouvre un fichier ASCII créé par un système sous un autre système, il faut souvent modifier les fins de ligne afin de pouvoir l'afficher de manière correcte. Les éditeurs de texte évolués (Wordpad mais pas Notepad sous Windows) peuvent détecter le type de fin de ligne et agir en conséquence.
- Le code 32 correspond à l'espace.
- Les lettres se suivent dans l'ordre alphabétique (codes 65 à 90 pour les majuscules, 97 à 122 pour les minuscules), ce qui simplifie les comparaisons.
- Les chiffres sont rangés dans l'ordre croissant (codes 48 à 57), et les 4 bits de poids faibles définissent la valeur en binaire du chiffre.

La table du code ASCII est donnée à l'annexe B1.

b) Le code code ISO-8859-1 (ou « ISO Latin-1 » ou « ASCII étendu »)
Dans cette norme de l'ISO (*International Standard Organisation*), le code ASCII précédent est étendu en ajoutant de nombreux caractères, en particulier accentués, qui sont utilisés par les langues européennes courantes : allemand, anglais, espagnol, français (sauf caractères Ÿ, œ et Œ), italien, néerlandais, norvégien, portugais, suédois, etc. L'afrikaans et le swahili sont également couverts. Cette norme est donc utilisée en Europe de l'Ouest, en Amérique, en Australie et dans une grande partie de l'Afrique. Elle contient 256 caractères correspondants à tous les codes possibles sur 8 bits. Les 128 premiers caractères sont ceux du code ASCII sur 7 bits.
ISO-8859-1 est le codage standard utilisé par le système X Window sur la plupart des machines UNIX.

La limitation principale du codage sur un octet est la nécessité d'utiliser plusieurs tables de code (les normes ISO-8859-x avec x variant de 1 à 10) pour couvrir plusieurs alphabets. Or certains documents sont typés à l'aide de méta-données. Dans ce cas, une table de caractères et une seule est associée à chaque document. Par exemple pour un document HTML la balise

```
<meta http-equiv="content-Type" content="text/html; charset=iso-8859-1" />
```

indique que le document utilise la table des caractères ISO-8859-1. De même en XML la balise

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

indique l'encodage des caractères utilisé. Par défaut, c'est la table de caractères du système d'exploitation qui est utilisée. Si elle ne correspond pas à celle du document l'affichage devient incohérent. De plus, il est impossible de mélanger dans un même document des alphabets qui ne sont pas définis par la même table, comme par exemple le français et l'hébreu. La seule solution est de passer à Unicode qui est présenté au paragraphe suivant.

La norme ISO-8859-15 a ajouté le symbole de l'euro et quelques caractères comme Ÿ, œ et Œ à la norme ISO-8859-1.

Windows-1252, disponible sur les systèmes Windows, diffère de l'ISO-8859-1 par l'utilisation de caractères imprimables (comme le symbole de l'euro, œ et Œ), plutôt que des caractères de contrôle, dans les codes 128 à 159.

La table du code ISO-8859-1 est donnée à l'annexe B2.

c) Unicode

Unicode est une norme développée par le Consortium Unicode qui vise à donner à tout caractère de toute langue un nom et un identifiant numérique quelle que soit la plate-forme informatique ou le logiciel. Unicode, dont la première publication remonte à 1991, a été développé dans le but de remplacer l'utilisation des pages de code. La version actuelle 5.1 définit plus de 120000 caractères.

Unicode est défini suivant un modèle en couches. Les autres normes ne faisaient typiquement pas de distinction entre le jeu de caractères et la représentation physique. Les principales couches sont présentées ici en partant de la plus éloignée de la machine :

- Répertoire des caractères abstraits (*Abstract Character Repertoire*)
  La couche la plus élevée est la définition du jeu de caractères. Unicode leur donne des noms. Par exemple, le caractère Ç est nommé « Lettre majuscule latine c cédille ».
- Jeu de caractères codés (*Coded Character Set*)
  Ici, on ajoute à la table précédente un index numérique. Il ne s'agit pas d'une représentation en mémoire, juste d'un nombre. Ce nombre, appelé « point de code », est noté `U+xxxx` où `xxxx` est en hexadécimal et comporte 4, 5 ou 6 chiffres. Ainsi, le caractère nommé « Lettre majuscule latine c cédille » a comme point de code `U+00C7`.
- Formalisme de codage des caractères (*Character Encoding Form*)
  On arrive à la représentation physique (en mémoire, sur disque . . .) : cette couche spécifie quelles unités de stockage (*code units*), octets ou bien mots de 16 ou de 32 bits, vont représenter un point de code. On ne peut donc pas parler de la taille d'un caractère Unicode car elle dépend du codage choisi.
  Unicode accepte plusieurs types de transformations universelles (*Universal Transformation Format* ou UTF) pour représenter un point de code valide : UTF-8, UTF-16, UTF-32. Le nombre après UTF représente le nombre *minimal* de bits avec lesquels un point de code valide est représenté.
  - L'UTF-8 est le plus commun pour les applications Unix et Internet. Son codage de taille variable lui permet d'être en moyenne moins coûteux en occupation mémoire (un octet pour les points de code assignés aux caractères ASCII, 2 à 4 octets pour les autres points de code).
  - L'UTF-16 est un bon compromis lorsque la place mémoire n'est pas trop restreinte, car la grande majorité des caractères des langues modernes peuvent être représentés sur 16 bits. C'est notamment le codage qu'utilise la plate-forme Java en interne, ainsi que Windows pour ses APIs compatibles Unicode (avec le type WCHAR). Les autres caractères sont représentés par deux « seizets ».
  - L'UTF-32 est utilisé lorsque la place mémoire n'est pas un problème. L'avantage de cette transformation est que tous les codes ont la même taille (32 bits).

Exercices

21. Décodez la séquence de caractères ASCII sur 8 bits suivante :
01001001 00100000 01001100 01001111 01010110
01000101 00100000 01011001 01001111 01010101

22. Comment transformer le code ASCII sur 8 bits d'une minuscule en le code de la même lettre en majuscule ?

d) Autres codes

Les codes étudiés dans ce paragraphe sont suffisants pour la programmation. Mais il en existe de nombreux autres. Lorsque des données sont transmises sur des réseaux informatiques, des informations redondantes sont envoyées afin de détecter voire même de corriger des altérations de bits pendant la transmission, toujours possibles en raison des perturbations électriques. Ces codes particuliers seront vus dans le cours réseau et un exemple simple est donné dans les exercices récapitulatifs à la fin de cette partie sur la codification.

2.8 Les types en Java et leur représentation en mémoire

Dans les langages de programmation évolués, différents types de données sont définis. A chaque type est associé une certaine représentation mémoire. Par exemple, Java offre huit types de base (ou types primitifs) dont les caractéristiques sont résumées dans le tableau suivant :

Mot clé	**Description**	**Format**	**Valeurs**
byte	octet (entier très court)	complément à 2 sur 8 bits	-128 à 127
short	entier court	complément à 2 sur 16 bits	-32768 à 32767
int	entier	complément à 2 sur 32 bits	-2147483648 à 2147483647
long	entier long	complément à 2 sur 64 bits	$\pm 9,22 \cdot 10^{18}$
float	réel en virgule flottante simple précision	IEEE 754 sur 32 bits	$\pm 3,4 \cdot 10^{-38}$ à $3,4 \cdot 10^{38}$ (précision 7 chiffres)
double	réel en virgule flottante double précision	IEEE 754 sur 64 bits	$\pm 1,7 \cdot 10^{-308}$ à $1,7 \cdot 10^{308}$ (précision 15 chiffres)
char	caractère	Unicode sur 16 bits	65536 caractères
boolean	booléen		vrai, faux

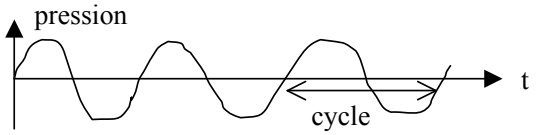
Remarque : le stockage effectif en mémoire dépend de la machine virtuelle Java.

2.9 Représentation en mémoire des autres objets : sons, images, vidéo

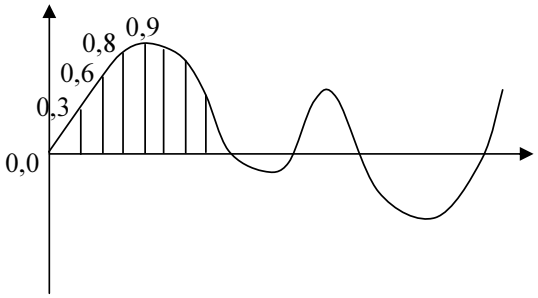
Il existe aussi des « codes » pour représenter numériquement les sons, les images, la vidéo dans le cadre des applications multimédia. Cet aspect est survolé dans ce paragraphe.
Comme toutes les informations en mémoire, ces objets doivent être codés par des bits ; on dit plutôt qu'ils sont « numérisés ». Il existe de très nombreuses représentations pour ces objets. On se limitera ici à quelque exemples simples.

2.9.1 Les sons

Un son est une vibration de l'air, c'est à dire une variation de la pression autour de la pression atmosphérique normale. Par exemple, la vibration de la note LA est une sinusoïde régulière avec 440 cycles par seconde, c'est à dire une fréquence de 440Hertz.



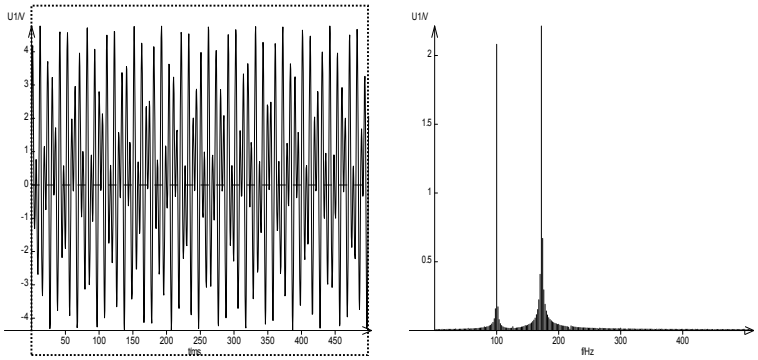
Pour numériser un son quelconque, l'ordinateur peut mesurer la valeur prise par cette courbe à des intervalles de temps réguliers.



En qualité CD audio, l'ordinateur prend 44100 mesures par seconde. On dit qu'il « échantillonne à 44100Hertz » (ou 44,1kHz).

Dans un fichier *wav* (format Microsoft), chaque mesure est codée en binaire sur 16 bits (65535 valeurs possibles) en mono et sur 2×16 bits en stéréo. Une minute de son en qualité CD audio stéréo occupera donc $(44100 \times 60 \times 16 \times 2) / (8 \times 1024 \times 1024) \approx 10$Mo.
Tous les sons (paroles, bruits, musiques) sont des vibrations complexes que l'on peut échantillonner ainsi. Mais cela génère un grand nombre de données par seconde.

Une autre approche, consiste à décomposer la vibration complexe en une somme de vibrations régulières (sinusoïdes).



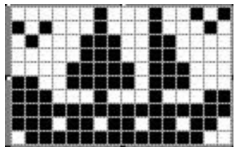
La vibration complexe Les fréquences trouvées

Pour chacune de ces vibrations régulières, comme le LA ci-dessus, il suffit d'enregistrer sa fréquence (nombre de vibrations par seconde exprimée en Hz ; par exemple 440Hz pour le LA).
La transformation mathématique qui permet de trouver les fréquences constituant un signal s'appelle la transformée de Fourier.
C'est sur ce principe qu'est basé le format MP3 (MPEG Audio Layer 3) : on enregistre une partie (plus ou moins grande) des fréquences qui composent le signal et on les note dans le fichier MP3. À qualité sensiblement égale, une minute de son occupe environ 1Mo, contre 10Mo en *wav*. Pour rejouer le son, on prend la liste des fréquences, on recrée des signaux des différentes fréquences et on les mélange pour reconstituer le son. Les logiciels qui créent des fichiers MP3 se basent sur un modèle psycho-acoustique pour supprimer certaines fréquences inutiles. En effet, l'oreille humaine et le cerveau ne perçoivent pas certains sons (fréquences très proches, sons faibles couverts par d'autres sons, etc.) Ce modèle psycho-acoustique fait toute la différence entre les différents codeurs/décodeurs (ou « codecs ») MP3.

2.9.2 Les images

Les images peuvent être représentées comme des ensembles de points ou pixels (formats *bitmaps*) ou comme des ensembles d'objets géométriques (formats vectoriels). Nous ne considérons ici que les formats *bitmaps* qui sont utilisés par les périphériques d'affichage et les périphériques de saisie (scanners).

Les images monochromes (noir et blanc) sont faciles à représenter. Chaque bit à 0 code un pixel blanc et chaque bit à 1 code un pixel noir. Les gris peuvent être obtenus visuellement par des mélanges de pixels blancs et de pixels noirs. Une image monochrome en résolution 1024×768 nécessite donc 786432 bits, soit 96Ko. On peut améliorer la qualité en associant à chaque pixel un niveau de gris, codé sur huit bits. La même image prendra alors 770Ko.



La codification des couleurs peut se faire selon deux modes :

1. en couleurs indexées : l'image est une succession d'index qui font référence à une palette de couleurs. Les index sur huit bits permettent de désigner 256 couleurs. La palette est une table stockée en début de fichier et qui fait correspondre à chaque index une couleur décrite par un mélange de trois couleurs primaires. En RGB il s'agit du % de rouge, du % de vert et du % de bleu, chacun sur huit bits. Soit une palette de 256 couleurs prises parmi $256 \times 256 \times 256 =$ 16,7 millions.
   Une image couleur en 1024×768 nécessite donc $(1024 \times 768 \times 8 + 256 \times 24)/(1024 \times 8) \approx$ 770Ko. Si on change de palette, le rendu de l'image change.
2. en couleurs vraies (*true color*) : chaque pixel est décrit par un code couleur, par exemple en RGB. Comme on l'a vu ci-dessus, avec 24 bits on peut coder 16,7 millions de couleurs. Une image couleur en 1024×768 et en 16,7 millions de couleurs nécessite donc $(1024 \times 768 \times 24)/(1024 \times 1024 \times 8) \approx$ 2,25Mo.

Au vu de ces chiffres, on comprend que la vidéo nécessite des techniques de compression des images. Prenons par exemple une petite séquence de 15 minutes en 1024×768 et 16,7 millions de couleurs. Pour obtenir une fluidité correcte de la vidéo, une vitesse de défilement de 25 images/seconde est nécessaire. La séquence sera donc composée de 22500 images qui pèsent chacune 2,25Mo (cf. ci dessus). Soit plus de 50Go à stocker et éventuellement à transférer sur Internet !

Il existe des techniques de compression sans perte de données et des techniques avec perte possible de données. La technique sans perte de données la plus simple est le RLE (*Run Length Encoding*) : quand n octets successifs sont identiques (avec n supérieur à un seuil donné) on les remplace par un marqueur spécial suivi du nombre de répétitions (sur un octet) suivi de la valeur de l'octet répété.
Exemple : seuil = 3, marqueur @
`GGOOOOOOOOOXAAAAAAAAMLLLLLLLLLL` donne `GG@9OX@7AM@10L` soit un passage de 30 à 14 caractères (gain de 53%).
C'est une technique utilisable par exemple avec des images contenant des plages homogènes de noir et de blanc. S'il y a peu de plages homogènes la méthode peut conduire à perdre de la place au lieu d'en gagner !

Le format JPEG exploite des techniques proches de celles vues pour le son en MP3. L'image est décomposée en petits carrés de 8×8 pixels. Chaque pixel est codé d'une certaine manière, par exemple en RGB avec les valeurs de rouge, vert et bleu. La matrice 8×8 pour chaque couleur est considérée comme un signal qui est décomposé en 8×8 fréquences par une variété de la transformation de Fourier, dite « transformée en cosinus discrète ». Les hautes fréquences, auxquelles l'œil humain est peu sensible, sont supprimées. Les valeurs des fréquences restantes sont elles mêmes compressées. Quand on enregistre un fichier JPEG, on peut choisir le taux de compression (de 1% à 99%) inversement proportionnel à la qualité de l'image. En effet, cette qualité correspond à la précision avec laquelle on conserve les résultats de l'analyse fréquentielle. Une image JPEG de faible qualité apparaît fortement « pixélisée » avec les blocs 8×8 presque uniformes et assez visibles. Par contre le volume de l'image est fortement diminué. Une image de bonne qualité est moins pixélisée mais prend plus de place en mémoire.

2.9.3 La vidéo

La première idée qui vient à l'esprit, après s'être intéressé à la compression des images, est d'en appliquer les principes à la vidéo. Le *Motion JPEG* (noté MJPEG ou M-JPEG) consiste à appliquer successivement l'algorithme de compression JPEG aux différentes images d'une séquence vidéo. Etant donné que le M-JPEG code séparément chaque image de la séquence il permet d'accéder aléatoirement à n'importe quelle partie d'une vidéo ce qui est intéressant par exemple pour les studios de montage numérique.

Dans beaucoup de séquences vidéo, de nombreuses zones dans les images sont fixes ou bien changent très peu, comme par exemple les arrière-plan. C'est ce que l'on nomme la « redondance temporelle ».

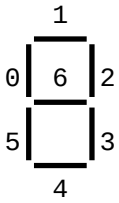
L'idée est de ne décrire que les changements d'une image à l'autre sous forme d'un ensemble de « macro blocs » codés en JPEG. C'est ce que font les standards MPEG vidéo. Il en existe plusieurs :

- MPEG-1 permet le stockage d'une heure de vidéo dans une qualité proche des cassettes VHS sur un support CD de 600Mo,
- MPEG-2 est utilisé par les DVD vidéos (4,7Go pour 2h30 de vidéo de haute qualité),
- MPEG-4 permet le codage et la diffusion de données multimédia sous forme d'objets numériques pour le Web et les périphériques mobiles (par téléchargement ou par *streaming*, c'est à dire transmission et affichage au fur et à mesure).

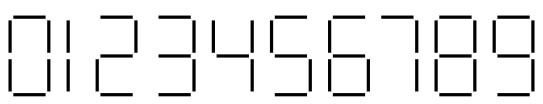
Le codec DivX utilise la norme MPEG-4 et permet à un DVD de tenir sur un simple CD au prix d'une légère dégradation de la qualité. Il facilite bien entendu le transfert (parfois illicite ...) des vidéos sur Internet, comme le MP3 permet le transfert des fichiers audio.

Exercices récapitulatifs (devoirs surveillés des années précédentes)

23. *Chiffres lumineux*
Pour symboliser un chiffre quelconque (0 à 9), on utilise sept segments lumineux numérotés de 0 à 6 et disposés selon le schéma suivant :



La visualisation de chaque chiffre correspond à l'activation d'une partie des segments. Le système adopté est résumé par la figure suivante :



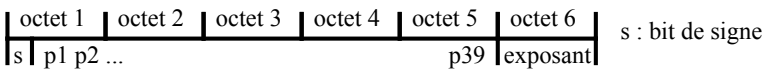
Note : le 1 utilise les barres de gauche.
On désire construire un code sur sept bits où le bit de rang i est positionné à 1 lorsque le segment correspondant est allumé.
a) Construisez la table qui à chaque chiffre associe son code sur 7 bits.
b) Représentez dans ce code la séquence de chiffres : 1987.

24. *Décodage d'une zone mémoire*
Une entreprise gère des informations pour chacun de ses clients. A chaque client est associé une zone mémoire de longueur fixe, organisée de la manière suivante :

- nom du client : la zone est prévue pour un nom de 15 caractères au maximum,
- nombre de commandes effectuées par le client : un entier,
- la somme que le client doit ou que l'entreprise doit au client (somme < 0) : un réel,
- un commentaire général sur le client : la zone est prévue pour un commentaire de 20 caractères au maximum.

L'ordinateur utilisé par l'entreprise code les informations de la manière suivante.

- Une chaîne de n caractères est codée sur $n + 1$ octets où le premier octet représente la taille effective de la chaîne. Les caractères sont codés en ASCII sur 8 bits.
- Les entiers sont codés sur 16 bits, les nombres négatifs étant codés en compléments à deux. L'octet contenant les bits de poids les plus faibles est placé avant l'octet contenant les bits de poids les plus forts.
  Exemple : la valeur 1 est codée en hexadécimal sur deux octets par 01 00.
- Les réels sont codés en virgule flottante sur 48 bits de la manière suivante.
  - Le nombre réel N est normalisé en $N = m \cdot 2^e$ où $0,1 \leq m < 1$.
  - L'exposant e est codé en excédent à 128.
  - Pour la mantisse m, on remarque que dans la forme normalisée $0, p_0p_1p_2...p_{39} \cdot 2^e$ le bit p_0 vaut toujours 1. En conséquence seuls les bits de p_1 à p_{39} sont représentés.
  - Le premier bit s code le signe.



Exemple : avec ce système de codification la représentation en machine du nombre 256,0 est 00 00 00 00 00 89.

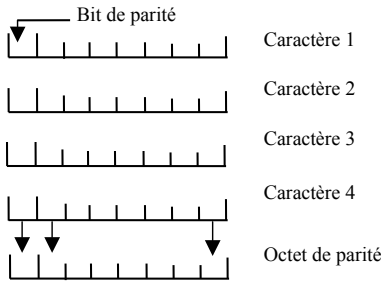
a) Quelle est la taille de la zone mémoire d'un client (en octets) ?
b) Décodez les zones mémoire des deux premiers clients commençant à l'adresse 0000. Les adresses sont en décimal et repèrent des octets.

```
Adresses -------  Codes Hexadécimaux  ------------
0000 07 42 6F 6C 6C 69 6E 6F 25 39 64 20 46 69 63 68
0016 19 00 33 90 00 00 00 8B 10 4A 41 4D 41 49 53 20
0032 43 4F 4E 54 45 4E 54 20 21 20 64 69 73 05 44 75
0048 76 61 6C 6E 6F 25 39 64 20 46 69 63 68 36 00 9E
0064 94 00 00 00 8C 0A 42 4F 4E 20 43 4C 49 45 4E 54
0080 54 45 4E 54 20 21 20 64 69 73 2E 1D 2F 1D 30 1D
0096 31 1D FF FF 33 1D 34 1D 35 1D 36 1D 37 1D 38 1D
0112 39 1D 3A 1D 3B 1D 3C 1D 3D 1D 3E 1D 3F 1D 40 1D
```

25. *Codage d'une zone mémoire*
Dans un fichier comptable, le descriptif de chaque compte comporte trois zones consécutives codées de la manière suivante :
- le numéro du compte (binaire sur un octet),
- le libellé du compte (20 caractères ASCII sur un octet chacun),
- le solde du compte (réel sur 32 bits en IEEE 754).

Donnez en hexadécimal la codification du compte contenant :
`204PRODUITS_FINANCIERS_520,50` (le souligné correspondant à un caractère espace).

26. *Codage avec bit de parité*
a) On ajoute au code ASCII sur sept bits un premier bit de parité. Le bit de parité vaut 1 quand le nombre de bits à 1 du code sur sept bits est impair et vaut 0 quand le nombre de bits à 1 du code sur sept bits est pair.
Exemple : 11001111 (le premier bit à gauche est le bit de parité).
Donnez dans ce code le caractère F.
b) On ajoute un octet de parité à chaque bloc de quatre caractères : le premier bit est le bit de parité des quatre bits de parité, le deuxième bit est le bit de parité des quatre premiers bits des codes ASCII, etc.



Donnez la codification du bloc de 4 caractères FUTE.
c) Si un des bits a été changé par erreur lors d'une transmission, comment peut-on trouver et corriger le bit erroné ?

27. *Réels double précision*
La norme IEEE 754 pour les réels longs sur 64 bits est similaire à la norme pour les réels courts sur 32 bits avec les caractéristiques suivantes :
- bit de signe (1 pour négatif, 0 pour positif),
- exposant sur 11 bits en excédent à 1023,
- mantisse sur 52 bits normalisée en 1.bbbb (où b est un chiffre binaire quelconque) le 1. n'étant pas représenté (bit caché).

a) donnez en hexadécimal la codification du nombre décimal 157.6875.
b) convertissez en décimal la valeur hexadécimale C00F000000000000.

28. *Réels IEEE 754*
Soient les deux nombres codés suivant la norme IEEE 754 et représentés en hexadécimal : 3EE00000 et 3D800000.
Calculez en la somme et donnez le résultat sous forme IEEE 754 et sous forme décimale.

29. *Sans calculs*
Classez les nombres suivants (exprimés en complément à 2) dans l'ordre croissant. Justifiez votre réponse sans calculer les nombres.

```
A : 11111111
B : 01111111
C : 01110001
D : 10000001
E : 11110001
```

30. *Suites*
On considère la suite L1 d'octets suivante en hexadécimal : 00 FF 00 41 00 42 FF FA
a) Déterminez la suite L2 de nombres décimaux qui correspond au cas où L1 est un codage en binaire pur sur un octet.
b) Déterminez la suite L3 de nombres décimaux qui correspond au cas où L1 est un codage en complément à deux sur un octet.
c) Déterminez la suite L4 de nombres décimaux qui correspond au cas où L1 est un codage en binaire pur sur des mots de deux octets.
d) Déterminez la suite L5 de nombres décimaux qui correspond au cas où L1 est un codage en complément à deux sur des mots de deux octets.
e) Donnez la suite d'octets L6 telle que L6 représente L2 codée en binaire pur sur deux octets.
f) Donnez la suite d'octets L7 telle que L7 représente L3 codée en complément à deux sur deux octets.

3 Le matériel

Dans ce chapitre, nous étudions le matériel selon une approche en différents niveaux ou couches. Nous nous limitons aux concepts de base, utiles pour la culture générale de tout informaticien professionnel. Les différents niveaux que nous allons décrire sont les suivants :

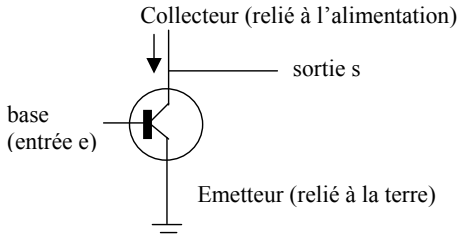
- les composants électroniques qui sont à la base de tout,
- les portes logiques ET, OU, NON, ... qui sont des assemblages de composants électroniques réalisant les fonctions logiques élémentaires,
- les circuits logiques (additionneur, comparateur, bascule, etc.) qui sont des assemblages de portes logiques,
- les constituants des machines (mémoire, unité arithmétique et logique, etc.) qui sont des assemblages de circuits logiques,
- les machines (PC, serveur, super-calculateur ...) qui sont des assemblages de constituants.

3.1 Les composants électroniques

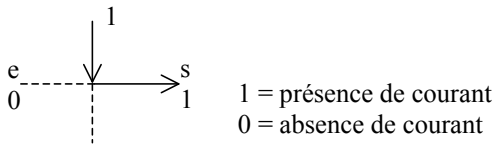
a) Le transistor

Le transistor est le composant actif de base.

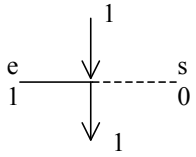
Il est décrit par l'électronicien de la manière suivante.



Son fonctionnement peut être résumé de la manière suivante. Du courant arrive par le collecteur. Si aucun courant n'arrive par la base, le transistor est « bloqué » et ne laisse pas passer le courant. Le courant entré par le collecteur sort obligatoirement par la sortie. Soit en simplifiant le dessin :



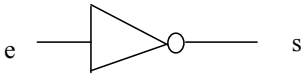
Au contraire, si du courant arrive par la base, le transistor est « saturé » ou « passant ». Il laisse passer le courant. Celui-ci va toujours vers la terre, via l'émetteur, et ne passe plus par la sortie.



En notant e la base et s la sortie, 0 l'absence de courant et 1 la présence de courant, on obtient la table suivante (table de vérité) qui résume la fonction réalisée par le transistor :

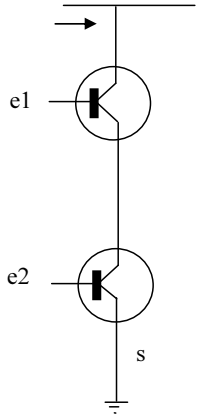
e	s
0	1
1	0

Remarque : on pourrait aussi désigner par VRAI la présence de courant et par FAUX, son absence.
On reconnaît aisément dans cette table la fonction logique NON ou négation.
Une autre représentation de cette fonction est l'équation logique : $s = \bar{e}$, qui se lit « **s vaut 1 (noté s) si et seulement si e vaut 0 (noté $\bar{e}$)** ».
Enfin la représentation graphique dans la norme internationale d'une porte NON est la suivante :



b) Combinaison de 2 transistors
En combinant 2 transistors on obtient 2 autres fonctions logiques.

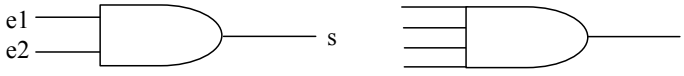
1. Combinaison en série selon le montage suivant :



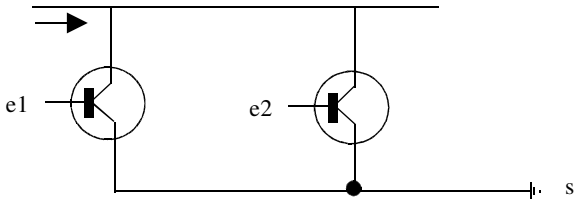
Pour que le courant aille vers la terre (s) il faut que les 2 transistors soient saturés (e1=1 et e2=1) ; sinon, un des transistors est bloqué et le courant sort obligatoirement par le haut.
La table de vérité est donc la suivante :

e1	e2	s
0	0	0
0	1	0
1	0	0
1	1	1

On reconnaît la fonction logique ET. L'équation logique s'écrit : s = e1.e2, qui se lit « **s vaut 1 si et seulement si e1 ET e2 valent 1** ».
La représentation graphique du ET (à 2 ou n entrées) est la suivante.



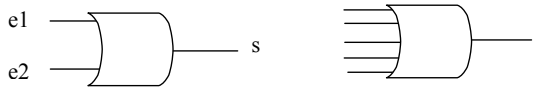
2. Combinaison en parallèle selon le montage suivant :



Le courant n'atteint pas s seulement si les deux transistors sont bloqués (e1 et e2 = 0) ; sinon, un des transistors est saturé et le courant va vers la terre (s).
La table de vérité est donc la suivante :

e1	e2	s
0	0	0
0	1	1
1	0	1
1	1	1

On reconnaît la fonction logique OU. L'équation logique est : $s = \bar{e1}.e2 + e1.\bar{e2} + e1.e2$, qui se lit « **s vaut 1 si et seulement si [e1 vaut 0 ET e2 vaut 1] OU [e1 vaut 1 ET e2 vaut 0] OU [e1 vaut 1 ET e2 vaut 1]** ».
La représentation graphique du OU (à 2 ou n entrées) est la suivante.

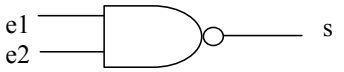


3.2 Les portes logiques

Avec le NON, le ET et le OU on peut construire toutes les fonctions logiques. Le NON-OU (négation du OU), le NON-ET (négation du ET), et le OU exclusif (XOR qui vaut 1 si l'une des entrées ou l'autre mais pas les deux valent 1) constituent d'autres portes logiques souvent utilisées.

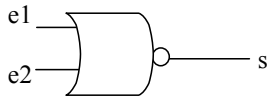
e1	e2	NON ET
0	0	1
0	1	1
1	0	1
1	1	0

$s = \overline{e1}.\overline{e2} + \overline{e1}.e2 + e1.\overline{e2}$



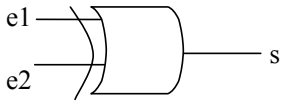
e1	e2	NON OU
0	0	1
0	1	0
1	0	0
1	1	0

$s = \overline{e1}.\overline{e2}$



e1	e2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

$s = \overline{e1}.e2 + e1.\overline{e2} = e1 \oplus e2$



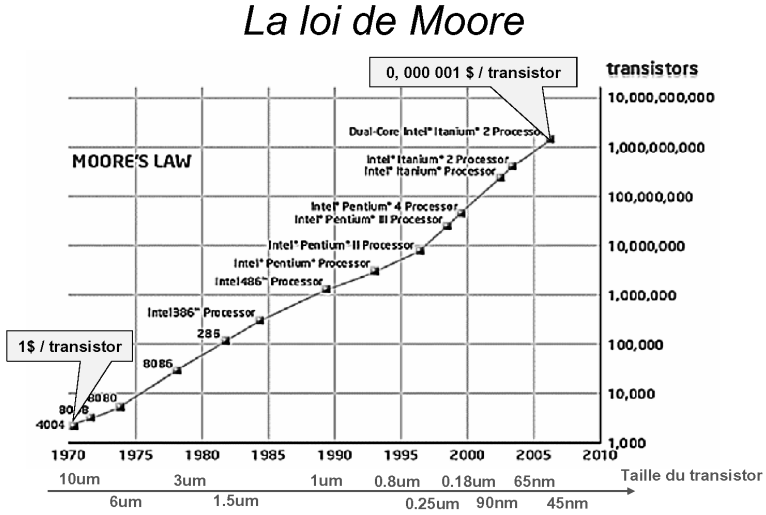
3.3 Les circuits logiques

Il existe deux grandes catégories de circuits logiques :

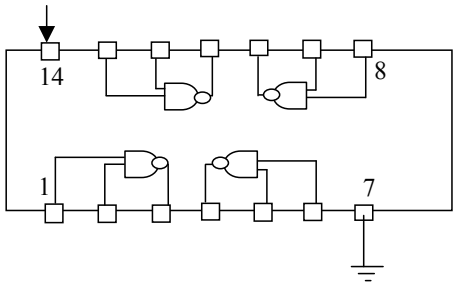
1. les circuits « combinatoires » ou « de calcul », où les sorties (S_i) ne dépendent que des entrées (E_i) : S_1, ... , S_n =f(E_1, ...,E_m),

2. les circuits « séquentiels », où les sorties à l'instant t dépendent des entrées et des sorties à l'instant t-1 : $S_1(t)$, ... , $S_n(t)$ =f(E_1, ...,E_m, $S_1(t-1)$, ..., $S_n(t-1)$) ; il y a donc possibilité de conserver de l'information du temps $t-1$ au temps t et donc de construire des mémoires (circuits qui « se souviennent » des valeurs précédentes).

Il ne faut pas confondre circuits logiques et circuits physiques. Les circuits physiques sont des boîtiers (puces) qui renferment un nombre plus ou moins important de portes logiques interconnectées. Les microprocesseurs sont des processeurs intégrés dans une puce et construits avec des circuits à très forte intégration (VLSI pour *very large scale integration*). La loi de Moore, énoncée en 1965 et approximativement vérifiée depuis, prédisait que le nombre de transistors intégrés dans les puces allait doubler chaque année.



Au contraire, le CMOS 4011 représenté ci-dessous est un circuit à très faible intégration qui contient quatre portes NON-ET et 14 broches (la broche 14 est l'alimentation et la broche 7 est à la terre). Un tel circuit permet de réaliser des petits montages en connectant les broches.



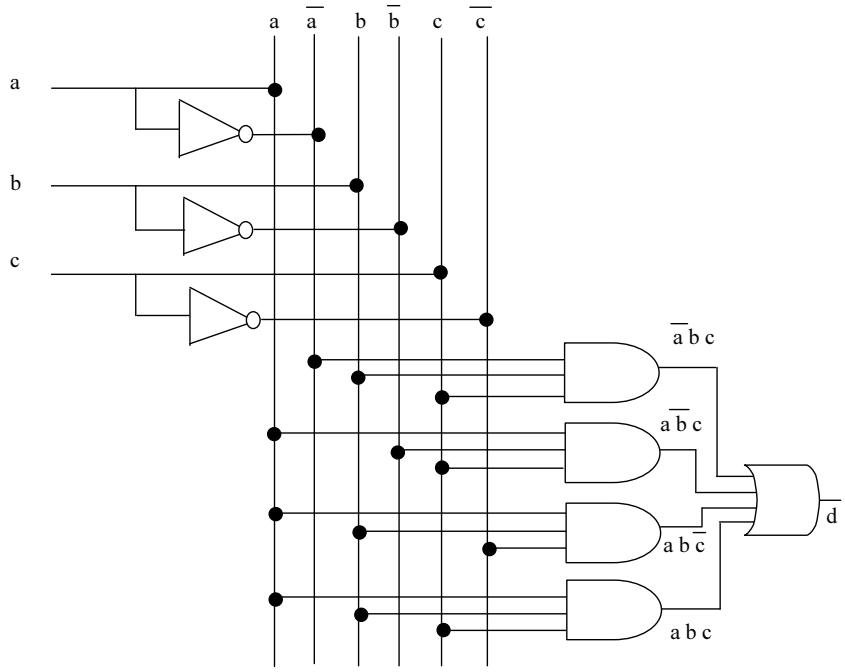
On peut noter au passage qu'avec uniquement des portes NON-ET (ou uniquement des portes NON-OU) on peut construire des circuits équivalents aux portes ET, OU et NON et donc n'importe quelle fonction logique. On le montrera dans un des exercices.

3.3.1 Construction d'un circuit combinatoire quelconque

Une démarche systématique est possible à partir de la table de vérité. Par exemple :

a	b	c	d
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

On écrit l'équation logique qui lui est associée. Soit : $d = \bar{a}.b.c + a.\bar{b}.c + a.b.\bar{c} + a.b.c$ Il suffit de prendre chaque ligne de la table de vérité où le résultat vaut 1 et de transcrire la conjonction des données qui produit ce résultat. Sur le dessin, on construit toutes les entrées possibles (a, $\bar{a}$, b, $\bar{b}$, c et $\bar{c}$) en plaçant systématiquement une porte NON sur chaque entrée. Puis on construit chaque terme de l'équation logique avec ces entrées et des portes ET. Enfin, on regroupe les termes avec des portes OU pour obtenir la fonction logique souhaitée. On obtient donc le circuit suivant :



Il est clair que plus l'équation logique est simple, plus le circuit est simple. Il est donc intéressant d'appliquer les règles de l'algèbre de Boole pour simplifier l'équation logique. Vous étudierez cette théorie plus en détail en cours d'algèbre. Elle est définie par les axiomes et théorèmes suivants :

$\bar{0} = 1$	$\bar{1} = 0$	constantes
$a + 0 = a$	$a.1 = a$	éléments neutres
$a + 1 = 1$	$a.0 = 0$	éléments absorbants
$a + a = a$	$a.a = a$	éléments idempotents
$a + \bar{a} = 1$	$a.\bar{a} = 0$	éléments complémentaires
$(a + b) + c = a + (b + c)$	$(a.b).c = a.(b.c)$	associativité
$a + b = b + a$	$a.b = b.a$	commutativité
$a.(b + c) = a.b + a.c$	$a + b.c = (a + b).(a + c)$	distributivité
$\bar{\bar{a}} = a$		double négation
$\overline{a + b} = \bar{a}.\bar{b}$	$\overline{a.b} = \bar{a} + \bar{b}$	théorèmes de De Morgan

Les démonstrations se font très facilement en utilisant les tables de vérité.
Exemples :

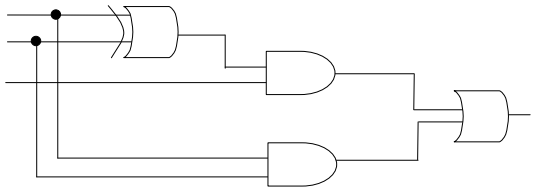
Montrons que $a + \bar{a} = 1$

a	$\bar{a}$	$a + \bar{a}$
0	1	1
1	0	1

Montrons que a.(b+c) = a.b + a.c

a	b	c	b+c	a.(b+c)	a.b	a.c	a.b+a.c
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

La simplification du circuit construit au début du paragraphe est donnée ci-dessous. L'équation logique est factorisée puis simplifiée : $d = \bar{a}.b.c + a.\bar{b}.c + a.b.\bar{c} + a.b.c = (\bar{a}.b + a.\bar{b}).c + a.b.(\bar{c} + c) = (a \oplus b).c + a.b.$ Soit le circuit :



À noter qu'il existe des méthodes systématiques de simplification, comme la méthode des tableaux de Karnaugh. Elle est expliquée à l'annexe C.

Les exercices suivants permettent de construire des circuits de calcul de plus en plus complexes jusqu'à obtenir une unité arithmétique et logique simplifiée.

Exercices

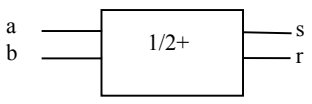
1. Donnez la table de vérité, l'équation logique et le schéma du OU exclusif (en suivant la méthode de construction systématique exposée dans le cours).

2. Montrez que $(a + b).(\bar{a} + \bar{b}) = \bar{a}.b + a.\bar{b}$. Donnez un deuxième circuit possible pour le OU exclusif.

3. Montrez qu'avec uniquement des NON-OU on peut réaliser le NON, le OU et le ET.

4. Construisez un demi-additionneur. Ce circuit logique a l'apparence suivante et correspond à la table de vérité ci-dessous.

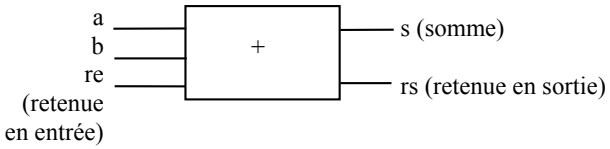
a	b	s	r
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Il réalise l'addition de deux bits a et b avec la somme dans s et la retenue dans r. Penser à utiliser le OU exclusif.

5. A partir du demi-additionneur on veut construire un additionneur complet. Ce circuit permet d'additionner deux bits et une retenue en entrée. On essayera de le construire en combinant deux

demi-additionneurs. Puis on vérifiera le résultat obtenu grâce à une table de vérité.

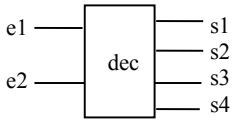


Remarque : l'applet JARCHI (disponible sur le site `www.loria.fr/~jloncham`, onglet enseignement) donne une simulation pas à pas d'un demi-additionneur et d'un additionneur complet.

6. Construisez un additionneur de deux mots de n bits à partir de n additionneurs complets.

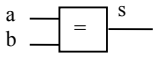
7. Construisez un décodeur qui transforme l'information portée par n entrées (le code binaire) en un choix d'un fil de sortie parmi les 2^n sorties possibles.
Exemple pour $n = 2$:

e1	e2	s1	s2	s3	s4
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

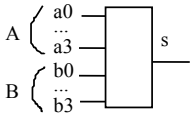


Remarque : l'applet JARCHI permet une simulation pas-à-pas du décodeur.

8. Construisez un comparateur (exercice facultatif).
a) cas simple (un bit) : s = 1 si a = b, sinon s = 0.



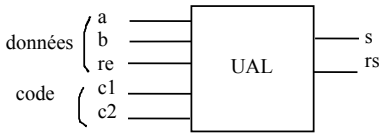
Ecrivez l'équation logique. Pensez au OU exclusif.
b) cas plus complexe (quatre bits) : s = 1 si A = B, sinon s = 0.



Généralisez la solution de la question a).

9. Construisez une unité arithmétique et logique élémentaire. A partir d'un code opération sur deux bits (quatre opérations possibles) et trois bits de données, elle réalise soit un OU, soit un ET, soit un NON, soit une addition de deux bits.

c1	c2	résultat
0	0	a ET b dans s, 0 dans rs
0	1	a OU b dans s, 0 dans rs
1	0	NON a dans s, 0 dans rs
1	1	somme de a et b dans s, retenue dans rs



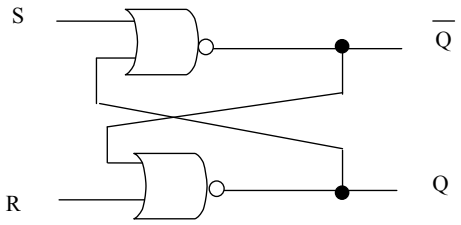
On utilisera un décodeur et un additionneur complet. Le principe est le suivant : à partir des fils de données, faites tous les « calculs » demandés ; grâce à une porte sur chaque fil de sortie du décodeur, sélectionnez un des résultats et amenez ce résultat dans les fils s et rs.

Remarques :

– l'applet JARCHI donne une simulation pas à pas de ce circuit,
– le lien entre matériel et logiciel commence un peu à apparaître. Comme nous le verrons plus tard, chaque instruction du langage machine comprend un code opération. Grâce au décodeur et aux portes de sélection d'un des circuits de calcul, ce code opération déclenche l'apparition du résultat de l'opération voulue sur les fils de sortie de l'UAL.

3.3.2 Un circuit séquentiel – la bascule

Une bascule est un circuit qui peut prendre deux états (0 et 1). En absence d'entrée la bascule reste dans l'état qu'elle possède. Elle fonctionne donc comme une mémoire de 1 bit. Les entrées permettent de modifier cet état. Les deux sorties donnent toujours la valeur de l'état et son complément. Il existe plusieurs types de bascules.
La bascule RS est constituée de deux portes NON-OU dont les sorties « bouclent » sur les entrées de manière croisée. L'état se lit sur le fil de sortie Q. A la mise sous tension Q et $\bar{Q}$ doivent avoir des valeurs opposées (0 et 1 ou 1 et 0). Ces valeurs doivent toujours rester opposées.

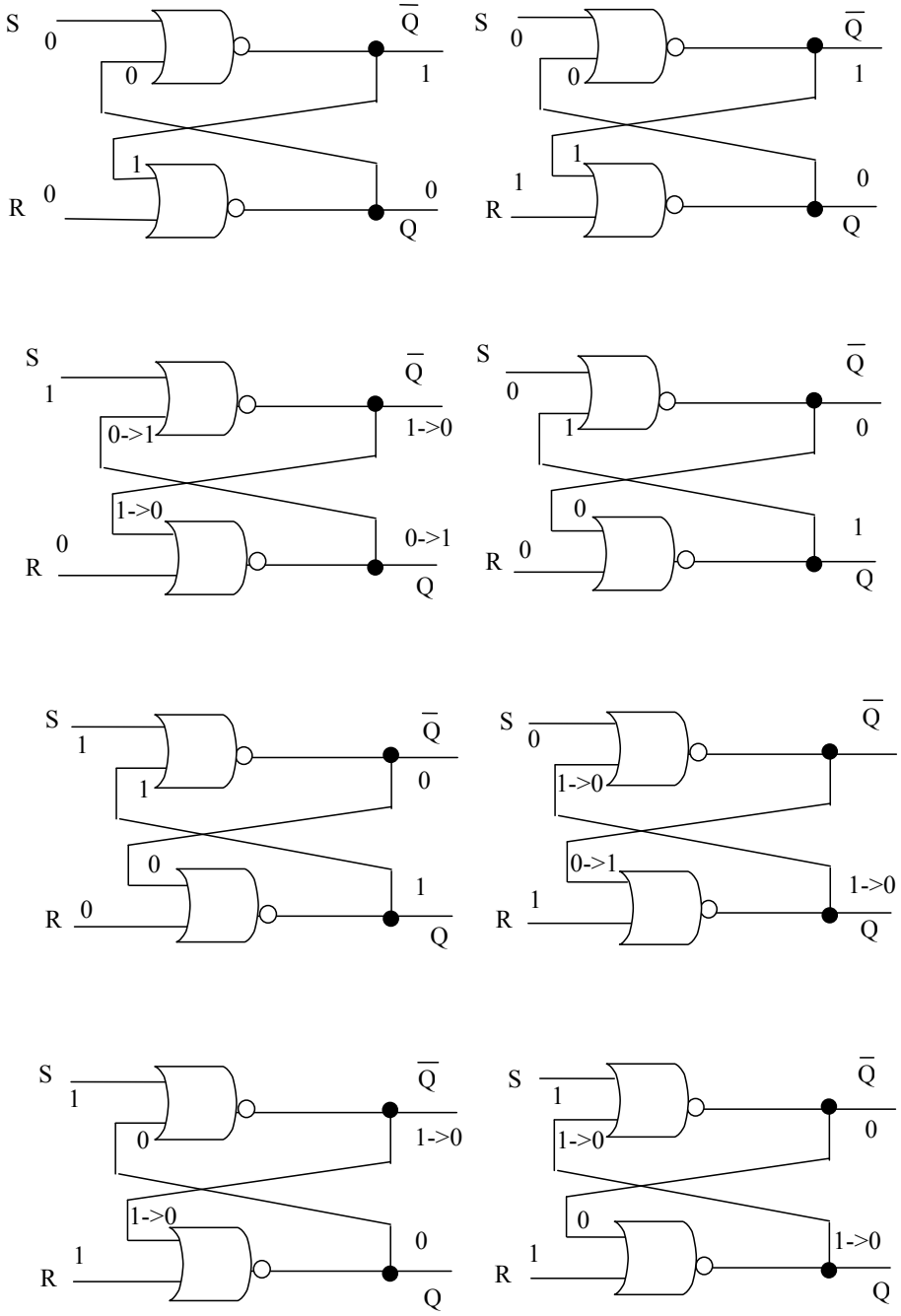


Il y a huit cas à regarder, fonction des deux entrées et de l'état initial de Q. Les figures ci-après illustrent ces huits cas.
Si $Q = 0$ et si $R = S = 0$ (pas d'entrées), Q reste à 0. La bascule garde indéfiniment en mémoire le bit 0 (état stable à 0).
Si $Q = 1$ et si $R = S = 0$ (pas d'entrées), Q reste à 1. La bascule garde indéfiniment en mémoire le bit 1 (état stable à 1).
Si $Q = 1$ et si $R = 1$, Q passe à 0 (*reset*). La valeur de la mémoire est modifiée.
Si $Q = 0$ et $R = 1$, la mémoire reste à 0.
Si $Q = 0$ et si $S = 1$, Q passe à 1 (*set*). La valeur de la mémoire est modifiée.
Si $Q = 1$ et $S = 1$, la mémoire reste à 1.
$S = 1$ en même temps que $R = 1$ doit être interdit. On arrive en effet à un état incohérent où $Q = \bar{Q}$.
En jouant avec les entrées R et S, on peut donner à la mémoire la valeur que l'on veut.

JARCHI permet de visualiser les évolutions pas à pas des 8 cas possibles (selon les valeurs de Q, S et R).

Si on veut utiliser une table de vérité pour décrire la bascule RS il faut distinguer Q et $\bar{Q}$ à l'instant n (avant l'arrivée des bits sur R et S) et à l'instant n+1 (après l'arrivée des bits sur R et S).

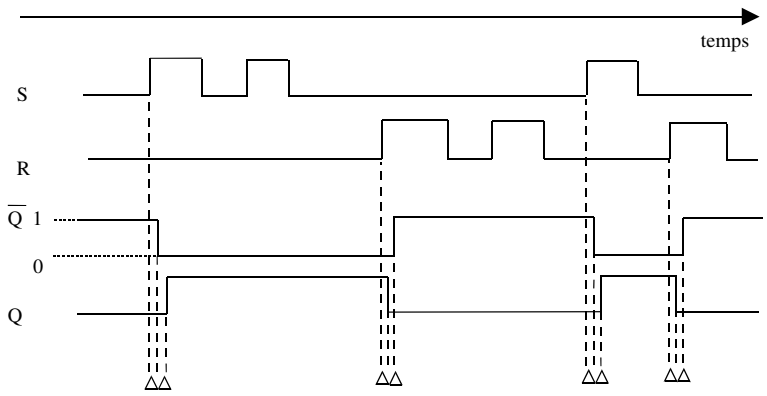
R	S	Q_n	$\overline{Q_n}$	Q_{n+1}	$\overline{Q_{n+1}}$	Commentaire
0	0	0	1	0	1	mémorisation de 0
0	0	1	0	1	0	mémorisation de 1
0	1	0	1	1	0	mise à 1
0	1	1	0	1	0	déjà à 1
1	0	0	1	0	1	déjà à 0
1	0	1	0	0	1	mise à 0
1	1	0	1	0	0	interdit
1	1	1	0	0	0	interdit



Cette table peut se simplifier en :

R	S	Q_{n+1}	Commentaire
0	0	Q_n	mémorisation
0	1	1	mise à 1
1	0	0	mise à 0
1	1	–	interdit

Si on veut décrire encore plus précisément les circuits séquentiels il faut tenir compte des durées de basculement des portes et de propagation des bits entre les portes. Le chronogramme ci-après est une manière de représenter les évolutions temporelles d'un circuit.

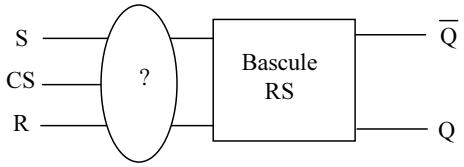


On note qu'un Set fait d'abord basculer $\bar{Q}$, après un délai Δ et ensuite Q, après un deuxième délai Δ. Un reset fait d'abord basculer Q, après un délai Δ et ensuite $\bar{Q}$, après un deuxième délai Δ. Δ correspond au temps de basculement de la porte NON-OU, les temps de propagation des signaux étant considérés comme négligeables.
Pour les circuits séquentiels complexes, les chronogrammes sont des éléments importants d'analyse (cf. exercice 15).

Les exercices suivants permettent de construire des circuits de mémoire de plus en plus complexes jusqu'à obtenir une mémoire centrale simplifiée.

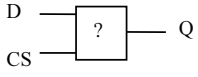
Exercices

10. A partir de la bascule RS, ajouter une entrée CS (*chip select*) qui inhibe les entrées R et S. Si $CS = 0$, R et S n'ont pas d'effet. Si $CS = 1$, R et S ont leur effet normal. Grâce à CS, la bascule ne peut changer de valeur que si la bascule est sélectionnée ($CS = 1$).



Si le fil CS est relié à une horloge, la bascule ne peut évoluer qu'au moment de la réception des tops d'horloge. Avec ce montage, on parle de bascule « synchrone ».

11. Reprendre la bascule précédente et construire une bascule avec une seule entrée D, telle que :
$D = 0 \Leftrightarrow S = 0$ et $R = 1$,
$D = 1 \Leftrightarrow S = 1$ et $R = 0$.
Ceci élimine les cas interdits (S et $R = 1$ en même temps).

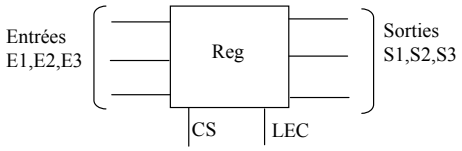


On a donc une mémoire de 1 bit qui garde sur Q la valeur de D (D comme *data* ou « donnée ») et qui ne peut changer de valeur que quand la mémoire est sélectionnée (CS à 1). On parle de « bascule D ». Sa table de vérité est très simple :

D	Q_{n+1}	Commentaire
0	0	mémorisation de 0
1	1	mémorisation de 1

Remarque : l'applet JARCHI donne la simulation pas à pas de la bascule RS et de la bascule D.

12. Construire un registre de trois bits avec trois bascules D.
On appelle CS_i et Q_i, les fils CS et Q de la i^e bascule.

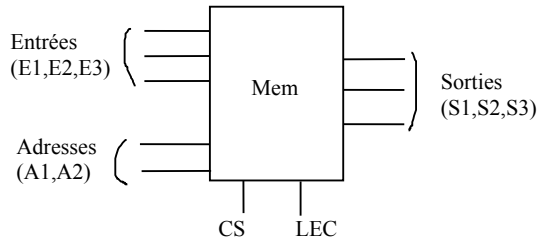


Si $CS = 0$, les bascules ne peuvent être modifiées ($CS_i = 0$) et les $S_i = 0$.
Si $CS = 1$ et $LEC = 0$, les bascules mémorisent les E_i et les $S_i = 0$ (le registre est « écrit »).
Si $CS = 1$ et $LEC = 1$, les S_i prennent les valeurs des bascules (Q_i) qui ne changent pas (le registre est « lu »).
On pourra s'aider des équations logiques des S_i et des CS_i :

$$S_i = CS.LEC.Q_i \text{ (S_i vaut 1 ssi } CS = 1 \text{ et } LEC = 1 \text{ et } Q_i = 1)$$
$$CS_i = CS.\overline{LEC} \text{ (CS_i vaut 1 ssi } CS = 1 \text{ et } LEC = 0).$$

Remarque : l'applet JARCHI permet de simuler ce registre.

13. Construire une mémoire de 4 mots de 3 bits, avec quatre registres de 3 bits et un décodeur pour l'adresse. L'adresse contient le numéro du mot à lire/écrire (de 0 à 3).

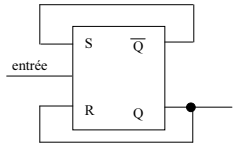


On pourra s'aider des équations logiques suivantes (où f_i est un fil de sortie du décodeur d'adresse) :

$CS_i = f_i.CS$ (seul un registre est sélectionné : celui dont on donne l'adresse),
$LEC_i = LEC$ (tous les registres sont soit lus soit écrits).

Remarques :
- l'applet JARCHI donne une simulation de cette mémoire.
- le lien entre matériel et logiciel se précise un peu plus ; comme nous le verrons plus tard, en plus du code opération, chaque instruction en langage machine contient une (des) adresse(s) d'opérande(s). Grâce au décodeur, chaque adresse permet de récupérer une valeur qui peut être envoyée en entrée de l'UAL pour subir l'opération décrite par le code opération.

14. Reprendre la bascule RS avec fil CS de l'exercice 10. On considère ce fil CS comme une entrée et on boucle la sortie $\bar{Q}$ sur l'entrée S et la sortie Q sur l'entrée R.

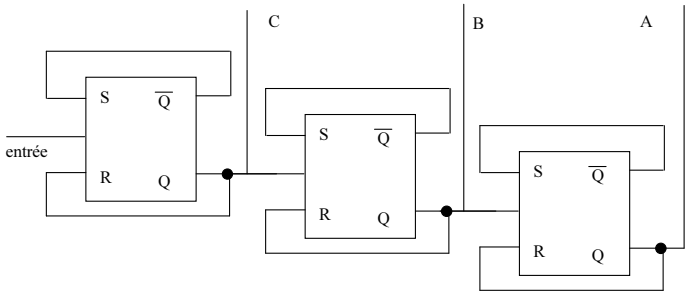


Donner le comportement de ce circuit en remplissant sa table de vérité et en donnant son chronogramme :

entrée	Q_t	Q_{t+1}
0	0	
0	1	
1	0	
1	1	

Résumer ce comportement en une phrase.

15. On monte en série trois circuits du type de celui de l'exercice 14.



On envoie de brèves impulsions à 1 successives sur le fil d'entrée. On suppose que par construction la bascule change de valeur à la fin de l'impulsion (sur son front descendant de 1 à 0). Initialement, les trois bascules sont à 0. Dessiner le chronogramme montrant les valeurs sur le fil d'entrée et sur les fils de sortie A, B et C. Quelle fonction réalise ce circuit ?

3.4 La structure d'un ordinateur

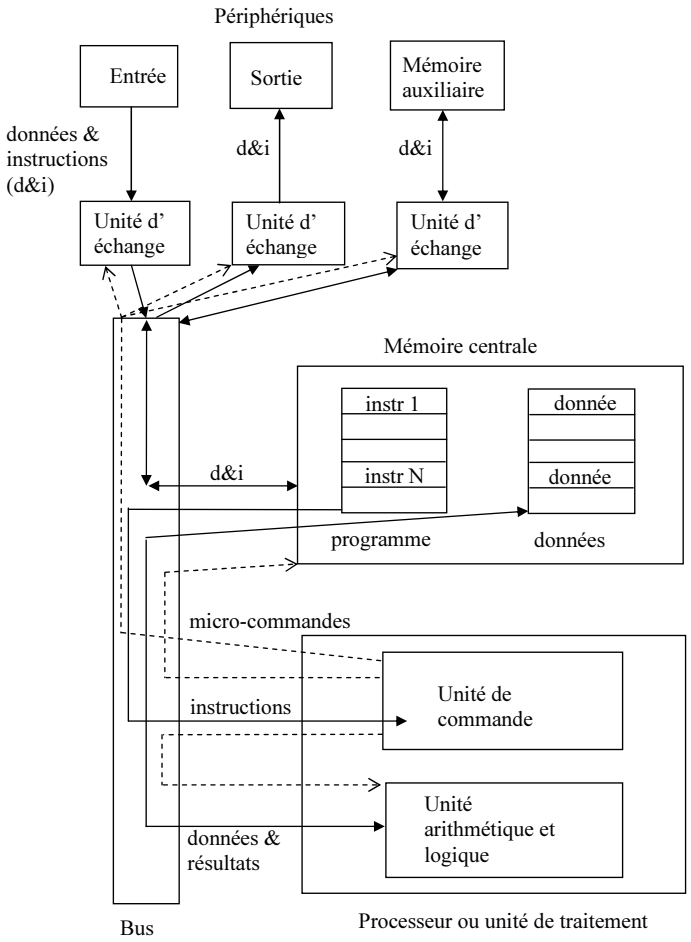
C'est en 1946 que John von Neumann a posé les principes de ce qu'est un ordinateur, c'est-à-dire une machine de traitement de l'information à programme enregistré. Les instructions du programme sont rangées dans les cellules successives d'une mémoire et sont exécutées dans l'ordre de leur rangement (sauf instruction de ≪ branchement ≫ ou ≪ saut ≫ ou ≪ rupture de séquence ≫ qui aiguille vers une autre partie du programme). Comme le programme est introduit en mémoire —et non figé dans les circuits de la machine— on peut en changer à volonté : on parle de machine ≪ universelle ≫ de traitement de l'information.

3.4.1 Organisation générale en composants

Nous nous intéressons ici aux ordinateurs ≪ classiques ≫ avec une seule unité de traitement ou processeur. Il existe en effet des architectures plus complexes avec plusieurs processeurs dont nous parlerons ultérieurement.
Un ordinateur classique comporte cinq types de composants :

1. la mémoire centrale,
2. l'unité de traitement (ou processeur ou unité centrale ou CPU) qui comprend l'unité de commande (ou unité de contrôle) et l'unité arithmétique et logique ; on peut noter au passage qu'un micro-processeur est un processeur implanté sur une seule puce,
3. les unités d'échange qui interfacent les périphériques,
4. les périphériques d'entrée, de sortie, de mémoire auxiliaire,
5. les bus pour la circulation des informations entre composants.

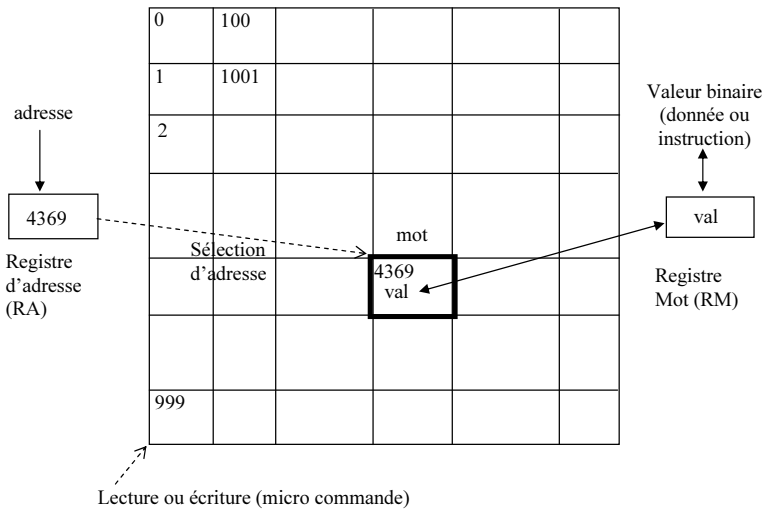
Les instructions (entrée, sortie, transfert vers ou depuis les mémoires auxiliaires, calcul arithmétique et logique ...) sont interprétées par l'unité de commande qui actionne les autres composants pour les réaliser via des micro-commandes (ordres donnés aux circuits).
La figure ci-dessous précise les flux d'informations (données, instructions, micro-commandes). On note le rôle essentiel de la mémoire centrale : tout (données, résultats, programmes) transite par elle.



Chaque type de processeur est capable d'exécuter un certain ensemble d'instructions, son « jeu d'instructions ». Cet ensemble d'instructions constitue le langage directement compréhensible par le processeur ou « langage machine ».
Les données sont transformées par l'unité arithmétique et logique qui effectue les opérations arithmétiques ($+$, $-$, $\times$, $/$ ...) et logiques ($=$, $<$, $>$, ET, OU ...).
Ces principes très simples n'ont pu être concrétisés que dans les années 1950 avec l'avènement de l'électronique moderne.

3.4.2 La mémoire centrale

a) Principes
C'est un ensemble de cellules mémoires (« mots ») qui peuvent contenir chacune une donnée ou une instruction en binaire. Chaque cellule est repérée par un numéro, ou adresse. On peut lire ou écrire une information à une adresse donnée. Le temps d'accès est le même quel que soit le mot concerné. L'adresse doit être placée dans le registre d'adresse (RA). Un dispositif sélectionne la bonne cellule (cf. le décodeur et le fil *chip select* vu précédemment). En cas d'écriture, l'information à écrire doit être placée dans le registre mot (RM). En cas de lecture, l'information lue est placée dans le registre mot.
La mémoire centrale perd tout son contenu lorque l'alimentation électrique est coupée (contrairement aux mémoires externes comme les disques). La mémoire centrale peut être sujette à des erreurs soit permanentes (usure, défaut de fabrication) soit intermittentes (problème d'alimentation électrique, particules alpha). On stocke souvent chaque mot de M bits avec K bits supplémentaires permettant de détecter et de corriger d'éventuelles erreurs. Ces K bits de contrôle sont calculés avant écriture en mémoire, stockés en mémoire et recalculés en lecture pour comparaison et éventuellement correction de la donnée.



b) Technologies

Du point de vue technologique on distingue 2 types de mémoires centrales (ou ≪ mémoires vives ≫ ou RAM pour *Random Access Memory*) dont les données sont perdues en absence d'alimentation électrique :

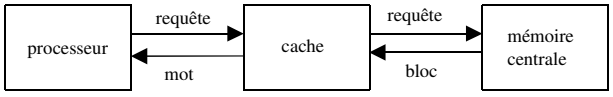
1. les RAM statiques (SRAM) qui utilisent le principe des bascules ; elles sont très rapides, de prix élevé et donc principalement utilisées dans les mémoires caches (cf. ci-après),
2. les RAM dynamiques (DRAM de divers types SDRAM, RDRAM, DR-SDRAM, DDR-SDRAM, DDR2-SDRAM . . .) qui nécessitent un rafraîchissement périodique des informations ; elles sont moins rapides, moins coûteuses et plus fortement intégrées. Elles sont utilisées pour les mémoires centrales.

Il existe aussi des ≪ mémoires mortes ≫ à lecture seule (ROM pour *Read Only Memory*). Elles gardent leurs valeurs même en absence d'alimentation électrique. On distingue les :

- ROM véritables : chargées en informations à la fabrication,
- PROM (Programmable ROM) : chargées par l'utilisateur du circuit (écriture une seule fois),
- EPROM (Erasable PROM) : effaçables en les plaçant dans une machine spéciale (sous rayons ultraviolets par exemple) et modifiables plusieurs fois,
- EEPROM (Electrically EPROM) : effaçables électriquement (plusieurs centaines de micro secondes par octet).

c) Mémoire cache

Le processeur est beaucoup plus rapide que la mémoire centrale. Pour tenir compte de cette différence de vitesse on intercale une mémoire cache plus rapide que la mémoire centrale entre le processeur et la mémoire centrale. La mémoire cache est réalisée avec des technologies coûteuses et reste donc de faible capacité. Son principe est le suivant : si l'information utile, c'est-à-dire un mot mémoire particulier, est dans le cache on y accède rapidement ; sinon, on accède à la mémoire centrale et on range dans le cache un bloc de K mots autour du mot désiré, car il y a une bonne probabilité que ce mot ou un mot proche soit de nouveau utile peu de temps après. C'est le principe de ≪ localité spatiale et temporelle ≫, lié au fait que les instructions se suivent en mémoire et que les données sont souvent stockées consécutivement. Un algorithme de remplacement décide quel bloc présent dans le cache est remplacé par le nouveau bloc : le plus ancien (*first in first out* ou FIFO), le plus anciennement utilisé (*last recently used* ou LRU), le moins fréquemment utilisé (*least frequently used* ou LFU), un pris au hasard, etc.



Les ordinateurs actuels possèdent souvent plusieurs niveaux de cache. Sur les *PC*, le cache *L1* (*level 1*), le plus rapide, est interne au micro-processeur. Il comporte un mini cache d'instructions et un

mini cache de données. Le cache *L2*, un peu moins rapide, est souvent inclus dans le même boîtier que le processeur. Il est parfois complété par un cache *L3*, situé sur la carte mère.

d) Hiérarchie des mémoires

L'idéal, pour un ordinateur, serait de disposer d'un seul type de mémoire, très rapide et de très grande capacité. Cela est malheureusement irréalisable actuellement du fait du coût engendré, les mémoires les plus rapides étant aussi les plus chères. Par ailleurs, bien que la capacité totale de données à mémoriser soit importante, il n'est pas nécessaire de disposer d'un temps d'accès rapide pour toutes les données. Seules les données les plus utilisées nécessitent un temps d'accès très court. On utilise donc une *hiérarchie de mémoires* qui va de la mémoire d'accès très rapide en petite quantité (le registre) à la mémoire de très grande capacité offrant un temps d'accès plus important (le disque). Le tableau ci-dessous présente les niveaux mémoire que l'on trouve dans les machines actuelles :

	Registre	Mémoire cache	Mémoire centrale	Disque
Capacités typiques	<Ko	Mo	Go	>100Go
Temps d'accès (ns)	$1-5$	$3-10$	$10-400$	5000000
Coût relatif	50	10	1	0,001

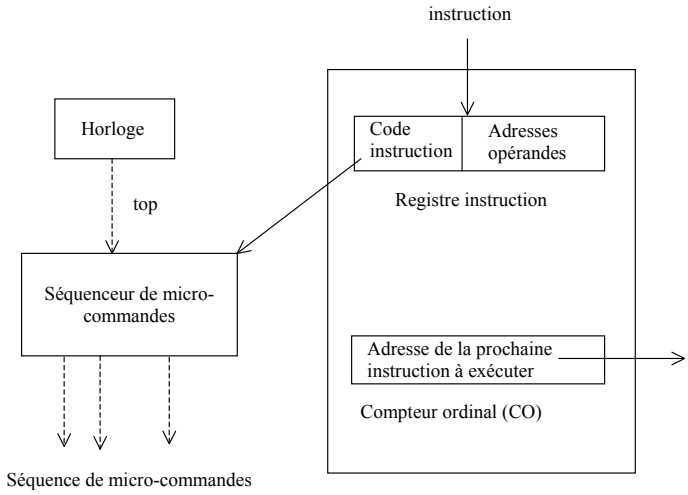
3.4.3 L'unité de commande

a) Les registres

L'unité de commande comporte 2 registres essentiels :

1. le compteur ordinal (CO) : il contient l'adresse mémoire de la prochaine instruction à exécuter ; il est incrémenté de 1 en 1 (sauf instruction de branchement et en supposant que toutes les instructions tiennent dans un et un seul mot) ;
2. le registre instruction (RI) : il reçoit l'instruction à exécuter ; l'instruction contient toujours un code opération (ou code instruction) et de 0 à n adresses d'opérandes.

La prise en compte du code opération consiste à déclencher un ensemble de micro-commandes cadencées par l'horloge qui commandent la réalisation de l'opération par les autres circuits.
On peut donc avoir un même langage machine (mêmes codes opérations) traduit en microcommandes différentes qui reflètent des architectures matérielles différentes.
Plus l'horloge bat vite (fréquence d'horloge exprimée en GigaHertz) et plus l'instruction est réalisée vite sur les circuits de la machine.



b) Le cycle « chercher–décoder–exécuter »

Le principe d'exécution d'un programme est donc la répétition du même cycle d'opérations, appelé cycle « chercher–décoder–exécuter » :

répéter
chercher l'instruction dont l'adresse est dans CO et la mettre dans RI ;
incrémenter de 1 CO ;
décoder le contenu de RI ;
s'il y en a, chercher le (les) opérande(s) dont on a l'adresse (les adresses) ;
exécuter l'opération correspondant au code opération ; si c'est un branchement CO est re-modifié
fin

c) Les architectures CISC et RISC

Le séquenceur, qui génère les microcommandes à partir du code opération, peut être câblé (c'est à dire réalisé par un circuit) ou microprogrammé. Dans ce dernier cas, le séquencement est décrit par une suite de micro-instructions. Ce microprogramme, localisé en ROM, est exécuté par une micro-machine dotée d'un micro-compteur de programme et d'une mémoire.

C'est souvent le cas des processeurs ayant un jeu d'instructions complexes et de tailles variables ou processeurs CISC (*Complex Instruction Set Computer*), comme les processeurs Intel. Au contraire, les processeurs RISC (*Reduced Instruction Set Computer*), comme les processeurs PowerPC, ALPHA, SPARC ou MIPS, ont un petit nombre d'instructions de taille fixe et utilisent en général un séquenceur câblé pour plus d'efficacité.

L'architecture CISC, la plus ancienne, se justife par la lenteur de la mémoire vis à vis du processeur. Il apparaît donc intéressant de définir des opérations complexes, réalisées au sein du processeur, plutôt que des suites d'opérations simples, impliquant de nombreux accès mémoire. Mais des études statistiques ont montré au début des années 70 que 80% des programmes générés par les compilateurs faisaient appel à seulement 20% des instructions machine. D'où l'idée de l'architecture RISC de réduire le jeu d'instructions à ces 20% d'instructions les plus utilisées, en cherchant à les optimiser au maximum avec des séquenceurs câblés. La tendance actuelle consiste à mitiger les architectures RISC avec des caractéristiques issues des architectures CISC ou inversement.

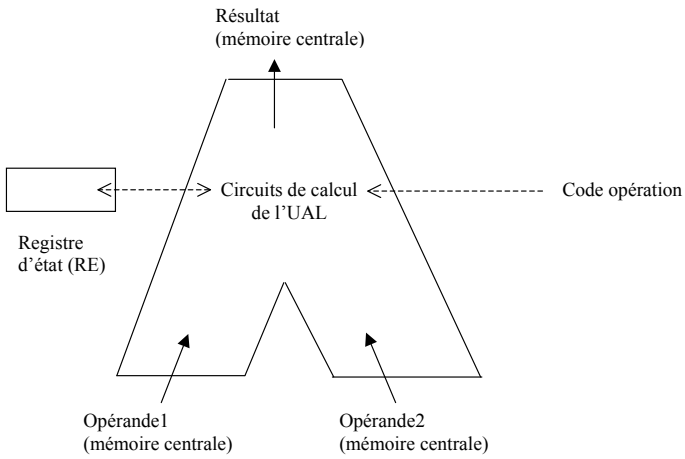
CISC	RISC
jeu d'instructions large	jeu d'instructions réduit
instructions de tailles différentes	instructions de mêmes tailles
instructions de durées différentes	instructions de mêmes durées
séquenceur microprogrammé	séquenceur câblé

3.4.4 L'unité arithmétique et logique (UAL ou ALU)

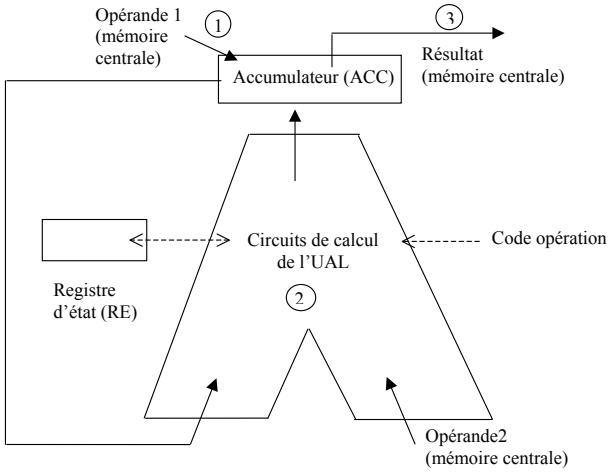
Elle exécute les calculs arithmétiques (comme l'addition) et logiques (comme la comparaison). Les différents bits du registre d'état (RE) renseignent sur le déroulement de l'opération (comme l'existence d'un débordement). On les appelle aussi indicateurs ou drapeaux (*flags*).

Normalement une instruction de calcul devrait comporter trois adresses : les deux opérandes à traiter et l'adresse où ranger le résultat. Très peu de machines sont conçues ainsi car les instructions à trois adresses demandent beaucoup de bits (VAX-11 de Digital).

Pour simplifier les instructions, un des opérandes est souvent placé dans un registre particulier, l'accumulateur (ACC), de même que le résultat. Il suffit donc d'une seule adresse dans l'instruction, puisque la machine travaille implicitement avec ACC pour les deux autres valeurs. En contrepartie, il faut trois instructions pour faire une opération :



1. transfert de l'opérande 1 dans ACC,
2. opération avec ACC et opérande 2 indiqué dans l'instruction ; rangement du résultat dans ACC,
3. transfert du résultat de ACC en mémoire.



Cette organisation avec accumulateur a été utilisée par les premiers microprocesseurs 8 bits (Intel 8080 et Motorola 6800). Dans les microprocesseurs 16 bits (Intel 8086 et Motorola 68000) l'accumulateur a été remplacé par un petit nombre de registres généraux. Pour repérer un registre parmi les quatre (R1, R2, R3, R4) 2 bits suffisent dans les instructions :

1. transfert de l'opérande dans un des 4 registres généraux (Ri),
2. opération avec Ri et l'opérande 2, résultat dans Ri,
3. transfert de Ri en mémoire.

Les microprocesseurs les plus récents (Pentium, PowerPC, ALPHA, SPARC, MIPS ...) utilisent uniquement des registres et plus aucune adresse dans les instructions :

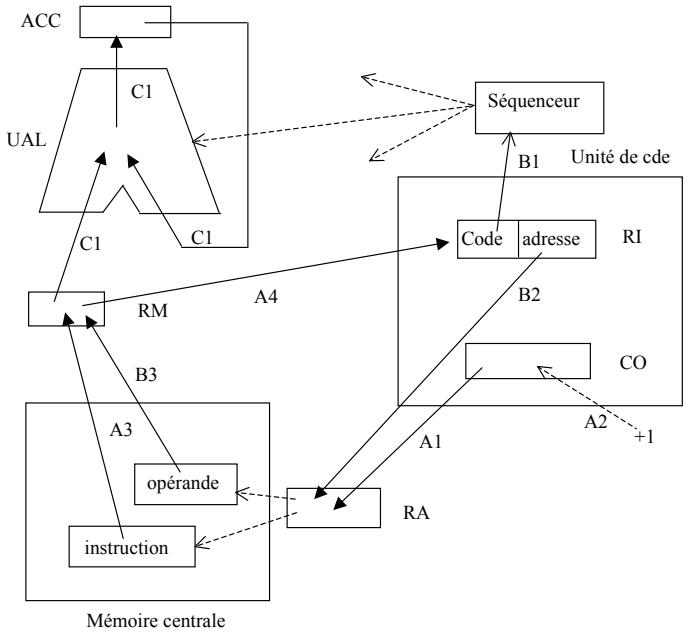
1. transfert de l'opérande 1 dans un registre Ri,
2. transfert de l'opérande 2 dans un registre Rj,
3. opération avec Ri et Rj, résultat dans Rk,
4. rangement de Rk en mémoire.

Il peut alors y avoir un grand nombre de registres adressés par quelques bits seulement.

À noter enfin qu'il peut exister d'autres unités de calcul que l'UAL dans un processeur. C'est le cas des unités de calcul pour les réels flottants (FPU) et des unités de calcul multimédia (MMU) pour les calculs vectoriels.

3.4.5 Le cycle complet d'exécution d'une instruction de calcul ou de branchement

On détaille le cycle « chercher–décoder–exécuter » pour une machine avec un accumulateur :
a) Phase *chercher*
A1 : transfert CO dans RA (adresse de la prochaine instruction à exécuter dans RA).
A2 : incrémentation de 1 de CO (la prochaine instruction est à l'adresse suivante).
A3 : lecture de l'instruction en mémoire et rangement dans RM.
A4 : transfert de l'instruction de RM à RI pour décodage.
b) Phase *décoder*
B1 : analyse du code instruction ; le séquenceur envoie les micro commandes adéquates.
B2 : adresse de l'opérande éventuel dans RA.
B3 : lecture de l'opérande en mémoire et rangement dans RM.
c) Phase *exécuter*
C1 : l'opération de calcul est appliquée aux opérandes dans RM et ACC (rappel : cette instruction de calcul a été précédée par une instruction de transfert du premier opérande de la mémoire vers ACC et sera suivie d'une instruction de transfert du résultat de ACC vers la mémoire).
Pour les instructions de branchement : A1, A2, A3, A4, B1 sont inchangés ; B2 et B3 sont absents ;
C2 : l'adresse de l'instruction de branchement est rangée dans CO (l'instruction à cette adresse devient donc la prochaine instruction à exécuter).



JARCHI permet de simuler l'exécution d'un programme sur une architecture avec un accumulateur. La machine est minimum et comporte N mots de 4 caractères, d'adresses 0 à N. Le programme doit commencer à l'adresse 0 (initialisation du CO à 0).
Chaque instruction comporte un code sur 2 caractères et éventuellement une adresse sur 2 caractères.

Codes	Instructions
10	chargement dans ACC du mot indiqué (ex : 1025 pour le chargement dans ACC du mot 25)
20	rangement de ACC dans le mot mémoire indiqué
30	addition de ACC et du mot mémoire indiqué
40	comparaison de ACC et du mot mémoire indiqué ; le drapeau reçoit 0,1,2 selon que ACC est =, >, < au mot mémoire
50	saut inconditionnel à l'adresse indiquée (ex : 5012)
51	si drapeau = 0, saut à l'adresse indiquée
52	si drapeau = 1, saut à l'adresse indiquée
53	si drapeau = 2, saut à l'adresse indiquée
99	arrêt du programme

On étudiera au chapitre suivant une machine simulée avec un langage machine plus réaliste.

Exercices

16. Écrire un programme qui additionne trois données et range le résultat dans un 4e mot de la mémoire. Simuler le avec JARCHI.

17. Même question avec cinq données.

3.4.6 Les bus

Un bus est une voie de communication connectant plusieurs composants. C'est un support de transmission partagé.
Les bus internes connectent les composants dans l'unité centrale. Ils contiennent de quelque dizaines à plusieurs centaines de lignes par lesquelles transitent en parallèle les bits. Il y a des lignes de données, des lignes d'adresses et des lignes pour signaux de contrôle (ou micro-commandes).
Un signal émis par un composant peut être reçu par tous les autres composants connectés au bus. Par contre, il faut interdire à deux composants d'émettre simultanément un signal car ceux-ci pourraient interférer. Une méthode d'arbitrage évitant cette situation doit être mise en place. Cet arbitrage est réalisé soit par un composant spécialisé (le contrôleur de bus), dans le cas d'un arbitrage centralisé, soit par la coopération des composants connectés au bus, dans le cas d'un arbitrage décentralisé.
Il y a de nombreux types de bus organisés selon différentes architectures de bus. On trouve en général le bus local entre le processeur et le cache, très rapide, le bus système entre la mémoire centrale et le cache, assez rapide, et des bus d'entrées/sorties ou bus d'extension entre le bus système et les périphériques, plus lents.
Parmi les bus d'entrées/sorties courants sur les PC on peut citer les bus AGP et PCI-EXPRESS pour les cartes graphiques, les bus PCI pour les cartes réseaux et son, les bus ATA, SATA, SCSI pour les disques, les bus USB et *Firewire* pour les périphériques externes.

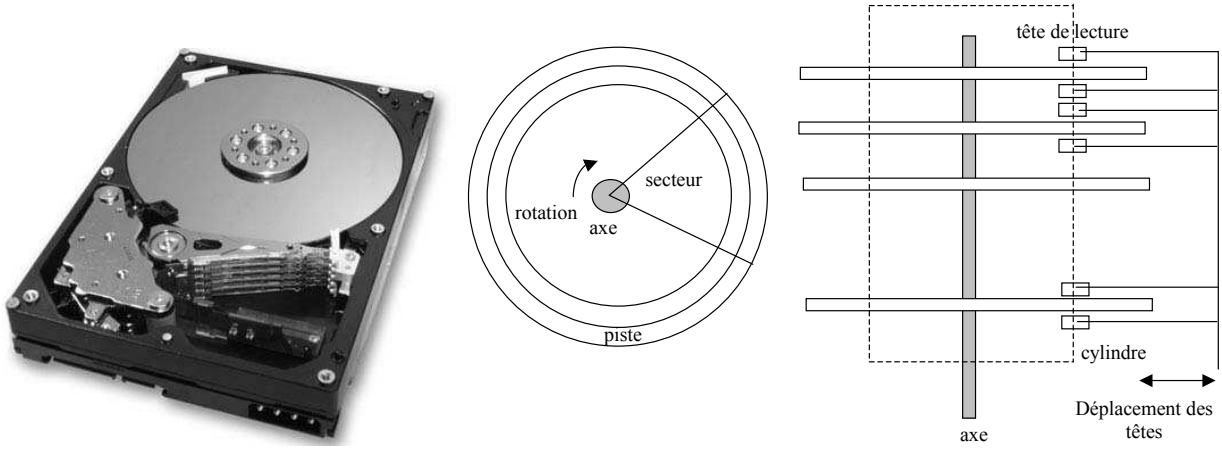
3.4.7 Les périphériques

La description technologique de tous les périphériques sort du cadre de ce cours. Nous nous limitons à quelques informations de base concernant les disques durs qui sont importantes à connaître lorsque l'on administre une machine.

Une unité de disque dur est constituée de plusieurs disques, ou plateaux, en aluminium ou en verre empilés et en rotation rapide autour d'un même axe (ex : 10000 tours/minute). Chaque face d'un plateau est lue ou écrite par une tête de lecture. Afin de simplifier le mécanisme, toutes les têtes se déplacent en même temps, radialement, sur un mécanisme en forme de peigne.
Les disques sont structurés en pistes et en secteurs. Le nombre de pistes est fixé par la densité trans-

versale (nombre de pistes par unité de longueur radiale). Cette densité dépend essentiellement de la précision du positionnement des têtes sur le disque.
Chaque piste ou secteur contient le même nombre d'octets (avec une densité qui augmente de l'extérieur vers le centre) ou non. L'unité de lecture ou d'écriture sur le disque est le secteur (capacité typique 512 octets). Le contrôleur du disque doit être capable d'écrire ou de lire n'importe quel secteur.
Pour repérer un secteur, il faut connaître son plateau, le numéro de sa piste, et le numéro du secteur dans la piste. La plupart des systèmes introduisent la notion de cylindre, formé par l'ensemble des pistes de même position sur tous les plateaux. Un secteur est alors repéré par :

- un numéro de cylindre (distance tête-axe de rotation) ;
- un numéro de piste (numéro de la tête de lecture à utiliser) ;
- un numéro du secteur (lié à l'angle).



Le temps d'accès pour lire ou écrire un secteur du disque dur dépend de la vitesse de rotation du disque, de la vitesse de déplacement des têtes et de la dimension du disque.
Chaque transfert (lecture ou écriture d'un secteur) demande les opérations suivantes :

- si les têtes ne sont pas sur le bon cylindre, déplacement des têtes ; on définit le temps de positionnement moyen (parcours en moyenne de la moitié du rayon),
- attente de l'arrivée du début du secteur visé sous la tête de lecture ; en moyenne, il faut que le disque tourne d'un demi-tour ; ce temps est appelé demi délai rotationnel,
- transfert des données, qui dure le temps nécessaire pour faire défiler le secteur entier sous la tête de lecture.

Le débit d'information maximal est déterminé par la vitesse de rotation du disque et la densité d'enregistrement longitudinale. Il est parfois limité par le débit du bus d'entrées/sorties reliant le disque à l'ordinateur. Les fabriquants de disques durs indiquent en général le temps d'accès moyen et le taux de transfert maximum (débit).
Le rangement des fichiers sur le disque dépend du système de gestion des fichiers du système d'exploitation.
Avant utilisation un disque dur doit être formaté.

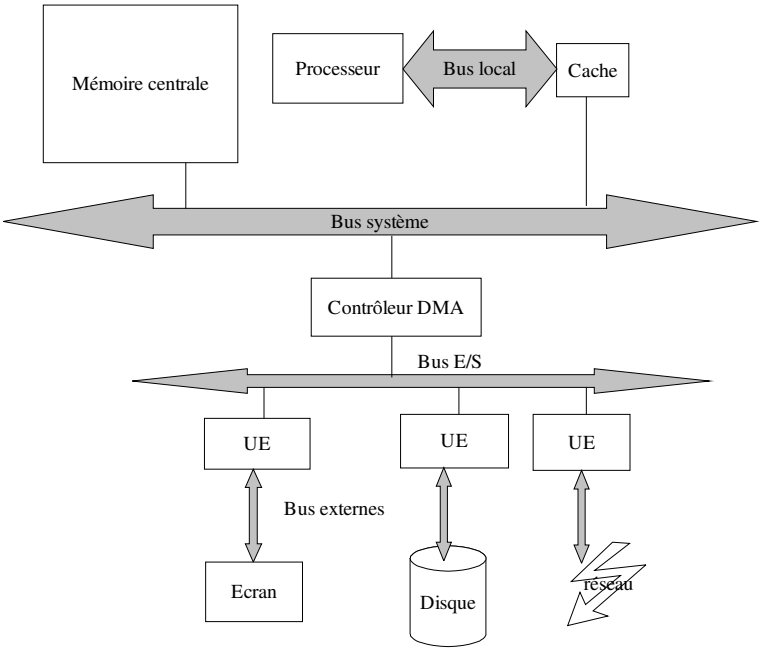
1. Formatage de bas niveau : les pistes du disque sont divisées en un nombre donné de secteurs.
2. Partitionnement : il permet de diviser le disque dur en plusieurs zones appelées partitions pouvant être gérées par différents système d'exploitation. Il est possible d'installer des partitions pour plusieurs systèmes, comme Windows (systèmes de fichiers FAT32 ou NTFS) et Linux (systèmes de fichiers ext2 et ext3).
3. Formatage de haut niveau (ou formatage logique) : pendant cette phase, le système d'exploitation écrit sur le disque les structures nécessaires pour gérer les fichiers et les données (secteur amorce, tables d'allocation, répertoires racines par partition ...).

3.4.8 Les unités d'échange

Il existe une grande variété de périphériques avec des vitesses, des débits et des formats de données très différents. Tous ces périphériques ont un point commun : ils sont très lents par rapport au processeur et à la mémoire (de quelques nanosecondes pour le processeur à des millisecondes ou secondes pour les périphériques). Il faut donc des unités d'échange (ou contrôleurs d'entrées/sorties) entre processeur et périphériques pour gérer leur coordination, le transcodage, la détection des erreurs, etc. Il existe trois techniques de base pour gérer les entrées/sorties.

1. La scrutation : le processeur interroge l'unité d'échange pour savoir si des transferts (entrées ou sorties) sont possibles. Tant que ces transferts ne sont pas possibles, le processeur attend puis redemande (boucle de scrutation). L'inconvénient majeur est que le processeur se retrouve souvent en phase d'attente. Il est complètement occupé par la réalisation de l'entrée/sortie. De plus, l'initiative de l'échange de données revient au programme. Il peut donc arriver que des entrées ne soient pas traitées immédiatement car le programme ne se trouve pas encore dans la boucle de scrutation. Ce type d'entrée/sortie est très lent.
2. Les interruptions : une interruption est un signal pouvant être émis par un dispositif externe au processeur. Le processeur possède une ou plusieurs entrées réservées à cet effet. L'interruption peut stopper le travail courant du processeur pour forcer l'exécution d'un programme traitant la cause de l'interruption.
   Dans une opération d'entrée avec interruption, le processeur exécute sa requête de lecture de données et reprend l'exécution d'un programme (le même ou un autre). Il n'attend donc plus. Quand l'unité d'échange peut récupérer les données elle envoie une interruption au processeur. Avant chaque exécution d'une instruction, le processeur examine s'il y a eu une requête d'interruption. Si c'est le cas, il interrompt le programme en cours et sauvegarde son état (registres, compteur ordinal, registre d'état...). Ensuite, il exécute le programme d'interruption (c'est-à-dire réalise la lecture des données et leur transfert en mémoire) puis restitue l'état sauvegardé avant de reprendre le programme interrompu.
3. L'échange direct avec la mémoire : ce mode permet le transfert de blocs de données entre la mémoire et un périphérique sans passer par le microprocesseur. Pour cela, un circuit appelé contrôleur de DMA (*Direct Memory Access*) prend en charge entièrement le transfert des blocs de données. Le microprocesseur doit tout de même :
   – initialiser l'échange en donnant au DMA l'identification du périphérique concerné,
   – donner le sens du transfert (entrée ou sortie),
   – fournir l'adresse du premier et du dernier mot concernés par le transfert.
   Le contrôleur de DMA est doté d'un registre d'adresse, d'un registre de donnée, d'un compteur et d'un dispositif de commande (c'est un vrai petit processeur spécialisé). Pour chaque mot échangé, le DMA demande au microprocesseur le contrôle du bus, effectue la lecture ou l'écriture mémoire à l'adresse contenue dans son registre et libère le bus. Il incrémente ensuite cette adresse et décrémente son compteur. Lorsque le compteur atteint zéro, le dispositif informe le processeur de la fin du transfert par une interruption.
   Le principal avantage est que, pendant toute la durée du transfert d'un bloc de données, le processeur est libre d'effectuer un autre traitement. La seule contrainte est une limitation de ses propres accès mémoire pendant toute la durée de l'opération, puisqu'il doit parfois retarder certains de ses accès pour permettre au dispositif d'accès direct à la mémoire d'effectuer les siens (arbitrage pour l'accès au bus).

La figure suivante donne une architecture de machine possible avec les différents types de bus et le contrôleur DMA.



3.4.9 Les interruptions

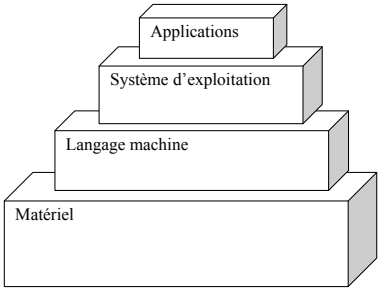
Outre les interruptions liées aux entrées/sorties (terminaison et erreur), il existe d'autres types d'interruptions :

- défaillance matérielle (ex : erreur mémoire, problème d'alimentation électrique ...),
- défaillance logicielle (ex : division par zéro, accès à une zone mémoire protégée ...),
- signal d'un temporisateur (exécution d'une fonction du système d'exploitation à certains instants), etc.

Il est possible qu'une demande d'interruption intervienne pendant l'exécution d'un autre programme d'interruption. Des priorités sont donc associées aux interruptions. L'interruption qui arrive est prise en compte si elle a une priorité supérieure à celle en cours. Ce mécanisme permet de gérer les urgences différentes des interruptions.

À l'occasion d'une entrée/sortie, nous avons vu qu'un programme bloqué en attente de données par exemple peut être remplacé par un autre. Cette idée de partage du processeur et de la mémoire par plusieurs programmes est mise en œuvre par le système d'exploitation (unix, windows ...).

Le système d'exploitation est un programme chargé de faciliter l'utilisation de l'ordinateur et d'en optimiser le fonctionnement. Il gère en particulier les ressources (mémoires, processeur, entrées/sorties) pour les partager entre plusieurs programmes en cours d'exécution (processus). L'étude des systèmes d'exploitation sort du cadre du cours d'architecture des ordinateurs et fera l'objet de cours spécifiques ultérieurement.



3.5 Les architectures avancées : architectures pipelinées, superscalaires et multi-core

Comme nous l'avons vu, le traitement d'une instruction comprend plusieurs phases. Au lieu de les exécuter séquentiellement on peut chercher à les effectuer en parallèle pour plusieurs instructions successives. Supposons que les phases correspondent aux trois étapes chercher (CH), décoder (DE), exécuter (EX). Dans une machine pipelinée on aura donc :

Instructions/étages	1	2	3	4	5	6	7	8	9
instr_i	CH	DE	EX						
instr_{i+1}		CH	DE	EX					
instr_{i+2}			CH	DE	EX				

au lieu de :

Instructions/étages	1	2	3	4	5	6	7	8	9
instr_i	CH	DE	EX						
instr_{i+1}				CH	DE	EX			
instr_{i+2}							CH	DE	EX

Le gain de rapidité est dû à l'augmentation du flux des instructions traitées et non pas à la rapidité de traitement de chaque instruction. Il faut noter que chaque étage du pipeline a comme durée celle de la phase la plus longue (car les trois s'exécutent en parallèle). L'efficacité du pipeline dépend aussi de la manière dont le programme est écrit. Par exemple, si une instruction de calcul a besoin du résultat de l'instruction de calcul juste avant, le pipeline est en défaut. Les calculs doivent être optimisés par le programmeur ou le compilateur. De plus, en cas de branchement conditionnel, on peut commencer le traitement des instructions qui suivent de manière inutile. Dans certains processeurs on tente de prédire si le branchement va avoir lieu ou non en fonction d'une table des exécutions passées des branchements (« anticipation de branchement »).

Les processeurs « superscalaires » possèdent plusieurs pipelines qui fonctionnent en parallèle. Un superscalaire de degré n possède n pipelines parallèles. On a donc du parallélisme entre instructions en plus du recouvrement des phases à l'intérieur des instructions.
Ce parallélisme est complexe à gérer à cause des synchronisations nécessaires. En outre, il n'est pas toujours possible d'utiliser à plein tous les pipelines.
Exemple d'exécution sur un superscalaire de degré 4 :

Instructions/étages	1	2	3	4	5
instr_i	CH	DE	EX		
instr_{i+1}	CH	DE	EX		
instr_{i+2}	CH	DE	EX		
instr_{i+3}	CH	DE	EX		
instr_{i+4}		CH	DE	EX	
instr_{i+5}		CH	DE	EX	
instr_{i+6}		CH	DE	EX	
instr_{i+7}		CH	DE	EX	
instr_{i+8}			CH	DE	EX
instr_{i+9}			CH	DE	EX
instr_{i+10}			CH	DE	EX
instr_{i+11}			CH	DE	EX

Par exemple, le Pentium 4 est un microprocesseur de type superscalaire avec :
- un jeu d'instructions très large avec des formats d'instruction variables,
- chaque instruction est traduite en une ou plusieurs micro-opérations de taille fixe,
- le processeur exécute les micro-opérations avec une organisation à plusieurs pipelines de 20 étages (soit 20 étapes par micro-opération !).

Les approches *multi-core* consistent à mettre plusieurs processeurs sur la même puce. Actuellement on trouve des processeurs *dual-core* ou *quadi-core* mais à l'avenir le nombre de processeurs intégrés pourra augmenter. Cela permet d'exécuter des applications en parallèle ou des *threads* en parallèle dans une même application.

3.6 La définition et la mesure des performances

a) Définition théorique

La vitesse de l'horloge cadence les instructions. A partir de là, on peut calculer le temps CPU pour un programme P comme le produit entre le nombre de cycles nécessaires pour le programme P par le temps de cycle d'horloge TCH. Pour déterminer le nombre de cycles nécessaires pour le programme P, on peut multiplier le nombre d'instructions exécutées par le processeur NI par le nombre moyen de cycles par instruction pour l'architecture considérée CPI :

$$\text{temps CPU} = NI \times CPI \times TCH$$

Pour réduire le temps CPU on peut soit augmenter la fréquence de l'horloge (mais il existe une limite matérielle à l'amélioration de la technologie), soit réduire le CPI (choix du jeu d'instructions ou accès mémoire plus rapides), soit réduire le nombre d'instructions NI (compilateur optimisant).

On peut calculer les MIPS (Millions d'Instructions Par Seconde) par :

MIPS $= FH/CPI$, où FH est la fréquence d'horloge en MHz (millions de cycles par seconde).

Sur le même modèle existent également les MegaFlops (millions d'instructions flottantes — sur des réels — par seconde).
Les MIPS et les MegaFlops calculés ne permettent pas réellement de comparer les machines entre elles car ils dépendent de la « quantité de travail » effectuée par les instructions qui peut varier beaucoup entre les architectures CISC (instructions complexes) ou RISC (instructions simples).
De plus, les performances globales d'une machine dépendent non seulement de l'organisation du processeur mais aussi de tous les composants qui y sont reliés : chemins de données (bus), interfaces, mémoire centrale et unités de stockage, ainsi que du système d'exploitation.

b) Mesure pratique

L'idée est de mesurer le temps d'exécution de programmes réels. Mais ce temps d'exécution peut se définir de plusieurs manières, avec des résultats très différents selon le choix qui est fait. On peut distinguer par exemple :

- le temps total pour terminer la tâche (en incluant les accès disques et mémoire, les opérations d'entrées-sorties),
- le temps CPU (processeur) utilisé pour la tâche,
- le temps CPU utilisé par le système d'exploitation pour cette la tâche, etc.

La commande *time* sous Unix permet d'avoir des informations sur ces durées sous la forme :

```
29.090 user  13.140 system  2:08:38.09 elapsed  0.5 %CPU
```

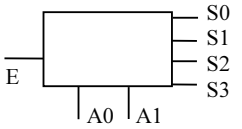
Le temps CPU pour la tâche est de 29.09 s. Le temps CPU pour le système est de 13.14 s. Le temps total est 7718 s (2 h 8 mn 38 s), d'où un temps d'occupation du processeur de $(29.09 + 13.40)/7718$ soit 0.5% seulement pour cette tâche.

En pratique, l'approche la plus réaliste pour mesurer les performances consiste à réaliser des tests (*benchmarks*). Ils comparent les machines par types d'applications (calculs numériques, jeux, bureautique, etc.) en utilisant des programmes et des données communes. Les résultats de certains tests sont publiés et permettent des comparaisons à peu près fiables entre machines. C'est le cas par exemple des test du SPEC (*Standard Performance Evaluation Corporation* `http://www.spec.org`) utilisés par beaucoup de constructeurs.

Exercices récapitulatifs (extraits de DS)

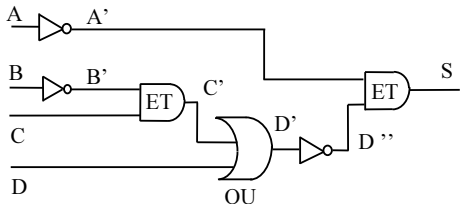
18. *Démultiplexeur*
On appelle démultiplexeur un circuit qui transmet le bit du fil d'entrée (E) sur un des fils de sortie (S0, S1, S2, S3) en fonction de l'adresse fournie sur les fils d'adresse (A0, A1).
Exemple : démultiplexeur 2 bits dont la table de vérité est la suivante.

E	A0	A1	S0	S1	S2	S3
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1



a) Donnez les équations logiques de S0, S1, S2, S3,
b) Construisez, par la méthode systématique, le circuit correspondant.

19. *Table de vérité*
Donnez la table de vérité du circuit suivant.



20. *Codification et circuit logique*
a) Calculez les compléments à 2 des nombres de 3 bits du tableau suivant :

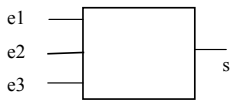
nombres	compléments à 2
000	
001	
010	
011	
100	
101	
110	
111	

b) On veut construire le circuit logique qui calcule le complément à 2 de 3 bits. On considère le tableau de la question a) comme sa table de vérité en appelant e1, e2 et e3 les entrées et s1, s2 et s3 les sorties.
Donnez les équations logiques de s1, s2 et s3.
c) Construisez par la méthode systématique vue en cours ce circuit logique (avec des portes ET, OU, NON).
d) Factorisez les équations logiques pour faire apparaître des OU exclusifs ($a.\bar{b} + \bar{a}.b$).
Dessinez le circuit simplifié (avec des portes ET, OU, NON, OU EXCLUSIF).

21. *Bit majoritaire*
a) Soit le circuit à trois entrées qui donne en sortie le bit majoritaire sur les trois entrées. Donnez

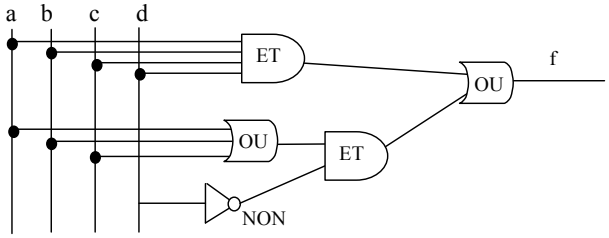
l'équation logique non simplifiée et dessinez le circuit correspondant avec des portes ET, OU, NON (les portes ET et OU pouvant comporter plus de deux entrées).
Le circuit est défini par la table de vérité ci-dessous :

e1	e2	e3	s (majorité de 0 ou de 1)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



b) Proposez un circuit réalisant la même fonction et utilisant moins de portes logiques. Vous pouvez utiliser le ET, le OU, le OU exclusif, et le NON.

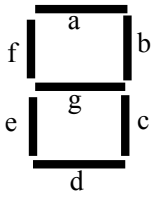
22. *Circuit programmable*
Selon la valeur de d (0 ou 1), le circuit suivant réalise deux fonctions différentes. On parle parfois de "circuit programmable". A l'aide des tables de vérité ou des fonctions logiques, démontrez quelles sont ces deux fonctions.

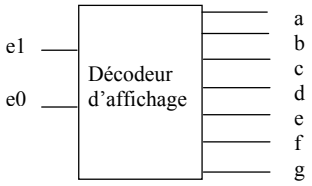


23. *Afficheur*
On considère un décodeur d'affichage permettant d'afficher les chiffres de 0 à 3 sur un ensemble de diodes électroluminescentes. Chaque diode correspond à un segment. Sept segments (sept diodes) a, b, c, d, e, f, g permettent d'afficher un chiffre.
Les chiffres sont affichés en allumant certains segments :

- a, b, c, d, e, f pour 0,
- b, c pour 1,
- a, b, g, e, d pour 2,
- a, b, g, c, d pour 3.

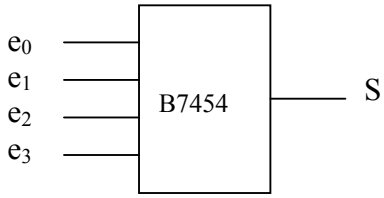
Le décodeur d'affichage reçoit en entrée le chiffre en binaire (poids faible dans e0). Il émet en sortie des 1 dans les fils des segments à allumer.





a) Donnez la table de vérité du décodeur d'affichage.
b) Donnez les sept équations logiques de a, b, c, d, e, f, g.
c) Simplifiez les sept équations logiques.
d) Dessinez le circuit du décodeur d'affichage à partir des équations simplifiées.

24. *Circuit BT7454*
Le circuit B7454 a 4 entrées e_0, e_1, e_2, e_3 et une sortie s.



L'équation logique de ce circuit est la suivante :

$$s = \overline{e_0.e_1 + e_2.e_3}$$

a) Donnez la table de vérité du circuit.
b) Dessinez le schéma du circuit à partir de son équation logique.
c) Soit la fonction g à six variables (a, b, c, d, e, f) :

$$g = \bar{a}.\bar{c}.\bar{e} + \bar{a}.d.\bar{e} + b.\bar{c}.\bar{e} + b.d.\bar{e} + f.b$$

c1) Simplifiez l'équation g.
c2) Construisez le schéma de g en utilisant le circuit B7454.

25. *Code de Gray*
Le code de Gray est un codage binaire des entiers naturels tel que l'on passe d'un entier au suivant en ne modifiant qu'un bit.
Exemple : code de Gray sur 3 bits

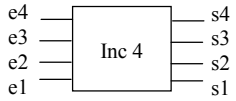
entier	code
0	000
1	001
2	011
3	010
4	110
5	111
6	101
7	100

Soit la table de vérité suivante avec à gauche les entiers binaires et à droite les codes de Gray correspondants :

b1	b2	b3	g1	g2	g3
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

a) Donnez les équations logiques de g1, g2, g3.
b) Simplifiez les équations logiques (rappel : le OU exclusif $a \oplus b = a.\bar{b} + \bar{a}.b$).
c) Dessinez le circuit qui traduit un entier binaire dans le code de Gray correspondant (à partir des équations simplifiées).

26. *Incrémenteur 4 bits*
On veut réaliser un incrémenteur 4 bits. Ce circuit, plus simple qu'un additionneur, ajoute 1 au nombre binaire en entrée (en cas de débordement le bit qui sort est oublié).



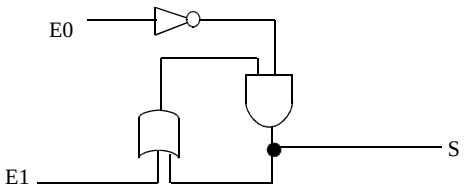
a) Donnez la table de vérité sous la forme :

e4	e3	e2	e1	s4	s3	s2	s1
..	..	..	..	..	..	..	..

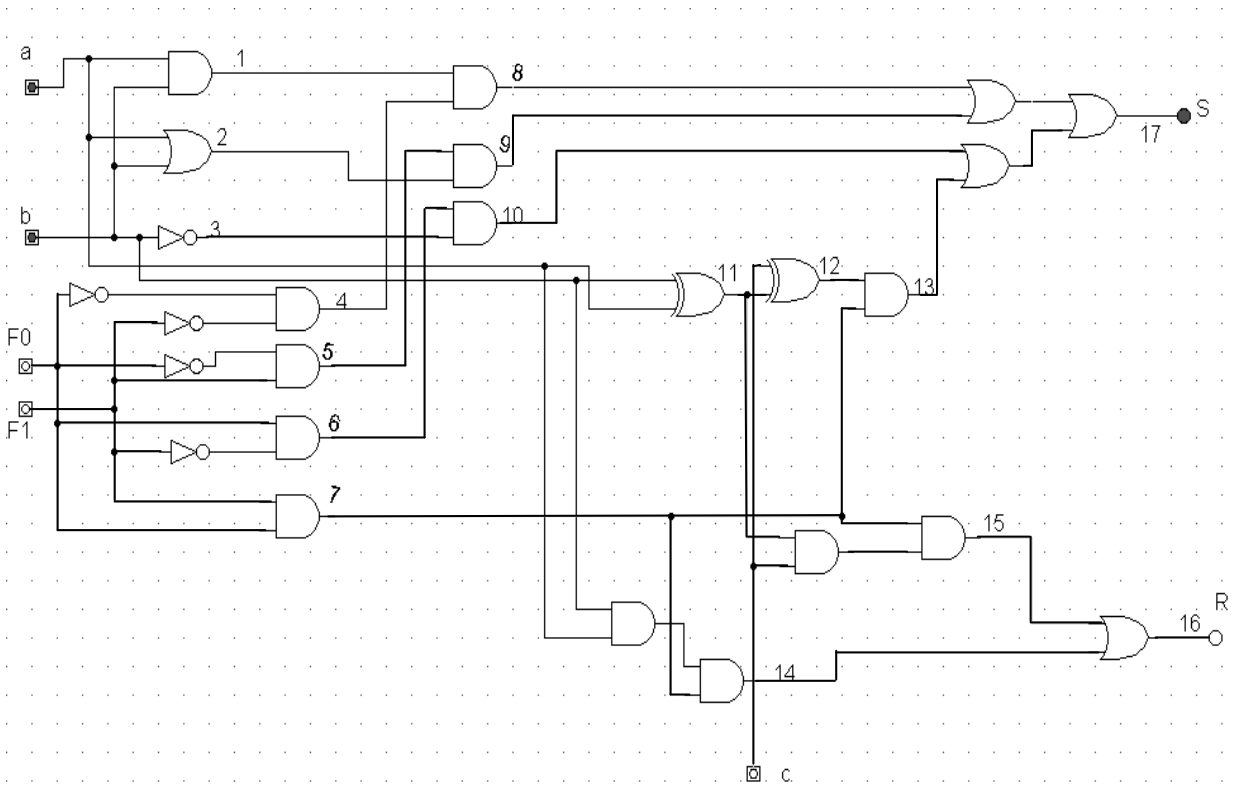
b) Au vu de la table de vérité et sans faire aucun calcul, déterminez l'équation logique simplifiée de s1.
c) Montrez avec des tables de vérité que :
$s2 = e2 \oplus e1$ (où $\oplus$ représente le ou exclusif)
$s3 = e3 \oplus (e2.e1)$
$s4 = e4 \oplus (e3.e2.e1)$
d) Dessinez le circuit à partir de ces équations logiques.

27. *Circuit avec décodeur*
a) On dispose d'un décodeur pour des codes sur 3 bits. Utilisez-le pour construire la valeur 1 si et seulement si l'entrée sur 3 bits représente un entier ≥ 5 et la valeur 0 dans le cas contraire.
b) Obtenez le même résultat avec un circuit construit directement avec des portes ET, OU et NON.

28. *Circuit mystère (1)*
Donnez la table de vérité de ce circuit (S_{t+1} en fonction de $E0$, $E1$ et S_t). De quel circuit vu en cours ce circuit se rapproche-t-il le plus ? Quels autres noms peut on donner aux entrées $E0$ et $E1$?



29. *Circuit mystère (2)*
a) Donnez l'expression logique des valeurs à la sortie de chaque porte (numérotées de 1 à 17 sur le dessin du circuit).
b) Simplifiez R quand c vaut 0 et quand c vaut 1.
c) Si a, b et c sont des données et F0 et F1 un code, dites quelles opérations sont effectuées par ce circuit. Pour F0=1 et F1=1 vous pouvez construire la table de vérité donnant S et R en fonction de a, b et c.


a
b
F0
F1
c
S
R
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17

4 Le « langage machine »

4.1 Définitions

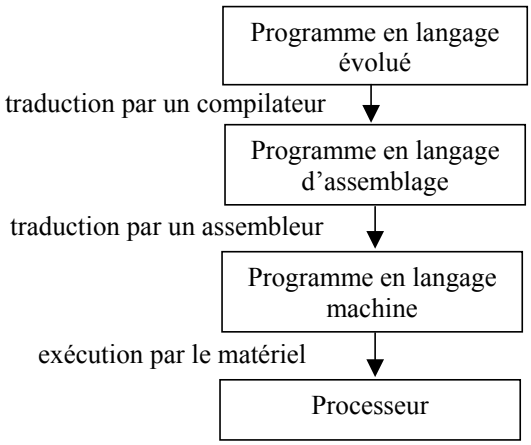
Le langage machine est constitué des instructions binaires directement interprétables par l'unité de commande d'un type de processeur donné. Il est donc propre à chaque type de processeur, puisque directement lié aux circuits qui le composent. Chaque instruction comporte en général un code opération binaire et des adresses binaires de registres et/ou de mots mémoire.

On appelle langage d'assemblage une version « codée en clair » du langage machine, où les codes opérations sont remplacés par des mnémoniques et où les adresses peuvent être remplacées par des identificateurs.
Exemple :

Langage	Code opération	Registre	Adresse mot mémoire
langage machine	00110011	0111	0011111000000001100
langage d'assemblage	ADD	AL,	VALEUR

Le programme de traduction qui convertit le langage d'assemblage en langage machine s'appelle un assembleur. Le langage d'assemblage n'est plus utilisé que dans des circonstances particulières (pour une efficacité maximum par contrôle direct de la machine) ou sur des processeurs très élémentaires pour lesquels il n'existe pas de traducteurs de langages évolués en langage machine. Néanmoins une connaissance minimale des langages de bas niveau permet de mieux comprendre certaines caractéristiques des langages évolués. C'est pourquoi nous le présentons dans ce cours.

La majorité des langages évolués (comme C, C++, COBOL) sont traduits en langage d'assemblage par un compilateur. Chaque instruction du langage évolué donne une suite plus ou moins complexe (et plus ou moins optimisée) d'instructions en langage d'assemblage.



Ci dessous trois exemples de traduction d'une itération par un compilateur sur des processeurs différents (Intel pour PC, Motorola pour Mac, SPARC).

```
L'itération (C):  processeur Intel :    processeur Motorola :   processeur SPARC :
s=0;                xorl  %edx,%edx       clrl   d1               mov  0,%o1
i=1;                movl  $1,%eax         moveq  #1,d0             mov  1,%o0
do{               L5:                   L5:                    L5:
  s=s+2;            addl  $2,%edx         addql  #2,d1             add  %o0,1,%o0
  i=i+1;            incl  %eax            addql  #1,d0             cmp  %o0,9
} while(i>9);       cmpl  $9,%eax         moveq  #9,d2             ble  L5
                    jle   L5              cmpl   d0,d2             add  %o1,2,%o2
                                          jge    L5$
```

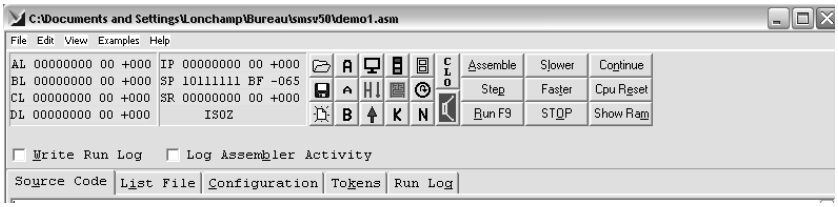
On note que la structure reste en gros constante mais que certains détails diffèrent (manière d'initialiser à 0, nombre d'opérandes pour les additions, forme et sens de la comparaison et branchement vers le début de l'itération ...). À noter pour le SPARC la notion de « branchement retardé » : l'instruction qui suit le branchement est toujours exécutée avant que le branchement soit réalisé !

Certains langages évolués ne sont pas compilés mais interprétés (comme Basic, PHP, javascript, ...). L'interpréteur est un programme qui traduit en langage machine chaque instruction du programme en langage évolué et l'exécute immédiatement. Une même instruction peut donc être traduite plusieurs fois, ce qui rend l'exécution moins rapide.
Enfin, certains langages récents (comme Java, C#) sont compilés dans un langage intermédiaire indépendant de la machine (appelé *bytecode*), avant d'être interprétés par un programme propre à chaque type de processeur (appelé « machine virtuelle »).

Nous présentons la programmation de bas niveau en langage d'assemblage sur une machine simulée qui simplifie beaucoup de détails.

4.2 Présentation de la machine simulée pour l'apprentissage du langage d'assemblage

La machine simulée sms32v50 est distribuée sous licence GNU GPL (`www.softwareforeducation.com`).
La mémoire centrale comporte 256 mots de un octet. Les adresses vont de 0 à 255 et sont notées [00] à [FF] en hexadécimal. Les crochets distinguent les adresses des constantes hexadécimales.
Ex : 1F est la constante 31 alors que [1F] repère le mot d'adresse 31.
Les mots d'adresse C0 à FF sont utilisés pour afficher automatiquement des codes ASCII dans une fenêtre de résultat (les mots sont initialisés avec des valeurs ASCII 20 en hexadécimal c'est à dire des espaces). On se limitera donc aux adresses [00] à [BF].
La machine ne comporte pas un accumulateur unique, mais 4 registres de un octet utilisables indifféremment et notés AL, BL, CL, et DL. Leurs contenus sont affichés en binaire, en hexadécimal et en décimal par le simulateur.
On visualise également le compteur ordinal (*Instruction Pointer* ou IP), le pointeur de pile (*Stack Pointer* ou SP) et le mot d'état (*Status Register* ou SR) avec un ensemble de drapeaux ou *flags* : *sign* (S) et *zero* (Z) pour les comparaisons, *overflow* (O) pour les débordements.
Le compteur ordinal est initialisé à 0. Les mots peuvent contenir des entiers entre -128 et $+127$ ou des codes ASCII.



L'interface sous Windows comporte plusieurs boutons :
- *Assemble* pour traduire l'assembleur en code machine,
- *Step* pour exécuter le programme en pas à pas,
- *Run* pour exécuter le programme d'un coup,
- *Slower* pour ralentir l'exécution,
- *Faster* pour l'accélérer,
- *Stop* pour l'arrêter,
- *Continue* pour la reprendre,
- *Cpu reset* pour remettre le compteur ordinal à 0,
- *Show Ram* pour visualiser la mémoire centrale.

L'interface comporte aussi des « périphériques » que l'on peut piloter : des feux rouges, un ascenseur, un moteur ...).

4.3 Instructions de base du langage de la machine simulée

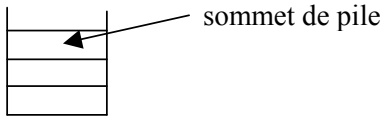
a) Instruction de transfert (mémoire à registre ou registre à mémoire)

```
MOV destination, source
```

La source est recopiée dans la destination *(attention à l'ordre !)* Les seules formes autorisées réalisent des transferts de constante vers registre, de mot mémoire vers registre et de registre vers mot mémoire :

Assembleur	Langage machine	Explication
MOV AL, 1B	D0 00 1B (3 mots)	AL = 1B range la constante hexadécimale 1B dans AL
MOV BL, [C2]	D1 01 C2	BL = [C2] copie le mot d'adresse C2 dans BL
MOV [15], CL	D2 15 02	[15] = CL copie CL dans le mot d'adresse 15

Les transferts de mémoire à mémoire sont interdits car il faudrait deux adresses mémoire pour une seule instruction (cf. paragraphe précédent sur l'UAL.). Les transferts entre registres sont aussi interdits sur cette machine simulée contrairement à beaucoup de machines réelles. Il faut passer par un mot mémoire intermédiaire ou par la pile.
La pile est une zone de mémoire où l'on empile et dépile des valeurs. À tout instant le registre SP pointe sur le sommet de la pile. Les instructions PUSH et POP permettent d'empiler et de dépiler en sommet de pile.



Assembleur	Langage machine	Explication
PUSH BL	E0 01	ajoute le contenu de BL en sommet de pile
POP CL	E1 02	retire le sommet de pile et le range dans CL

Le transfert du registre AL au registre BL peut donc s'écrire :

```
PUSH AL     ; empile la valeur de AL au sommet de la pile
POP BL      ; dépile cette valeur et la range dans BL
```

b) Instructions de calcul

Il s'agit de l'addition, de la soustraction, de la multiplication, de la division entière, du reste de la division entière ou modulo, de l'incrémentation de un, de la décrémentation de un. Elles utilisent des registres et rangent leur résultat dans le premier registre.

Assembleur	Langage machine	Explication
ADD AL, BL	A0 00 01 (3 mots)	AL = $AL + BL$
SUB BL, CL	A1 01 02	BL = $BL - CL$
MUL CL, DL	A2 02 03	CL = $CL * DL$
DIV DL, AL	A3 03 00	DL = DL/AL
MOD AL, BL	A6 00 01	AL = AL mod BL
INC DL	A4 03 (2 mots)	DL = $DL + 1$
DEC AL	A5 00	AL = $AL - 1$

Dans les cinq instructions à deux opérandes, l'opérande de droite peut aussi être une constante hexadécimale (codes machine Bn au lieu de An).

c) Instructions de comparaison

```
CMP destination,  source
```

Compare la source et la destination et met à jour des bits particuliers appelés drapeaux (ou *flags* en anglais). Plus précisément, soustrait la source à la destination et met 1 dans le drapeau Z (*zero*) si le résultat est nul et 1 dans le drapeau S (*signe*) si le résultat est négatif.

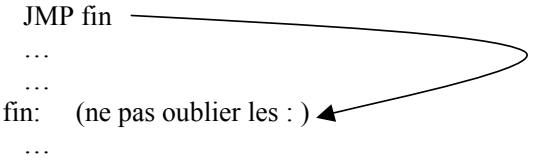
Assembleur	Langage machine	Explication
CMP AL, BL	DA 00 01 (3 mots)	met Z à 1 si AL = BL met S à 1 si AL < BL
CMP BL,13	DB 01 13	met Z à 1 si BL = 13 met S à 1 si BL < 13
CMP CL, [20]	DC 02 20	met Z à 1 si CL = [20] met S à 1 si CL < [20]

d) Instructions de branchement inconditionnel

```
JMP  étiquette_de_destination
```

L'étiquette vers laquelle se fait le saut occupe une ligne du programme assembleur mais ne prend pas de place en mémoire. Elle représente une certaine adresse. Elle ne doit pas comporter de caractères accentués.

Assembleur	Langage machine	Explication
JMP HERE	C0 12	augmente IP de 12 (l'étiquette HERE est 12 mots après l'instruction courante)



e) Instructions de branchement conditionnel

```
JZ (ou JNZ ou JS ou JNS) étiquette_de_destination
```

Une instruction de branchement conditionnel suit en général une instruction de comparaison (CMP) et réalise le saut en fonction de la valeur des drapeaux, c'est-à-dire en fonction du résultat de la comparaison. Si le saut n'est pas possible le processeur passe à l'instruction suivante.
Il y a quatre branchements conditionnels :

- JZ pour *Jump if Zero* qui réalise le saut si Z = 1,
- JNZ pour *Jump if Not Zero* qui réalise le saut si Z = 0,
- JS pour *Jump if Sign* qui réalise le saut si S = 1,
- JNS pour *Jump if Not Sign* qui réalise le saut si S = 0.

Assembleur	Langage machine	Explication
JZ A	C1 09	augmente IP de 9 si Z est à 1 (l'étiquette A est à 9 mots après l'instruction courante)
JNZ PLACE	C2 04	augmente IP de 4 si Z est à 0
JS R	C3 E1	diminue IP de 31 si S est à 1 (l'étiquette R est à 31 mots avant l'instruction courante)
JNS START	C4 04	augmente IP de 4 si S est à 0

En jouant sur l'ordre des registres on peut réaliser toutes les comparaisons ($>$, $<$, $\geq$, $\leq$, $=$, $\neq$) comme nous le détaillerons au paragraphe 4.4.

f) Instructions diverses

Assembleur	Langage machine	Explication
END	00	fin du programme (une seule sur la dernière ligne)
HALT	00	arrêt de l'exécution (éventuellement plusieurs)
DB constante	constante hexadécimale	définit une donnée (*DefineByte*)

g) Présentation d'un programme
Nous prenons l'habitude de placer les données en début de mémoire avec un JMP qui les saute pour aller au début du programme (car le compteur ordinal est initialisé à 0). En effet, si on place les données après le programme on ne peut connaître leurs adresses qu'après avoir écrit tout le programme ce qui n'est guère pratique. Notez les commentaires commençant par un ; qui suivent certaines instructions.

```
   JMP  debut      ; cette instruction occupe les mots 0 et 1
   DB    5         ; mot 2 (donnée 1)
   DB    2         ; mot 3 (donnée 2)
   DB    0         ; mot 4 (résultat)
debut:
   MOV AL, [2]     ; première instruction du programme
   MOV BL, [3]
   ADD AL, BL      ; résultat dans AL
   MOV [4],  AL
   END             ; les mots d'adresse 2 et 3 ont étés additionnés
                   ; et le résultat rangé dans le mot 4
```

Exercices

1. Écrivez un programme qui multiplie une donnée rangée dans le mot d'adresse 2 par 5 puis range le résultat dans le mot d'adresse 50.

2. Avec le programme précédent testez un débordement (*overflow*). On pourra prendre 20 (en hexadécimal) multiplié par 5.

4.4 Le schéma conditionnel

C'est le schéma :

```
SI condition ALORS instructions_du_cas_vrai SINON instructions_du_cas_faux FSI
```

où condition s'écrit : `opérande1 opérateur_de_comparaison opérande2`, avec $>$, $<$, $>=$, $<=$, $=$ ou $!=$ comme `opérateur_de_comparaison`.
En assembleur, il n'y a pas d'instruction conditionnelle « SI ALORS SINON ». Il faut la traduire avec des branchements conditionnels et inconditionnels. Le schéma de traduction standard est le suivant :

```
              CMP  opérande1, opérande2 ou opérande2, opérande1
              J??  vrai
              instructions_du_cas_faux
              JMP  fin
       vrai:
              instructions_du_cas_vrai
       fin:
              ...
```

où J ? ? est un des quatre branchements possibles (JZ ou JNZ ou JS ou JNS). Il y a 2 cas où il faut inverser opérande1 et opérande2 comme le montre le tableau suivant :

Condition	Traduction	Explication
a = b	CMP a, b suivi de JZ...	a-b = 0 <=> a = b
a != b	CMP a, b suivi de JNZ...	a-b != 0 <=> a != b
a < b	CMP a, b suivi de JS...	a-b < 0 <=> a < b
a >= b	CMP a, b suivi de JNS...	a-b >= 0 <=> a >= b
a > b	CMP b, a suivi de JS...	b-a < 0 <=> a-b > 0 <=> a > b
a <= b	CMP b, a suivi de JNS...	b-a >= 0 <=> a-b <= 0 <=> a <= b

Les compilateurs utilisent ce schéma pour traduire les conditionnelles en langage d'assemblage. Dans le cas d'un schéma sans partie sinon :

```
SI condition ALORS instructions_du_cas_vrai FSI
```

le schéma précédent est inutilement compliqué car `instructions_du_cas_faux` est vide. Il est conseillé de prendre la négation de la condition et de sauter à l'étiquette fin si cette négation est vraie (autrement dit « si la condition est fausse alors je ne fais rien »). Le schéma de traduction est le suivant :

```
     CMP  opérande1, opérande2 ou opérande2, opérande1
     J??  fin
     instructions_du_cas_vrai
fin:
     ...
```

où le choix de J ? ? et l'ordre des opérandes est donnée par le tableau précédent pour la négation de la condition.

Exercices

3. Écrivez un programme qui met dans le mot 4 la plus grande des valeurs des mots 2 et 3.

4. Écrivez un programme qui calcule la valeur absolue d'une donnée et la range dans le même mot.

4.5 Le schéma itératif

C'est le schéma :

```
POUR i DE 1 A limite FAIRE
   instructions_à_répéter
FPOUR
```

Il faut aussi le traduire avec des branchements conditionnels et inconditionnels. Le schéma de traduction standard est le suivant :

```
     MOV registre1, 1        ; registre compteur
     MOV registre2, limite
re:
     CMP registre2, registre1
     JS fin                  ; limite-compteur<0 <=> compteur>limite
     bloc_à_répéter
     INC registre1
     JMP re\textit{
fin:
```

Exercices

5. Écrivez un programme qui fait la somme des n premiers entiers ; n est dans le mot d'adresse 2 et le résultat est rangé dans le mot d'adresse 3.

```
somme <- 0
pour i de 1 à n faire
  somme <- somme + i
fpour
```

6. Écrivez un programme qui calcule $n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$. n est dans le mot d'adresse 2 et le résultat est rangé à l'adresse 3.

```
fact <- 1
si n > 1 alors
  pour i de 2 à n faire
    fact <- fact x i
  fpour
fsi
```

Attention : limite à $5! = 120$.

4.6 Parcours d'un ensemble de données consécutives – l'adressage indirect

Pour parcourir et traiter un ensemble de données consécutives en mémoire (un tableau de données), il faut pouvoir désigner successivement tous les éléments de cet ensemble dans une itération. Pour ce faire, on met l'adresse de la première donnée dans un registre (ex : AL) et on utilise la notation [AL] qui signifie « le contenu du mot dont l'adresse est dans AL ». Il suffit d'incrémenter AL de 1 en 1 dans l'itération pour accéder successivement à toutes les données de l'ensemble. On parle « d'adressage indirect ». C'est un des modes d'adressage classique en langage d'assemblage :

```
- Adressage immédiat    : 10      (constante 10)
- Adressage direct      : [10]    (mot mémoire d'adresse 10)
- Adressage indirect    : [AL]    (le registre AL contient l'adresse voulue)
- Adressage basé        : [AL+4]  (contenu de AL + 4 est l'adresse voulue)
                                  Attention! n'existe pas dans la machine simulée
- Adressage indexé      : [AL+BL] (contenu de AL + contenu de BL)
                                  Attention! n'existe pas dans la machine simulée
```

Attention : dans la machine simulée, l'adressage indirect n'est autorisé que dans les instructions MOV, ce qui est très restrictif par rapports aux processeurs réels !

Assembleur	Langage machine	Explication
MOV DL, [AL]	D3 03 00	DL = [AL] Copie le mot [AL] dans DL
MOV [CL], AL	D4 03 00	[CL] = AL Copie AL dans le mot [CL]

Exercices

7. Écrivez un programme qui fait la somme de 10 mots rangés dans les mots 3 à 12 ; le résultat est rangé à l'adresse 2.

```
somme <- 0
pour i de 3 à 12 faire
  somme <- somme + mot[i]   (le ième mot)
fpour
```

8. Écrivez un programme qui cherche le maximum des mots 2 à 11 de la mémoire et le range à l'adresse 12.

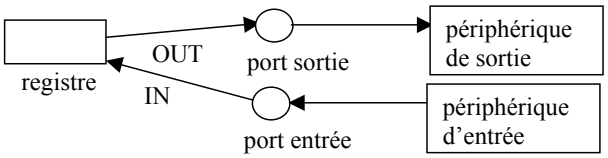
```
max <- -128    (plus petite valeur sur 8 bits)
pour i de 2 à 11 faire
  si mot[i] > max alors
    max <- mot[i]
  fsi
fpour
```

9. Écrivez un programme de tri décroissant des mots 2 à 11 de la mémoire selon l'algorithme suivant (≪ tri par permutations ≫) :

```
pour i de 2 à 10 faire
  pour j de i+1 à 11 faire
    si mot[i] < mot[j] alors
      échanger mot[i] et mot[j]
    fsi
  fpour
fpour
```

4.7 Les entrées/sorties

Les entrées/sorties s'effectuent via des ports. Les ports sont des adresses spéciales associées aux périphériques d'entrée/sortie et repérées par des numéros. Un octet peut être lu ou écrit dans ces ports grâce aux instructions IN et OUT.



Assembleur	Langage machine	Explication
IN 00	F0 00	entrée depuis le port 00 vers AL
OUT 01	F1 01	sortie depuis AL vers le port 01

Dans le simulateur le port de sortie 01 est connecté à deux feux tricolores ; selon la valeur de l'octet transmis ces deux feux s'allument d'une certaine manière :

```
bits   7    6     5      4      3     2     1     0
    rouge orange vert   rouge orange vert  non utilisés
          feu de gauche   feu de droite
```

Le port d'entrée 00 est connecté au clavier. Le registre AL reçoit le code ASCII de la touche du clavier qui a été enfoncée.

Cette forme d'entrée/sortie sans condition correspond aux périphériques qui sont toujours disponibles, comme les interrupteurs, les voyants lumineux ...
Dans le cas de périphériques plus complexes il faut recourir aux entrées/sorties avec condition : en effet, il faut connaître l'état du périphérique avant d'envoyer ou de lire des informations (ex : réseau, disque, ...). Comme nous l'avons vu dans la partie sur le matériel, il existe deux approches principales. Dans la première, le processeur teste répétitivement l'état jusqu'à ce que le périphérique soit prêt. Cette boucle de scrutation occupe inutilement le processeur. La deuxième approche utilise la notion

d'interruption. Une demande d'entrée/sortie est émise. Dès que le périphérique est prêt il envoie une interruption qui est prise en compte immédiatement par le processeur en interrompant le traitement en cours. En effet, pendant l'attente, le processeur peut être affecté par le système d'exploitation à un autre programme.

Exercices

10. Écrivez un programme qui fait alterner indéfiniment les deux feux :

```
     feu gauche : rouge     feu droite : vert
puis feu gauche : rouge     feu droite : orange
puis feu gauche : vert      feu droite : rouge
puis feu gauche : orange    feu droite : rouge
et on recommence...
```

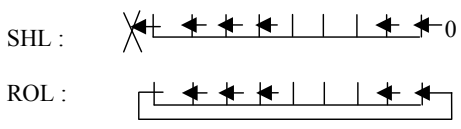
11. Écrivez un programme qui pilote le feu de gauche à partir de la saisie au clavier des lettres r pour rouge, v pour vert, o pour orange. L'autre feu reste éteint. Pour toute autre saisie le feu reste éteint.

4.8 Les manipulations de bits

a) Les instructions de décalage et de rotation

Ces opérations décalent vers la gauche ou vers la droite les bits d'un registre. Elles sont utilisées pour décoder bit à bit des données, ou simplement pour diviser ou multiplier rapidement par une puissance de 2. En effet, décaler AL de n bits vers la gauche revient à le multiplier par 2^n (sous réserve qu'il représente un nombre naturel et qu'il n'y ait pas de dépassement de capacité). De même, un décalage vers la droite revient à diviser par 2^n .

Assembleur	Langage machine	Explication
SHL CL	9C 02	Décalage à gauche : le bit de poids fort est perdu, le bit de poids faible devient 0 (*SHift Left*).
SHR DL	9D 03	Décalage à droite : le bit de poids faible est perdu, le bit de poids fort devient 0 (*SHift Right*).
ROL AL	9A 00	Rotation à gauche : le bit de poids fort devient le bit de poids faible (*ROtate Left*).
ROR BL	9B 01	Rotation à droite : le bit de poids faible devient le bit de poids fort (*ROtate Right*).



b) Les instructions logiques

Les instructions logiques effectuent des opérations logiques bit à bit. On dispose de trois opérateurs logiques : ET, OU et OU exclusif. Il n'y a jamais propagation de retenue lors de ces opérations (chaque bit du résultat est calculé indépendamment des autres).

```
    0 0 1 1         0 0 1 1           0 0 1 1
OU  0 1 0 1    ET   0 1 0 1    OU EX  0 1 0 1
    0 1 1 1         0 0 0 1           0 1 1 0
```

b1) L'instruction OR :

```
    OR   AL, BL             ; AL = AL OU BL
```

OR est souvent utilisé pour forcer certains bits à 1.
Exemple :

```
    MOV BL, F0              BL:  11110000
    MOV AL, 55              AL:  01010101
    OR  AL, BL                   11110101
Les 4 premiers bits de AL sont forcés à 1, les 4 autres sont inchangés.
```

b2) L'instruction AND :

```
    AND   AL, BL            ; AL = AL ET BL
```

AND est souvent utilisé pour forcer certains bits à 0.
Exemple :

```
    MOV BL,  0F             BL:  00001111
    MOV AL, 55              AL:  01010101
    AND AL, BL                   00000101
Les 4 premiers bits de AL sont forcés à 0, les 4 autres sont inchangés.
```

b3) L'instruction XOR :

```
    XOR AL, BL              ; AL = AL OU EXCLUSIF BL
```

XOR est souvent utilisé pour inverser certains bits.
Exemple :

```
    MOV BL, F0              BL:  11110000
    MOV AL, 55              AL:  01010101
    XOR AL, BL                   10100101
Les 4 premiers bits de AL sont inversés, les 4 autres sont inchangés.
```

Remarque : on n'a pas besoin de recourir à l'assembleur pour manipuler des bits ; ces opérateurs existent dans beaucoup de langages évolués, comme Java. En Java SHL s'écrit `>>`, SHR s'écrit `<<`, OR s'écrit | , AND s'écrit & et XOR s'écrit ∧. Exemple :

```
    short i = 13;           // i a la valeur       0000000000001101
    i = i << 2;             // i prend la valeur   0000000000110100
```

Exercices

12. Comment mettre le registre AL à 0 avec un XOR ?

13. Comment tester si AL est pair sans calculer le reste de la division par 2 ?

14. Le mot mémoire 2 contient un entier sur sept bits. Le bit de poids faible est nul. On veut mettre un 1 dans le bit de poids faible si le nombre de bits à 1 parmi les sept premiers est impair et laisser le 0 sinon (bit de parité).

4.9 Les sous-programmes

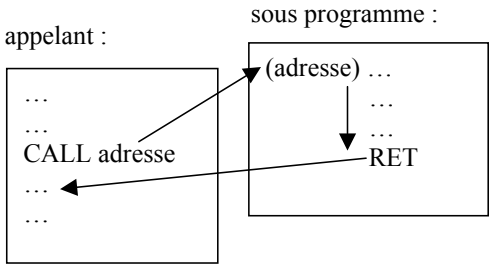
Un sous-programme est une suite d'instructions effectuant un certain traitement et qui sont regroupées par commodité : découpage d'un gros programme en « morceaux », définition d'un « morceau » qui est utilisé plusieurs fois.
Un sous-programme est repéré par l'adresse de sa première instruction. L'exécution du sous programme est déclenchée par un programme appelant. Un sous-programme peut lui même en appeler un autre et ainsi de suite.
L'appel du sous-programme est effectué par l'instruction CALL :

```
CALL adresse
```

L'exécution se poursuit à l'adresse indiquée dans le CALL. La fin du sous-programme est marquée par l'instruction :

```
RET
```

Le processeur revient à l'instruction placée immédiatement après le CALL.



L'adresse de retour, utilisée par RET, est sauvegardée sur la pile par l'instruction CALL. Lorsque le processeur exécute l'instruction RET, il dépile l'adresse de la pile et la range dans le compteur ordinal. Ce mécanisme fonctionne même dans le cas des appels imbriqués (un CALL dans un sous-programme vers un autre sous-programme) et dans le cas des appels récursifs (un sous-programme qui s'appelle lui même un certain nombre de fois).

Assembleur	Langage machine	Explication
CALL 30	CA 30	sauvegarde CO sur la pile et saute à 30.
RET	CB	prend CO sur la pile et saute à cette adresse.

Le plus souvent, le sous-programme effectue un traitement sur des données (paramètres) qui sont fournis par le programme appelant et produit un résultat qui est retourné à ce programme.
Plusieurs stratégies peuvent être employées pour passer les paramètres :

a) passage par les registres

Les valeurs des paramètres et du résultat sont contenues dans des registres bien définis. C'est une méthode simple qui ne convient que si le nombre de paramètres est faible, car il y a peu de registres.
Exemple : le sous-programme à l'adresse hexadécimale 30 calcule le maximum de deux entiers naturels. On convient que les entiers sont passés par les registres AL et BL, et que le résultat est placé dans le registre AL. L'appel s'écrit :

```
MOV AL, ...  ; donnée1
MOV BL, ...  ; donnée2
CALL 30
....         ; résultat disponible dans AL
```

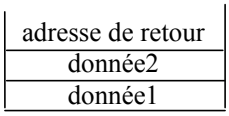
Le sous-programme s'écrit :

```
ORG 30       ; directive qui indique que le code qui suit est situé
             ; à l'adresse 30. Cette directive ne prend pas de place
             ; en mémoire
CMP AL, BL
JNS   saut   ; si AL >= BL  (AL-BL >= 0)
PUSH BL      ; sinon échange de AL et BL
POP AL  ; en utilisant la pile de telle sorte que le max soit dans AL
saut :
RET
```

b) passage par la pile
Les valeurs des paramètres sont empilées. Il peut y en avoir un nombre quelconque. Le sous-programme récupère les valeurs dans la pile.
Exemple : le sous-programme à l'adresse 40 utilise la pile pour passer les paramètres et reçoit le résultat dans la pile. L'appel s'écrit :

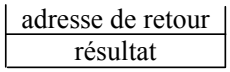
```
PUSH AX      ; donnée1 préalablement rangée dans le registre AX
PUSH BX      ; donnée2 préalablement rangée dans le registre BX
CALL 40
POP  AX  ; récupération du résultat par exemple dans AX
```

Après le CALL la pile contient :



Le sous-programme peut s'écrire :

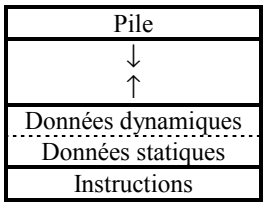
```
ORG 40
POP  CL      ; c'est l'adresse de retour empilée par le CALL
POP  BL      ; deuxième paramètre
POP  AL      ; premier paramètre
CMP BL, AL
JNS   saut   ; si AL >= BL
PUSH BL  ; sinon échange de AL et BL
POP AL       ; en utilisant la pile de telle sorte que le max soit dans AL
saut :
PUSH AL      ; résultat dans la pile
PUSH CL      ; adresse de retour dans la pile
RET          ; dépile l'adresse de retour
```



Le programme apparaît plus complexe. Mais il faut noter que dans les langages d'assemblage réels on peut accéder aux valeurs de la pile sans dépiler, ce qui simplifie le début et la fin du sous-programme. Ce point sera illustré au paragraphe suivant.
À noter que les sous-programmes sauvegardent également dans la pile le contexte du programme appelant (ses registres, ses drapeaux, etc.) avant de commencer leur exécution et restituent ce contexte avant de retourner. De la sorte, le programme appelant n'est pas perturbé par les traitements réalisées dans le sous-programme. Un registre retrouve au retour du sous-programme la valeur qu'il avait avant l'appel même s'il a été utilisé par le sous-programme pour ses calculs propres.

Enfin, la pile sert également à accueillir les variables locales aux sous-programmes car leur durée de vie est la même que celle du sous-programme.

Dans le langage C, la mémoire est gérée avec d'un côté la pile (avec pour chaque appel de sous-programme ses variables locales, ses paramètres, son adresse de retour et le contexte sauvegardé) et de l'autre côté le code et le tas qui contient les données statiques dont la place est réservée à la compilation et les données dynamiques dont la place est allouée à l'exécution. C'est le programmeur qui doit gérer explicitement les données dynamiques (allocation par *malloc* et libération par *free*).



En Java, un programme peut comporter plusieurs *threads* qui s'exécutent en parallèle, avec chacun son compteur ordinal et ses registres. La mémoire est organisée avec une pile par *thread* et un tas commun à tous les *threads*. Dans la pile sont rangés les paramètres des appels de méthodes, les valeurs retournées, leurs contextes et leurs variables locales (valeurs pour les types de base et références pour les objets). Dans le tas sont rangés tous les objets créés dynamiquement par *new*. Le ramasse-miettes (*garbage collector*) libère périodiquement l'espace occupé dans le tas par les objets qui ne sont plus référencés. le programmeur n'a pas à s'en soucier.

4.10 Examen du code généré par un compilateur

Nous examinons pour terminer quelques exemples de traduction de programmes C en assembleur Intel 16 bits. Il s'agit d'une architecture ancienne mais dont la machine simulée utilisée précédemment s'est inspirée. Le compilateur utilisé est le compilateur C/C++ libre Open Watcom qui fonctionne sous Windows (`www.openwatcom.org`) et peut compiler vers toutes les architectures Intel 16 bits et 32 bits.
Le premier exemple illustre le stockage des variables dans la pile et la traduction d'une conditionnelle. Le code C est le suivant :

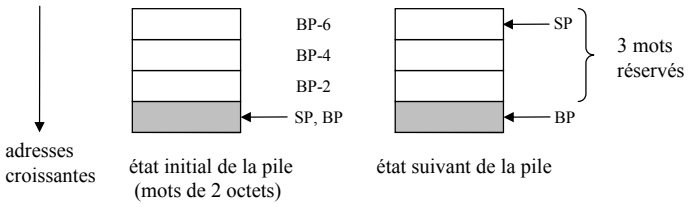
```
int main(void) {
    int x = 11;
    int res;
    char c = 'A';
    if (x < 0)
        res = -1;
    else
        res = 1;
}
```

La traduction assembleur produite par le compilateur pour une architecture à mots de 16 bits est la suivante (le code C est rappelé en début de ligne) :

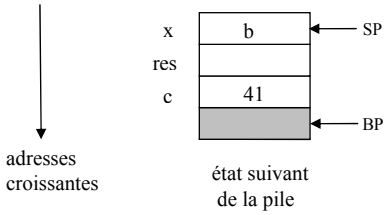
```
                mov     bp,sp
                sub     sp,0x0006                (réserve 3 mots dans la pile)

int x = 11;
            L$1:
                mov     word ptr -0x6[bp],0x000b (met 11 dans x dans la pile)

int res;
char c = 'A';
                mov     byte ptr -0x2[bp],0x41   (met 'A' dans c dans la pile)

if (x < 0)
                cmp     word ptr -0x6[bp],0x0000 (compare x et 0)
                jge     L$2                      (si >= saute en L$2)

  res = -1;
```

```
                mov     word ptr -0x4[bp],0xffff   (met -1 dans res dans la pile)

else
                jmp     L$3                        (saute à la fin)

  res = 1;
            L$2:
                mov     word ptr -0x4[bp],0x0001   (met 1 dans res dans la pile)
            L$3:                                   (fin)
```

La traduction utilise deux registres de un mot (16 bits) : SP (*Stack Pointer*) qui pointe vers le sommet de la pile et BP (*Base Pointer*) qui permet de faire de l'adressage basé, c'est-à-dire qui ajoute ou retire une certaine valeur entière au contenu du registre.
La première instruction copie SP dans BP.
La seconde instruction soustrait 6 en hexa (6 en décimal) à SP. Cela revient à réserver trois mots dans la pile comme le montre la figure suivante, afin de stocker les variables du programme.



L'instruction suivante range `b` en hexadécimal (soit 11 en décimal) dans un mot (*word ptr*) à l'adresse contenu de BP moins 6 en hexadécimal (noté `-0x6[bp]`), qui correspond à l'emplacement de la variable x.
L'instruction suivante range 41 en hexadécimal (soit le code ASCII de A) dans un octet (*byte ptr*) à l'adresse BP-2, qui correspond à l'emplacement de la variable c.
Le mot à l'adresse BP-4 est réservé par le compilateur à la variable res.



La traduction de la conditionnelle reprend le schéma vu en cours. Les étiquettes sont notées L$2 et L$3. La comparaison (CMP) compare x (en BP-6) et 0. Si x est supérieur ou égal à 0 (`jge` ou *jump if greater or equal*) le programme range 1 dans res (en BP-4) et sort de la conditionnelle. Sinon le programme range `ffff` (soit -1 en complément à 2) dans res.

Le deuxième exemple illustre un appel de sous-programme. Le code C est le suivant :

```
int ma_fonction(int x, int y) {
    return x+y;
}

int main(void) {
    int a = 11;
    int b = 22;
    int res;
    res = ma_fonction(a,b);
}
```

La traduction assembleur produite par le compilateur est la suivante :

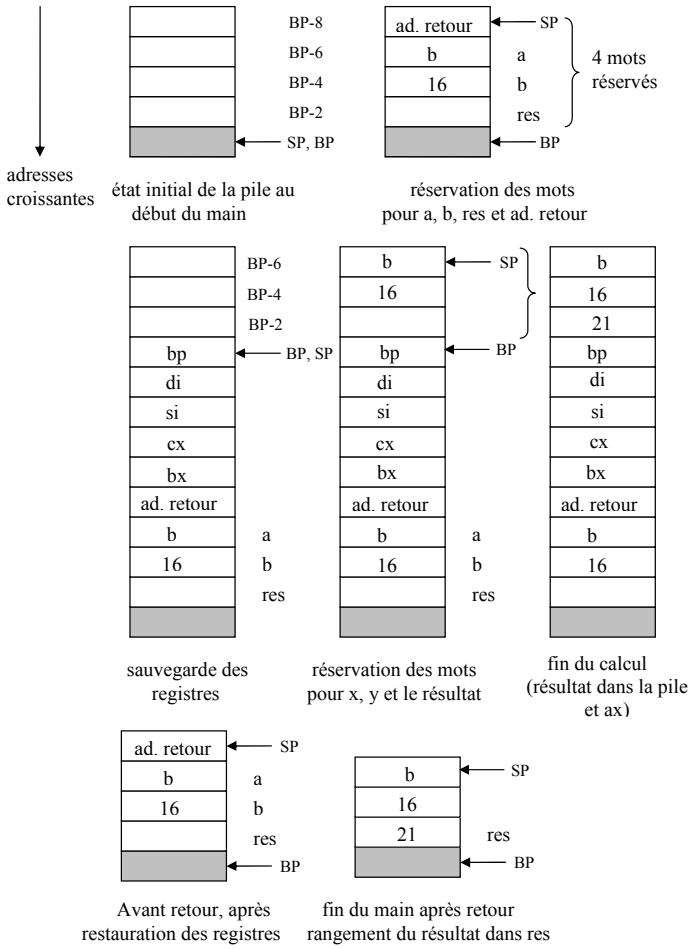
```
main
                mov     bp,sp
                sub     sp,0x0008                (réserve 4 mots dans la pile)
int a = 11;
             L$3:
                mov     word ptr -0x6[bp],0x000b  (met 11 dans a dans la pile)

int b = 22;
                mov     word ptr -0x4[bp],0x0016  (met 22 dans b dans la pile)

int res;
res = ma_fonction(a,b);
                mov     dx,word ptr -0x4[bp]      (met b dans la pile dans dx)
                mov     ax,word ptr -0x6[bp]      (met a dans la pile dans ax)
                call    ma_fonction_              (appel de la fonction)
                mov     word ptr -0x2[bp],ax      (résultat dans ax rangé dans
                                                  res dans la pile)
-------------------------------------------------------------------------------

int ma_fonction(int x, int y) {
             ma_fonction_:
                push    bx
                push    cx
                push    si                        (sauvegarde tous les registres
                push    di                        sauf ax et dx qui contiennent
                push    bp                        a et b)
                mov     bp,sp
                sub     sp,0x0006                 (réserve 3 mots dans la pile)
             L$1:
                mov     word ptr -0x6[bp],ax      (met ax, cad a, dans la pile)
                mov     word ptr -0x4[bp],dx      (met bx, cad b, dans la pile)

  return x+y;
                mov     ax,word ptr -0x6[bp]      (met a dans ax)
                add     ax,word ptr -0x4[bp]      (met a+b dans ax)
                mov     word ptr -0x2[bp],ax      (met a+b dans la pile)

}
                mov     ax,word ptr -0x2[bp]      (met a+b dans ax)
             L$2:
                mov     sp,bp
                pop     bp
                pop     di
                pop     si
                pop     cx
                pop     bx                        (restaure tous les registres)
                ret
```

Le compilateur a choisi de passer les deux arguments dans les registres `ax` et `dx` et de récupérer le résultat dans `ax`. Les cinq autres registres (`cx`, `dx`, `si`, `di`, `bp`) sont sauvegardés (`push`) sur la pile en début de sous-programme.

En début du programme principal (`main`), quatre mots sont réservés sur la pile pour `a`, `b`, `res` et l'adresse de retour. Les arguments sont rangés dans `ax` et `dx` avant l'appel du sous-programme (`call`) et le résultat est rangé de `ax` dans `res` au retour.
En début de sous-programme, après sauvegarde des registres, trois mots sont réservés sur la pile pour les deux arguments et le résultat. Puis la somme est effectuée : le premier argument est mis dans `ax` (alors qu'il y est déjà !) ; le second argument est ajouté à `ax` ; la somme est remise dans la pile.
En fin de sous-programme le résultat est retourné dans `ax` (il y est placé alors qu'il y est déjà !) et les registres sont restaurés depuis la pile (`pop`) afin d'éviter les effets de bord indésirables (le sous-programme doit laisser les registres dans l'état où il les a trouvés).

Le code est généré par le compilateur en appliquant des schémas de traduction prédéfinis. Il n'est donc pas optimal. On trouve, comme nous l'avons noté, plusieurs transferts inutiles. Les compilateurs peuvent supprimer ces instructions inutiles en réalisant une analyse du code généré dans une seconde passe.

Les évolutions de la pile sont résumées dans la figure suivante :



Exercice

15. Écrivez un sous-programme situé à l'adresse 20 qui lit un texte au clavier terminé par la touche entrée (code hexadécimal 0D) et range ce texte à l'adresse mémoire reçue dans la pile.
Écrivez le programme appelant qui appelle ce sous-programme avec l'adresse C0 en paramètre dans la pile (cette adresse entraîne l'affichage du texte dans une fenêtre par le simulateur).

Exercices récapitulatifs (tirés des DS des années précédentes)

16. *Opérations sur des ensembles de données*
a) Écrivez un programme qui enlève 1 à tous les nombres contenus dans les mots d'adresses 2 à 11.
b) Écrivez un programme qui enlève 1 seulement aux mots mémoire qui contiennent un nombre impair. Utilisez l'instruction MOD qui donne le reste de la division entière pour tester la parité. Exemple :

```
   avant                 après
mot 2: 12             mot 2: 12
mot 3: 13             mot 3: 12
mot 4: 19             mot 4: 18
mot 5:  4             mot 5:  4
```

17. *Calcul itératif d'une puissance*
La répétition "tant que" se traduit de la manière suivante :

```
                                re:
tant que  op1 > op2 répéter       CMP op2, op1
  instructions                    JNS fin          ; op2-op1 >= 0
fin tant que                      instructions
                                  JMP re
                                fin:
                                  ...
```

Écrivez le programme qui calcule 2^5 suivant l'algorithme :

```
e <- 1
n <- 5
x <- 2
tant que n > 0 répéter
  e <- e * x
  n <- n - 1
fin tant que
```

18. *La suite de Syracuse*
Elle est définie par :

u_1 = a, où a est un entier quelconque
...
$u_{n+1} = u_n/2$, si u_n est pair
$u_{n+1} = (3\ u_n{+}1)/2$, si u_n est impair

Cette suite a la particularité de toujours atteindre 1 au bout d'un certain temps (mais cela n'a jamais pu être prouvé).
Exemple :

$u_1 = 7$
$u_2 = (3 \cdot 7 + 1)\ /\ 2 = 11$
$u_3 = (3 \cdot 11 + 1)\ /\ 2 = 17$
$u_4 = (3 \cdot 17 + 1)\ /\ 2 = 26$
$u_5 = 26\ /\ 2 = 13$
$u_6 = (3 \cdot 13 + 1)\ /\ 2 = 20$
$u_7 = 20\ /\ 2 = 10$
$u_8 = 10\ /\ 2 = 5$
$u_9 = (3 \cdot 5 + 1)\ /\ 2 = 8$
$u_{10} = 8\ /\ 2 = 4$

$u_{11} = 4 \ / \ 2 = 2$
$u_{12} = 2 \ / \ 2 = 1$

Écrivez un programme qui compte le nombre d'éléments à calculer pour atteindre 1 (12 dans l'exemple). La donnée initiale (a) est dans le mot d'adresse 2 et le résultat est rangé dans le mot d'adresse 3.

L'algorithme est le suivant :

```
nb <- 1
u <- a
tant que u != 1 répéter
  si u est pair alors
    u <- u / 2
  sinon
    u <- (3 * u + 1) / 2
  finsi
  nb <- nb + 1
fin tant que
```

19. *Passage en octal*
On a 24 chiffres binaires dans les mots d'adresses 2, 3 et 4. Écrivez le programme qui calcule les chiffres octaux correspondant aux 6 premiers chiffres binaires de ces 3 mots dans les mots 5 à 10.
Exemple :

```
Mots            2         3         4          5    6    7    8    9    10
Valeurs (hexa)  34        17        84         1    5    0    5    4    1
Binaire         00110100  00010111  10000100
                          données             résultats (tranches de 3 bits)
```

L'algorithme est le suivant :

```
k = 5
pour i de 2 à 4 répéter
  décaler mot[i] de 5 bits à droite
       (on pourrait écrire un sous-programme de décalage à droite de n bits)
  ranger résultat dans mot[k]
  k++
  forcer les 3ers bits de mot[i] à 0
  décaler le résultat de 2 bits à droite
  ranger le résultat dans mot[k]
  k++
fin pour
```

20. *Recherches*
a) Écrivez un programme de recherche d'une valeur dans un tableau de 10 entiers par la méthode naïve correspondant à l'algorithme ci-dessous.
La variable trouvé est rangée dans le mot 2. Elle vaut 1 si le mot est trouvé et 0 sinon. La valeur cherchée est rangée dans le mot 3. Le tableau de 10 entiers est rangé dans les mots 4 à 13 (en décimal).

```
trouvé <- 0
pour i de 4 à 13 faire
  si (mot[i] = val-cherchée) alors
    trouvé <- 1 ; terminer  (arrêt ou sortie de la boucle)
  fsi
finpour
```

b) Reprenez le même problème en utilisant l'algorithme de recherche dichotomique :

```
trouvé <- 0
bas <- 4
haut <- 13
tantque bas <= haut faire
  milieu <- (haut+bas)/2
  si (mot[milieu] = val-cherchée) alors
    trouvé <- 1
    terminer
  sinon
    si (mot[milieu] < val-cherchée) alors
      bas <- milieu + 1
    sinon haut <- milieu - 1
    fsi
  fsi
fintantque
```

21. *Entrées et sorties*
Ecrivez un programme qui lit des caractères au clavier jusqu'à la frappe de la touche entrée. Les lettres minuscules sont transformées en majuscules. Les autres caractères ne sont pas modifiés. Tous les caractères modifiés ou non sont rangés dans la zone de mémoire d'adresses C0 à FF (cette zone est automatiquement affichée dans une fenêtre par la machine simulée).
Rappel :

- l'instruction IN 00 attend l'appui d'une touche et range dans le registre AL le code ASCII du caractère saisi,
- la table des codes ASCII est donnée en annexe ; la touche entrée correspond au code ASCII décimal 13.

22. *D'autres machines*
On considère les trois machines suivantes, dans lesquelles les mots mémoire sont repérés par des identificateurs (ex : A, B, X, ...).

1. M0 est une machine à pile : elle peut ranger un mot mémoire au sommet de la pile (PUSH X), ranger le sommet de la pile en mémoire et le retirer de la pile (POP X). L'opération ADD enlève 2 mots de la pile, fait l'addition et met le résultat au sommet de la pile.
2. M1 est une machine avec un accumulateur : elle peut transférer un mot mémoire dans l'accumulateur (LOAD X) et sauvegarder l'accumulateur dans un mot mémoire (STORE X). Elle peut additionner un mot mémoire avec le contenu de l'accumulateur et ranger le résultat dans l'accumulateur (ADD X).
3. M2 est une machine avec 8 registres (Ri pour i de 1 à 8) : elle peut transférer un mot mémoire vers un registre par LOAD Ri, X et transférer un registre vers un mot mémoire par STORE X, Ri. Les opérations arithmétiques se font uniquement avec des registres : ADD Ri, Rj, Rk où Ri est la destination.

a) Soient 3 mots mémoire A, B, C. Écrivez pour les trois machines, le code qui réalise A = B + C.
b) De même, écrivez pour les trois machines, le code correspondant à la suite d'instructions :

```
A = B + C
B = A + C
```

Comparez ces codes en nombre d'instructions et en nombre d'accès mémoire, c'est-à-dire de lectures ou écritures en mémoire. Essayez de minimiser ce nombre d'accès mémoire en écrivant les codes. Les accès aux registres et à la pile ne sont pas des accès mémoire.

5 Annexe A : table des puissances de 2

n	2^n	2^{-n}
0	1	1
1	2	0,5
2	4	0,25
3	8	0,125
4	16	0,0625
5	32	0,03125
6	64	0,015625
7	128	0,0078125
8	256	0,00390625
9	512	0,001953125
10	1024	0,0009765625
11	2048	0,00048828125
12	4096	0,000244140625
13	8192	0,0001220703125
14	16384	0,00006103515625
15	32768	0,000030517578125
16	65536	0,0000152587890625
17	131072	0,00000762939453125
18	262144	0,000003814697265625
19	524288	0,0000019073486328125
20	1048576	0,00000095367431640625

6 Annexe B1 : code ASCII

Base 10	Base 16	Caractère	Signification
0	00	NUL	*Null* (caractère nul)
1	01	SOH	*Start of Header* (début d'entête)
2	02	STX	*Start of Text* (début de texte)
3	03	ETX	*End of Text* (fin du texte)
4	04	EOT	*End of Transmission* (fin de transmission)
5	05	ENQ	*Enquiry* (demande)
6	06	ACK	*Acknowledge* (accusé de réception)
7	07	BEL	*Bell* (caractère d'appel)
8	08	BS	*Backspace* (espacement arrière)
9	09	HT	*Horizontal Tab* (tabulation horizontale)
10	0A	LF	*Line Feed* (saut de ligne)
11	0B	VT	*Vertical Tab* (tabulation verticale)
12	0C	FF	*Form Feed* (saut de page)
13	0D	CR	*Carriage Return* (retour chariot)
14	0E	SO	*Shift Out* (fin d'extension)
15	0F	SI	*Shift In* (démarrage d'extension)
16	10	DLE	*Data Link Escape* échappement de liaison de données)
17	11	DC1	*Device Control 1* (contrôle de périphérique 1)
18	12	DC2	*Device Control 2*
19	13	DC3	*Device Control 3*
20	14	DC4	*Device Control 4*
21	15	NAK	*Negative Acknowledge* (accusé de réception négatif)
22	16	SYN	*Synchronous Idle* (inactivité)
23	17	ETB	*End of Transmission Block* (fin du bloc de transmission)
24	18	CAN	*Cancel* (annulation)
25	19	EM	*End of Medium* (fin de support)
26	1A	SUB	*Substitute* (substitution)
27	1B	ESC	*Escape* (échappement)
28	1C	FS	*File Separator* (séparateur de fichier)
29	1D	GS	*Group Separator* (séparateur de groupe)
30	1E	RS	*Record Separator* (séparateur d'enregistrement)
31	1F	US	*Unit Separator* (séparateur d'unité)
32	20	SP	*Space* (espace)
33	21	!	Point d'exclamation
34	22	”	Guillemet droit
35	23	#	Croisillon ou Dièse
36	24	$	Dollar (symbole)
37	25	%	Pourcent
38	26	&	Esperluette ou « et commercial »
39	27	’	Apostrophe droite
40	28	(	Parenthèse ouvrante
41	29	)	Parenthèse fermante
42	2A	*	Astérisque
43	2B	+	Plus
44	2C	,	Virgule
45	2D	-	Moins
46	2E	.	Point
47	2F	/	Barre oblique (*slash*)
48	30	0	Chiffre 0

Base 10	Base 16	Caractère	Signification
49	31	1	
50	32	2	
51	33	3	
52	34	4	
53	35	5	
54	36	6	
55	37	7	
56	38	8	
57	39	9	
58	3A	:	Deux-points
59	3B	;	Point-virgule
60	3C	<	Inférieur
61	3D	=	Égal
62	3E	>	Supérieur
63	3F	?	Point d'interrogation
64	40	@	A commercial (arrobase)
65	41	A	Lettre A majuscule
66	42	B	
67	43	C	
68	44	D	
69	45	E	
70	46	F	
71	47	G	
72	48	H	
73	49	I	
74	4A	J	
75	4B	K	
76	4C	L	
77	4D	M	
78	4E	N	
79	4F	O	
80	50	P	
81	51	Q	
82	52	R	
83	53	S	
84	54	T	
85	55	U	
86	56	V	
87	57	W	
88	58	X	
89	59	Y	
90	5A	Z	
91	5B	[	Crochet ouvrant
92	5C	\	Barre oblique inversée (*backslash*)
93	5D	]	Crochet fermant
94	5E	^	Accent circonflexe
95	5F	_	Tiret bas ou souligné (*Underscore*)
96	60	‘	Apostrophe gauche
97	61	a	Lettre a minuscule
98	62	b	
99	63	c	
100	64	d	

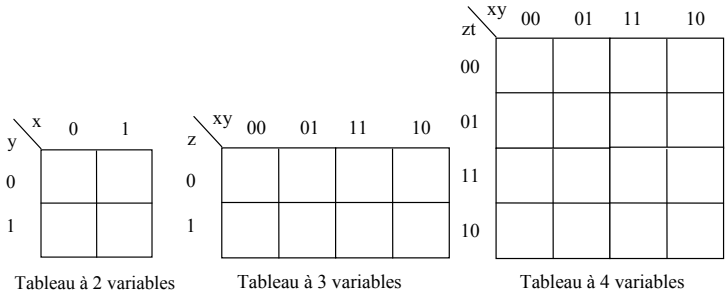
Base 10	Base 16	Caractère	Signification
101	65	e	
102	66	f	
103	67	g	
104	68	h	
105	69	i	
106	6A	j	
107	6B	k	
108	6C	l	
109	6D	m	
110	6E	n	
111	6F	o	
112	70	p	
113	71	q	
114	72	r	
115	73	s	
116	74	t	
117	75	u	
118	76	v	
119	77	w	
120	78	x	
121	79	y	
122	7A	z	
123	7B	{	Accolade ouvrante
124	7C	|	Barre verticale
125	7D	}	Accolade fermante
126	7E	~	Tilde
127	7F	DEL	*Delete* (effacement)

7 Annexe B2 : code ISO-8859-1

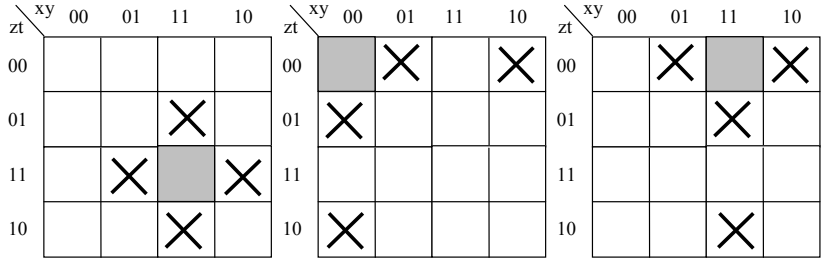
ISO-8859-1 (codes héxa)																
->	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	*NUL*	*SOH*	*STX*	*ETX*	*EOT*	*ENQ*	*ACK*	*BEL*	*BS*	*HT*	*LF*	*VT*	*FF*	*CR*	*SO*	*SI*
1	*DLE*	*DC1*	*DC2*	*DC3*	*DC4*	*NAK*	*SYN*	*ETB*	*CAN*	*EM*	*SUB*	*ESC*	*FS*	*GS*	*RS*	*US*
2	*SP*	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	*DEL*
8	*PAD*	*HOP*	*BPH*	*NBH*	*IND*	*NEL*	*SSA*	*ESA*	*HTS*	*HTJ*	*VTS*	*PLD*	*PLU*	*RI*	*SS2*	*SS3*
9	*DCS*	*PU1*	*PU2*	*STS*	*CCH*	*MW*	*SPA*	*EPA*	*SOS*	*SGCI*	*SCI*	*CSI*	*ST*	*OSC*	*PM*	*APC*
A	*NBSP*	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
B	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

8 Annexe C : les tableaux de Karnaugh

La méthode de simplification de Karnaugh repose sur l'identité :
$(A \cdot B) + (A \cdot \bar{B}) = A \cdot (B + \bar{B}) = A$
Elle est basée sur l'inspection visuelle de tableaux disposés de façon telle que les cases adjacentes en ligne et en colonne ne diffèrent que par l'état d'une variable et une seule, ce qui permet d'exploiter l'identité ci-dessus pour simplifier la fonction.
Si une fonction dépend de n variables il y a 2^n valeurs possibles dans la table de vérité. Chacune de ces valeurs est représentée par une case dans un tableau. Les figures suivantes donnent la structure des tableaux de Karnaugh pour 2, 3 et 4 variables (on peut généraliser la méthode à 5 et 6 variables). Les lignes et les colonnes sont définies de telle sorte que d'une case à sa voisine une seule variable change de valeur.



Il faut comprendre chaque ligne et chaque colonne comme une structure cyclique continue : chaque case a toujours quatre voisins qu'il faut éventuellement chercher à l'autre extrémité de la ligne ou de la colonne. De manière imagée, on peut dire que l'on enroule la table en rapprochant le bas et le haut du tableau ou bien son bord droit et son bord gauche.
Les figures suivantes illustrent cette idée, les croix y matérialisent les voisins des cases colorées qui correspondent respectivement aux termes $x \cdot y \cdot z \cdot t$, $\bar{x} \cdot \bar{y} \cdot \bar{z} \cdot \bar{t}$, $x \cdot y \cdot \bar{z} \cdot \bar{t}$:



Le passage de la table de vérité au tableau de Karnaugh consiste à remplir chaque case avec la valeur correspondante de la fonction.

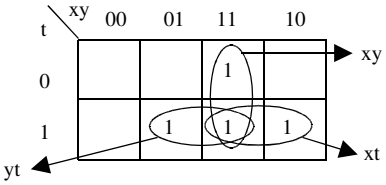
La méthode de simplification de Karnaugh consiste à rassembler les cases adjacentes contenant des 1 par groupes de 2, 4 ou 8 termes. Dans un groupement de deux termes on élimine la variable qui change de valeur et on conserve le produit des variables qui ne changent pas. Dans un groupement de quatre on élimine les deux variables qui changent de valeurs. Dans un groupement de huit on élimine trois variables, etc.

On cherche à avoir le minimum de groupements, chaque groupement rassemblant le maximum de termes. Une même case peut intervenir dans plusieurs groupements car $A + \bar{A} = A$. Pour les cases isolées on ne peut éliminer aucune variable. L'expression logique finale est la réunion des groupements.

Exemple 1 : soit la fonction F définie par la table de vérité suivante.

x	y	t	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

On obtient le tableau :

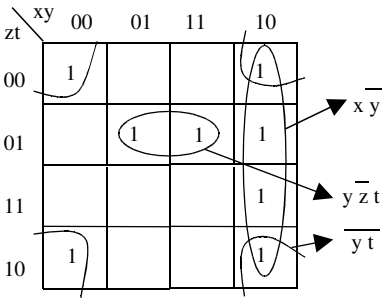


La fonction simplifiée s'écrit : $F = x \cdot y + x \cdot t + y \cdot t$.

Exemple 2 : soit la fonction F définie par la table de vérité suivante.

x	y	z	t	F
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

On obtient le tableau :



La fonction simplifiée s'écrit : $F = x \cdot \bar{y} + \bar{y} \cdot \bar{t} + y \cdot \bar{z} \cdot t$.

9 Annexe D : le langage d'assemblage de la machine simulée

9.1 Instructions de transfert

Les *flags* ne sont pas modifiés.

Assembleur	Langage machine	Explication
MOV AL, 1B	D0 00 1B	AL = 1B range 1B (hexa) dans AL
MOV BL, [C2]	D1 01 C2	BL = [C2] copie le mot d'adresse C2 dans BL
MOV [15], CL	D2 15 02	[15] = CL copie CL dans le mot d'adresse 15
MOV DL, [AL]	D3 03 00	DL = [AL] copie le mot [AL] dans DL
MOV [CL], AL	D4 03 00	[CL] = AL copie AL dans le mot [CL]

9.2 Instructions de calcul arithmétique et logique sur des registres

Les *flags* sont modifiés.

Assembleur	Langage machine	Explication
AL, BL	A0 00 01	AL = AL + BL
SUB BL, CL	A1 01 02	BL = BL − CL
MUL CL, DL	A2 02 03	CL = CL × DL
DIV DL, AL	A3 03 00	DL = DL / AL (division entière)
MOD AL, BL	A6 00 01	AL = AL mod BL (reste de la division entière)
INC DL	A4 03	DL = $DL + 1$
DEC AL	A5 00	AL = $AL - 1$
AND AL, BL	AA 00 01	AL = AL et BL
OR CL, BL	AB 02 01	CL = CL ou BL
XOR AL,BL	AC 00 01	AL = AL ou exclusif BL
NOT BL	AD 01	BL = non BL
ROL AL	9A 00	Rotation à gauche : le bit de poids fort devient le bit de poids faible
ROR BL	9B 01	Rotation à droite : le bit de poids faible devient le bit de poids fort.
SHL CL	9C 02	Décalage à gauche : le bit de poids fort est perdu, le bit de poids faible devient 0.
SHR DL	9D 03	Décalage à droite : le bit de poids faible est perdu, le bit de poids fort devient 0.

9.3 Calculs avec une constante hexadécimale (adressage immédiat)

Les *flags* sont modifiés.

Assembleur	Langage machine	Explication
ADD AL, 12	B0 00 12	AL = $AL + 12$
SUB BL, 15	B1 01 15	BL = $BL - 15$
MUL CL, 03	B2 02 03	CL = $CL \times 03$
DIV DL, 02	B3 03 02	DL = $DL/02$ (division entière)
MOD AL,10	B6 00 10	AL = AL mod 10 (reste de la division entière)
AND AL, 0F	BA 00 0F	AL = AL et 0F
OR CL, F0	BB 02 F0	CL = CL ou F0
XOR AL, AA	BC 00 AA	AL = AL ou exclusif AA

9.4 Instructions de comparaison

Les *flags* sont modifiés.

Assembleur	Langage machine	Explication
CMP AL, BL	DA 00 01	met le *flag* Z à 1 si AL = BL
		met le *flag* S à 1 si AL < BL
CMP BL,13	DB 01 13	met le *flag* Z à 1 si BL = 13
		met le *flag* S à 1 si BL < 13
CMP CL, [20]	DC 02 20	met le *flag* Z à 1 si CL = [20]
		met le *flag* S à 1 si CL < [20]

9.5 Instructions de branchement

Les *flags* ne sont pas modifiés.
Le compteur ordinal (IP) est modifié. Les plus grands sauts possibles sont de +127 mots et −128 mots.

Assembleur	Langage machine	Explication
JMP HERE	C0 12	augmente IP de 12 (l'étiquette HERE est 12 mots après l'instruction courante)
JZ A	C1 09	augmente IP de 9 si le *flag* Z est à 1 (l'étiquette A est à 9 mots après l'instruction courante)
JNZ PLACE	C2 04	augmente IP de 4 si le *flag* Z est à 0
JS R	C3 E1	diminue IP de 31 si le *flag* S est à 1 (l'étiquette R est à 31 mots avant l'instruction courante)
JNS START	C4 04	augmente IP de 4 si le *flag* S est à 0

9.6 Appel de sous-programmes

Les *flags* ne sont pas modifiés.

Assembleur	Langage machine	Explication
CALL 30	CA 30	sauvegarde IP au sommet de la pile et saute à 30.
RET	CB	prend IP au sommet de la pile et saute à cette adresse

9.7 Manipulation de la pile

Les *flags* ne sont pas modifiés.

Assembleur	Langage machine	Explication
PUSH BL	E0 01	BL est sauvegardé au sommet de la pile
POP CL	E1 02	CL est restauré depuis le sommet de la pile

9.8 Instructions d'entrée/sortie

Les *flags* ne sont pas modifiés.

Assembleur	Langage machine	Explication
IN 00	F0 00	entrée depuis le port 00 vers le registre AL
OUT 01	F1 01	sortie depuis le registre AL vers le port 01

9.9 Instructions diverses

Les *flags* ne sont pas modifiés.

Assembleur	Langage machine	Explication
END	00	fin du programme (une seule sur la dernière ligne)
HALT	00	arrêt de l'exécution (éventuellement plusieurs)
DB constante	constante hexa	définit une donnée (*DefineByte*)
DB "HELLO"	suite de codes	les codes ASCII de HELLO en mémoire
NOP	FF	instruction vide

10 Annexe E : glossaire des principaux termes

Ce glossaire contient une sélection des principaux termes utilisés dans ce cours et dont la connaissance fait partie du jargon de base des informaticiens professionnels.

accès direct à la mémoire mécanisme qui permet un accès direct à la mémoire vive sans passer par le processeur ce qui accélère les performances des entrées/sorties (cf. DMA)

accumulateur registre de calcul de l'unité arithmétique et logique

adressage direct dans une instruction, référence à un mot mémoire par son adresse

adressage immédiat dans une instruction, référence à une valeur

adressage indirect dans une instruction, référence à un mot mémoire dont l'adresse est dans un registre

ASCII codification des caractères sur 7 ou 8 bits

assembleur programme de traduction du langage d'assemblage en langage machine

bascule circuit réalisant une mémoire de un bit

benchmark programme et données permettant de tester les performances des ordinateurs ou des processeurs

bit chiffre binaire (abbréviation de *binary digit*)

bit de poids faible bit de puissance 0 (unité)

bit de poids fort bit correspondant à la plus grande puissance de 2

byte cf. octet

branchement conditionnel instruction de saut conditionné par les drapeaux du registre d'état

branchement inconditionnel instruction de saut sans condition

bus support de transfert d'information entre les composants d'un ordinateur

bus d'extension bus où sont connectés les contrôleurs de périphériques

bus local bus interne entre processeur et mémoire cache

bus système bus entre processeur et mémoire centrale

circuit combinatoire circuit de calcul où les sorties dépendent uniquement des données

circuit intégré circuit localisé sur une puce électronique

circuit séquentiel circuit de mémoire où les sorties dépendent des entrées et des sorties précédentes

compilateur programme de traduction d'un langage évolué dans un langage de plus bas niveau (langage d'assemblage ou code intermédiaire)

complément à 2 mode de représentation des entiers relatifs

compteur ordinal registre de l'unité de commande qui contient l'adresse de la prochaine instruction à exécuter

CPU cf. unité centrale (*central processing unit*)

DCB cf. décimal codé binaire

DMA cf. accès direct à la mémoire (*Direct Memory Access*)

décimal codé binaire représentation des entiers par codification binaire de chaque chiffre décimal

drapeau indicateur d'état (bit du registre d'état) cf. *flag*

représentation en excédent reprsésentation des entiers relatifs par décalage

exposant puissance de 2 dans un réel flottant

flag cf. drapeau

formattage préparation d'un disque en vue de recevoir des données

garbage collector cf. ramasse-miettes

giga 2 puissance 30 (milliard)

hexadécimal représentation en base 16

horloge permet à l'unité de commande de générer les micro commandes selon un échéancier précis

IEEE 754 norme de codification des réels

instruction de transfert instruction de déplacement de données entre registres et/ou mots mémoire

instruction de branchement instruction de saut ou rupture de séquence

interprète programme de traduction et d'exécution ligne par ligne d'un programme

interruption événement généré par le matériel ou par programme qui interrompt le processeur et permet de transférer le contrôle d'un programme à un autre

jeu d'instructions ensemble des instructions d'un processeur

kilo 2 puissance 10 (1024 ou un millier)

langage d'assemblage langage machine sous forme symbolique (avec mnémoniques, identificateurs, étiquettes, etc.)

langage évolué langage de plus haut niveau que le langage d'assemblage et qui doit être traduit vers lui (ex : java, C, COBOL, ...)

langage machine langage directement exécutable par le processeur (forme binaire)

mantisse partie décimale et fractionnaire d'un réel flottant

méga 2 puissance 20 (un million)

mégaflop million d'instructions flottantes par seconde

mémoire auxiliaire mémoire externe persistante

mémoire cache mémoire plus rapide que la mémoire centrale où sont stockées les informations les plus utilisées pour accélérer le traitement

mémoire centrale mémoire vive (non persistante) de l'ordinateur où sont stockés programmes, données et résultats

mémoire morte mémoire en lecture seulement

mémoire vive cf. mémoire centrale

micro commande ordre envoyé par le séquenceur après décodage d'une instruction pour la faire réaliser par les composants de l'ordinateur

MIPS million d'instructions par seconde

mot groupement d'octets qui peut être traité d'un bloc

octal représentation en base 8

octet ensemble de huit bits (*byte*)

péta 2 puissance 50 (un million de milliards)

pile zone mémoire gérée en dernier entré premier sorti ou LIFO (*last in first out*)

pipeline mode de traitement avec parallélisme entre les étapes de réalisation des instructions (étages)

pixel position d'affichage sur un écran en mode graphique

port adresse d'entrée/sortie

porte logique circuit logique élémentaire (ET, OU, NON, OU EXCLUSIF)

processeur CISC processeur avec un jeu d'instruction étendu (*Complex Instruction Set Computer*)

processeur RISC processeur avec un jeu d'instruction réduit (*Reduced Instruction Set Computer*)

RAM cf. mémoire vive (*Random Access Memory*)

ramasse-miettes programme qui récupère la mémoire des objets qui ne sont plus utilisés (cf. *garbage collector*)

registre zone mémoire d'un mot à accès très rapide

registre d'état registre contenant les drapeaux (*flags*) positionés par l'unité arithmétique et logique après traitement d'une instruction

registre instruction registre de l'unité de commande qui accueille l'instruction à décoder

ROM cf. mémoire morte (*Read Only Memory*)

séquenceur de commandes composant de l'unité de contrôle cadencé par l'horloge qui génère les micro commandes

superscalaire ordinateur avec plusieurs unités de traitement spécialisées fonctionnant en parallèle (opérations entières, opérations flottantes ...)

tas zone de stockage en mémoire

tera 2 puissance 40 (mille milliards)

UAL cf. unité arithmétique et logique

Unicode codification des caractères sur seize bits

unité arithmétique et logique unité réalisant les calculs arithmétiques et logiques (UAL)

unité centrale processeur + mémoire (cf. CPU)

unité de commande unité qui décode et exécute les instructions

unité d'échange unité qui interface les périphériques