Architecture des ordinateurs

Notes de cours (un peu incomplètes) sur archi + vidéos d'explications des badges

?? Tuto badge d'archi

- **Badge AA1 - Histoire de l'informatique** : pas de vidéo mais relire le chapitre sur l'histoire dans cette synthèse et dans le syllabus devrait être suffisant
- **Badge AA2 (partie 1) - Présentation générale d'un ordinateur** : pas de vidéo mais c'est juste du savoir donc pareil que pour le badge précédent
- **Badge AA2 (partie 2) - Les mémoires**

https://www.youtube-nocookie.com/embed/K9h6-NiQrMU

- **Badge AA2 (partie 3) - Les architectures à processeurs multiples** : pas de vidéo
- **Badge AA3 (partie 1) - Systèmes de numération et conversions entre bases** : pas de vidéo
- **Badge AA3 (partie 2) - Représentation interne des informations**

https://www.youtube-nocookie.com/embed/sZerEOydlNU

- **Badge AA3 (partie 3) - Détection et correction d'erreurs**

https://www.youtube-nocookie.com/embed/ecw8AbBKZCM

- **Badge AA4 - Les circuits logiques**

https://www.youtube-nocookie.com/embed/7mLNb8XIRHw

- **Badge AA5 - L'unité centrale de traitement (CPU)**

https://www.youtube-nocookie.com/embed/wNdr9hWLT7Q

Histoire de l'informatique

Histoire de l'informatique

Dès le début de l'humanité, nous avons commencé à compter. D'abord on a compté en **base 1** (base unitaire). Par exemple, 1 mouton = 1 caillou (d'où *calculus* qui donne le mot *calcul* en français)

> Un *nombre* est une quantité que l'on veut représenter, tandis qu'un *chiffre* est un symbole qui va être utiliser pour représenter le nombre.

Ensuite on a commencé à utiliser la *numérotation de position*. C'est toujours ce que l'on utilise aujourd'hui, c'est à dire que la position d'un chiffre dans l'écriture d'un nombre a de l'importance.

Par exemple, en **base 10** (ce que l'on utilise au quotidien pour compter), si je prends le nombre "542", on peut le décomposer comme ceci :

$$ 542 = 5*10^2 + 4*10^1 + 2*10^0 $$

C'est ainsi que le *zéro* fait son apparition, avant représenter "rien" n'avait pas d'utilité, mais en numérotation de position, ça a une utilité très claire d'éviter la confusion.

$$ 1 \ 1 \neq 11 \neq 101 \neq 1001 \neq 1010 $$

Voici quelques exemples de bases (celle indiquées en gras sont très liées à l'informatique)

Base	Nom	Usage	Origine
1	Unitaire	Comptage (doigts, cailloux, entailles, etc)	
2	**Binaire**	**Logique, électronique, informatique**	
5	Quinaire		Aztèques (doigts d'une main)
7	Septénaire	Jours de la semaine, notes (tons)	
8	**Octal**	**Informatique**	**Premiers ordinateurs**
10	Décimal	Système le plus répandu	Chinois (doigts des 2 mains)
12	Duodécimal	Mois, heures, musique (ton et demi-tons)	Egyptiens

Base	Nom	Usage	Origine
16	**Hexadécimal**	**Informatique**	
20	Vicésimal		Mayas (doigts + orteils)
60	sexagésimal	Trigonométrie (angles), minutes, secondes	Babyloniens, indiens, arabes, ...

Deux millénaires de progrès

Vers 500av JC, nous avons commencé à compter avec des bouliers, des tables à compter et des bâtons de Neper

- Fonctionnement d'un bâton de Napier

1614: John Neper invente la théorie des logarithmes. Grace aux logarithme il est possible de remplacer la multiplication ou la division de 2 nombres par l'addition ou la soustraction de leur logarithmes : $ \log{a*b} = \log{a} + \log{b} $ et $ \log{\frac{a}{b}} = \log{a} - \log{b} $

1623: Wilhelm Shickard (mathématicien) consoit une machine à calculer qui reprends l'idée des batons de Neper

1642: Blaise Pascal (philosophe et scientifique) invente une machine à calculer qui permet principalementd'additionner et de soustraire 2 nombres de six chiffres. En répetant l'opération on pouvait ainsi multiplier.

- Additioner avec la réplique de la Pascaline 1945

1673: Gottfried Wilhelm Leibniz améliore la Pascaline pour faire de multiplications et des divisions. Mais, il invente aussi le système binaire et l'arithmétique binaire qui sont à la base des ordinateurs actuels.

En 1805 Joseph Jacquard invente le premier système de programmation. Il s'agit de bandes de carton perforées pour créer de manière automatique des motifs complexes sur un métier à tisser.

En 1833 Charles Babbage invente la machine à différence pour réaliser des tables de calculs et ensuite la *machine analytique* qui permet de réaliser différentes opérations à partir d'un programme établi sur une carte perforée. Elle était cependant irréalisable par les moyens techniques de l'époque. Mais l'idée était très novatrice, en intégrant des principes comme les programmes, le processeur, les entrées et sorties et de la mémoire.

- The Babbage Difference Engine #2 at CHM

En 1854, George Boole invente les bases mathématiques de la logique moderne. L'algèbre de Boole est à la base de la conception de circuits électroniques. C'est de Boole que viens "boolean".

- Wikipedia - Algèbre de Boole (logique)

En 1890, Herman Hollerith construit un calculateur statistique avec des cartes perforées pour accélérer le recensement de la population américaine. 6 ans plus tard il fonde la *Tabluating Machine Company* qui deviendra finalement *International Business Machine Corporation* (IBM) en 1924.

En 1904, John Ambrose Fleming invente le premier "tube à vide" (on va voir plus tard à quoi ça sert)

En 1936, Alan Turing crée l'idée de la *machine de Turing* suseptible de résoudre tout problème calculable. Tout problème qui ne peut être calculé avec la machine de Turing est dit incalculable. Il cree les concepts d'algorithme et de calculabilité

- Computerphile - Turing Machines Explained
- Computerphile - Turing & The Halting Problem

En 1938, Claude Shannon crée la *théorie de l'information* qui fait la synthèse de nombres binaires, de l'algèbre booléenne, et de l'électronique.

- Khan Academy - What is information theory?

En 1936, Konrad Zuse construuit les premiers calculateurs électromécaniques basés sur le système binaire (capable de faire une multiplication en 5 seconde)

De 1944 à 1947, Howard Aiken conçoit le Mark I. Un énorme calculateur électromécanique qui fut ensuite remplacé par le Mark II qui utilise des relais plus rapides. Ces calculateurs sont obsolètes dès leur construction car l'ère de l'électronique commence.

Les tubes à vide (1945-1955)

En 1943, le premier ordinateur digital "COLLOSUS" mais est un secret militaire.

En 1945, l'ENIAC est créé. Voir l'article Wikipedia pour se rendre compte à quel point l'ENIAC est énorme.

- Wikipedia - ENIAC

En 1945, John von Neumann propose une évolution de l'ENIAC appelée EDVAC, notamment pour résoudre le problème majeur de la programmation très laborieuse de l'ENIAC.

Il crée ainsi l'architecture von Neumann où la machine est controlée par un programme dont les instructions sont stoquées en mémoire, le programme pouvant modifier ses propres instructions.

En 1949, Maurice Wilkes construit l'EDSAC, qui est le premier ordinateur basé sur l'architecture von Neumann.

En 1951 construit l'UNIVAC dont les données sont stoquées sur bande magnétiques.

En 1953, IMB lance l'IBM 701 puis ensuite le 704 et 709 et 650.

Les transistors (1955-1965)

Après les tubes à vide, il y eu les transistors. Beaucoup plus rapide, plus fiable et moins cher que les tubes.

C'est aussi à cette période que des languages plus évolués comme le FORTTRAN et le COBOL apparaissent. Ainsi que des composants comme des imprimantes ou des bandes magnétiques.

En 1960, le premier *mini-ordinateur*, le DEC PDP-1 ($120000), ainsi que le premier écran graphique et premier jeu vidéo.

En 1964, premier super ordinateur scientifique (qui introduit la notion de paralléllisme, c'est adire plusieurs unités fonctionnelles travaillant en même temps) et de coprocesseur pour s'occuper des tâches et des entrées-sorties

- Wikipedia - Parallélisme (informatique)
- Wikipedia - Coprocessor

En 1965, le DEC PDP-8 ($18000), premier ordinateur de masse avec 50000 exemplaires vendu. Introduit le concept de *bus* pourinterconnecter les différents éléments de l'ordinateur

- Wikipedia - Bus (électricité)

Les circuits intégrés (1965-1970)

Les circuits intégrés sont essentiellement de très petits trasnsitors dans une petite boite. Encore une fois: moins cher, plus fiable et plus rapide

En 1958, le premier circuit imprimé par *Texas Instruments*

En 1964, première gamme d'ordinateurs par IBM qui soit compatibles entre eux mais ayant des puissances croissantes en fonction des utilisateurs (modèles 30/40/50/65). Et multiprogrammation (plusieurs programmes en mémoire) et émulation des modèles précédents (1401 et 7094) par microprogrammation.

En 1969, création de MULTICS et UNIX (qui donnera Linux et macOS)

Microprocesseurs (1970-???)

Toujours pareil, plus petit, moins cher, plus performant et plus fiable.

En 1971, premier microprocesseur (ayant une puissance similaire à l'ENIAC)

En 1973, création du language C pour le développement de UNIX

En 1978, création de la famille de processeur x86 par Intel. Comme une garantie que tous les processeur de cette famille soit compatibles entre eux car le jeu d'instructions est standardisé.

- x86

En 1981, design publique du IBM personal computer permetantde faire un stadnard au niveau de tout l'ordinateur.

En 1984, lancement du Apple Macintosh avec une interface graphique grand public.

L'évolution des performances

A partir des années 70s, avec l'intégration toujours plus poussée des composants sur une puce et l'augmentation de la capacité de calcul

En 1965 Gordon Moore, constate que depuis 1959, la complexité des circuits intégrés à base de transistors double tous les ans à coût constant. Et il postule que cela va continuer ainsi; c'est la première loi de Moore.

En 1975, il revoit sa loi, il précisa sa loi en disant que le nombre de transitors sur une puce de silicium (microprocesseurs, mémoires) doublera tous les deux ans. C'est la deuxième loi de Moore. Qui s'est avérée être assez proche de la réalité jusqu'a 2010.

Avantages et inconvénients de l'intégration à large échelle

Avantages:

- Augmentation de la capacité des puces mémoires
- Augmentation de la vitesse, performance et sophistication des composants (distances plus courte et donc fréquence d'horloge, nombre d'instructions par secondes) plus grande
- Plusieurs composants sur une seule puce

Inconvénients:

- Il n'est pas possible de réduire infiniment la taille d'un transistor, il faut un minimum d'atomes our que ce soit suffisament fiable, et les limites atomiques seront bientot atteintes.
- Une densité forte induit des phénomènes parasites et fuites de courrant
- La quantité d'énergie dissipée devient problématique (ça chauffe trop). C'est à cause de la "friction" des électrons. Les transitors libèrent de l'énergie. Et la quantité d'énergie est liée à la tension et à la fréquence. Mais la tension ne peut dessendre en dessous de 0.9V sans causer des problèmes techniques. Et la fréquence est ainsi plafonée à 5GHz.

Definition d'un ordinateur

Un ordinateur est une machine automatique de traitement de l'information obéissant à des programmes (suites d'instructions arithmétiques et logiques). Il y a donc la partie matérielle et la partie logicielle.

- L'**unité centrale** (processeur, mémoire centrale, gestionaire d'entrée sorties)
- Les **périphériques d'entrée** permettent d'acquérir des informations (par clavier, souris, scanner, etc)
- Les **périphériques de sortie** permettent de restituer des informations (par clavier, souris, scanner, etc)
- Des **périphériques de stockage** (mémoire secondaire) internes ou externes (ex. disque dur, lecteur-graveur CD, clé USB, etc)
- Les **périphériques de communication** soit les cartes réseau (éthernet, bluetooth, wifi) ou les modems (téléphonique, ADSL, RNIS)

Il y a différentes gammes d'ordinateurs (allant du moins cher au plus cher et évolué)

- Les **ordinateurs jetables** qui sont des puces très simples et coutant moins d'un euro produites en très grandes série pour un usage précis (exemple, carte musicale, puce RFID)
- Les **ordinateurs embarqués** est un ordinateur inclu dans un autre appareil dans le but de le piloter (électroménager, MP3, TV, scanner, pace maker, jeux, armement, distributeurs, etc). Ces ordinateurs coûtent en général quelques EUR.
- Les **téléphones intelligents et consoles de jeu** est une catégorie qui se rapproche beaucoup de l'ordinateur classique maus qui peut être réserver à des usages plus précis et à être moins évolutif dans ses performances et gamme de prix.
- Les **ordinateurs personnels**, soit les PC, en général utilisé par un utilisateur à la fois, la gamme de prix varie plus (ex tours, portables, tablettes)
- Les **serveurs** est un ordinateur prévu pour servir des services à plusieurs utilisateurs (stockage de fichier, calcul, serveur Web, etc). En général on entend par là des ordinateurs plus puissant que des ordinateurs personnels, mais il peu aussi il y avoir des "miniserveurs" qui soit tout autant puissant ou moins puissant (ancien ordinateur portable, raspberry pi, etc)
- Les **cluster** (ou **ferme de serveurs** ou **data center**) sont prévu pour fournir une puissance de calcul et/ou une grande capacité de stockage en connectant plusieurs serveurs entre eux pour se répartir les taches.
- Les **mainframes** (ou **super-ordinateurs**) avais des architectures très spécifiques, prévu pour un usage très particulier avec une puissance de calcul très grande (qui sont maintenant remplacé par les data centers qui sont plus performant et coute moins cher). En revanche il existe encore des mainframes, principalement dans le secteur bancaire ou une migration serait trop couteuse.

Le **cloud** n'est rien d'autre qu'un ou plusieurs serveurs ou fermes de serveurs qui fournissent des services à travers internet. L'interet est de mutualiser des ressources, rendre l'information accessible de n'importe où et de synchroniser les données à un moindre coût.

Il existe différents types de *services cloud* :

- Le **IaaS** (Infrastructure as a Service) qui comprends le réseau, le serveur + stockage et la virtualisation (souvent ce ne sont pas des ordinateurs dédiés mais des machines virtuelles tournant sur de gros ordinateurs)
- Le **PaaS** (Platform as a Service) qui comprends réseau, serveur + stockage, virtualisation mais aussi le système d'exploitation et les logiciels de bases (par exemple, serveur web)
- Le **SaaS** (Software as a Service) qui comprends réseau, serveur + stockage, virtualisation, système d'exploitation, logiciel de base, et logiciel cible (par exemple webapp) déjà installé pret à être utilisé.

Représentation interne des informations

Un bit est représenté par un 0 ou 1. Allumé ou éteint.

Un octet (ou *byte* en anglais) corresponds à 8 bits.

Un *nibble*, moins courrant que l'octet est un demi-octet (4 bits)

Les cellules mémoires sont elles aussi organisées en blocs de taille plus importante, qui n'est pas standardisée et dépends du processeur utilisé. Elle peut être de 16, 32 ou 64 bits.

Pour savoir combien de configurations peuvent exister avec x nombres de bits, on peut utiliser la formule :

$$ 2^x $$

Pour connaitre l'inverse, le nombre de bits nécessaire pour représenter y nombre de configurations (arrondis à la hausse) :

$$ log_2(x) $$

La signification d'une séquence binaire dépends du contexte. Sans ce contexte, il est impossible de savoir à quoi corresponds une séquence binaire.

Les types de variables déclarent une zone mémoire de X bits. Par exemple en java `int` corresponds à une zone mémoire de 64 bits

Attention ! Il faut bien faire attention qu'un nombre ne peut pas être représenter si il prends plus de bits que le nombre de bits disponibles dans l'espace mémoire (exemple, 128 ne peut pas être représenté dans un espace de 7 bits). Il faut donc toujours vérifier que le nombre n'excède pas le nombre maximal.

Représentation des nombres entiers

BCD (Binary Coded Decimal)

Pour faciliter la vie du programmeur (quand l'assembleur n'était pas encore très utilisé et que les données étaient manuellement codées en binaire) le système "BCD" (Binary Coded Decimal) a été créé.

A la place de coder un nombre en binaire directement :

```
14 (decimal) = 1110 (binaire)
```

On va le codé par équivalent décimal (soit on converti chaque chiffre décimal en nombre binaire) :

```
14 (decimal) = 0001 0100 (bcd)
```

Ce système, bien que plus facile pour le programmeur est plus complexe pour le processeur, donc avec l'arrivée de l'assembleur et autres, ce système est vite abandonné.

Représentation de nombres naturels

L'espace mémoire pour la représentation d'un nombre est toujours un nombre entier d'octet, la taille dépends de la valeur représentée :

Bits	Nom FR	Nom EN
8	Octet	Byte
16	Entier court	Short integer
32	Entier	Integer
64	Entier long	Long integer

Si un nombre représenté laisse des emplacements libres sur la gauche, on les remplace par des 0 pour entrer dans l'emplacement mémoire prévu.

Par exemple pour représenter 5 dans un emplacement mémoire de 8 bits. On le transforme en binaire et on obtient 101 mais pour le faire entrer dans un octet cela va être 00000101.

- Pour connaitre le nombre de configurations possible de $ n $ bits :

$$ 2^n $$

- Pour connaitre le nombre maximal pouvant être représenté en $ n $ bits (on diminue le nombre de configuration de 1, car il y a le 0)

$$ 2^n - 1 $$

- Pour savoir combien de bits il faut avoir pour représenter des nombres de 0 à $ x $:

$$ log_2(x+1) $$

- Pour savoir la caractéristique binaire d'un multiple d'un nombre $ n $ divisible par 2, il terminera par $ log_2(n) $ zéros.
- Comment savoir comment évolue un nombre $ n $ qui est multiplié ou divisé par un multiple de 2. Il suffit de décaler les nombres vers la droite ou vers la gauche de $ log_2(n) $
- Comment faire une division avec reste d'un nombre $ n $ multiple de 2. Il faut regarder les $ log_2(n) $ bits de droite (premiers), ces bits représentent le Reste, tandis que les autres bits de droites seront le Quotient.

Représentation des entiers relatifs

```
74  : 01001010
-74 : 11001010
```

Pour représenter des chiffres négatifs aussi bien que positif on peut réserver un bit au début de l'espace mémoire pour le signe. Si ce bit est à 1, le chiffre est négatif, sinon, il est positif.

L'un des problèmes est que les opérations arithmétiques ne sont pas facile et qu'il existe la valeur +0 et -0 qui n'ont pas de sens

- Pour savoir le nombre minimal et maximal de nombres pouvant être représenté en n bits. On va de -2^{n-1} à $+2^{n-1}$. On soustrait 1 car il y a maintenant un bit utilisé pour représenter le signe.
- Pour savoir le nombre maximal de chiffres pouvant être représentés.

$$ 2^{n-1} * 2 $$

Complément à 1

```
74 en décimal   : 01001010
-74 en comp à 1 : 10110101 (c'est l'inversion de tout les bits)
```

Pour faciliter les opérations arithmétiques nous pouvons utiliser une autre méthode. Celle d'inverser tous les bits.

Donc pour représenter $ 74_{10} $ sur 8 bits en complément à 1, on obtient `01001010` et pour changer le signe, on inverse tous les bits ce qui donne donc `10110101`. Le bit de signe est donc toujours en action.

Complément à 2

```
74                   : 01001010
-74 en complément à 1 : 10110101 (inversion de tous les bits)
-74 en complément à 2 : 10110110 (complément à 1 + 1)
```

Le complément à 2 corresponds au complément à 1 vu précédemment + 1. Mais il peut être calculé plus rapidement simplement en inversant tout à partir du premier 1 en partant de la gauche:

```
74                    : 01001010
-74 en complément à 2 : 10110110 (on inverse inverse tout jusqu'au premier 1 de gauche)
```

Représentation des données non numériques

Il va très vite aussi être nécessaire de réprésenter d'autres données que des nombres.

On a donc inventer plusieurs codes *alphanumériques*. C'est à dire des codes qui permettent de représenter des chiffres, des lettres, des signes de ponctuation, etc.

Chaque code lie un symbole à un nombre. Ainsi on utilise des "tables de codage" pour savoir quel nombre correspond à tel caractère.

EBCDIC (Extended Binary Coded Decimal Interchange Code)

Ce code est une extension du "BCD" (Binary Coded Decimal) mais sur 8 bits (4 bits de "zone" et 4 bits numériques)

Caractère	Bits de zone	Bits numériques	Hexa
0	`1111`	`0000`	`F0`
1	`1111`	`0001`	`F1`
2	`1111`	`0010`	`F2`
3	`1111`	`0011`	`F3`
4	`1111`	`0100`	`F4`
5	`1111`	`0101`	`F5`
6	`1111`	`0110`	`F6`
7	`1111`	`0111`	`F7`
8	`1111`	`1000`	`F8`
9	`1111`	`1001`	`F9`
A	`1100`	`0001`	`C1`
B	`1100`	`0010`	`C2`
C	`1100`	`0011`	`C3`

Caractère	Bits de zone	Bits numériques	Hexa
D	`1100`	`0100`	`C4`
E	`1100`	`0101`	`C5`
F	`1100`	`0110`	`C6`
G	`1100`	`0111`	`C7`
H	`1100`	`1000`	`C8`
I	`1100`	`1001`	`C9`
J	`1101`	`0001`	`D1`
K	`1101`	`0010`	`D2`
L	`1101`	`0011`	`D3`
M	`1101`	`0100`	`D4`
N	`1101`	`0101`	`D5`
O	`1101`	`0110`	`D6`
P	`1101`	`0111`	`D7`
Q	`1101`	`1000`	`D8`
R	`1101`	`1001`	`D9`

ASCII (American Standard Code for Information Interchange)

Le code ASCII beaucoup plus connu et utilisé permet de stoquer 128 caractères sur 7 bits.

ASCII table

Ensuite le code ASCII fut stoqué sur 8 bits car IBM a ajouté 127 codes supplémentaires pour d'autres caractères plus spécifiques (mais pas supportés par tous les ordinateurs).

Les codes 0 à 31 sont des caractères de contrôle pour la transmission des données ou l'affichage sur un terminal (tab, retour à la ligne, etc).

Unicode et UTF-8

Le code ASCII étant d'origine américaine, beaucoup de caractères n'étaient pas supportés (par exemple, les caractères accentués).

L'organisation internationale de normalisation (ISO) a donc tenté de créer un standard pour l'échange de textes dans différentes langues : l'Unicode.

L'une des formes les plus utilisées (surtout sur internet) de ce standard est le code UTF-8 qui est extensible est compatible avec l'ASCII.

L'UTF-8 est à taille variable, les caractères peuvent être représenté sur 1, 2, 3 ou 4 octets. Si le caractère est représenté sur 1 octet, alors c'est un caractère ASCII.

Ainsi l'UTF-8 permet de représenter 149 186 caractères différents (et a la possibilité de coder au total 2 097 152 caractères différents).

Représentation des données dans les languages de programmation

Chaque language a sa propre manière de représenter ses données. Par exemple, en C une chaine de caractère est stoquée comme étant une succession de caractères séparés par des caractères sépciaux.

En savoir plus

- Wikipedia - Unicode
- Wikipedia - UTF-8 (explique aussi comment connaitre le nombre d'octet d'un caractère)

Détection et résolution d'erreurs

https://www.youtube-nocookie.com/embed/ecw8AbBKZCM

Maintenant que l'on sait représenter des nombre et des caractères en binaire, on peut maintenant voir comment faire dans les cas où il pourrait y avoir des erreurs.

Que ce soit dans la mémoire ou lors de communication, ce n'est pas infaillible et 1 seul bit qui change pourrait changer complètement l'information.

Le bit de parité (code auto-verificateur)

Supposons que notre code est le suivant : `01001000` pour calculer la parité paire, il suffit de compter le nombre de `1`. Dans ce cas il y a 2 fois `1`. Donc le bit de parité est `0`.

Si on compte un nombre impair de `1`, alors le bit de parité est `1`. En bref, il faut que le nombre total de `1` (en comptant à la fois le message et le bit de parité) soit **pair**.

Et pour la parité impaire, il suffit de faire l'inverse. On s'assure donc que le nombre total de `1` (en comptant message + bit de partité) soit **impair**.

Ainsi maintenant on peut vérifier un code et savoir si il a été altéré ou non. En revanche cette méthode est limitée, car si il y a un nombre pair d'erreurs, alors le bit de parité ne dédectera rien d'anormal. Aussi, on n'a aucune idée de où se trouve l'erreur.

La double parité (code auto-correcteur)

Maintenant imaginons que l'on a un message plus long. Comme `SHUMI` par exemple. On va donc le transcrire en binaire (avec le code UTF-8)

Car	b7	b6	b5	b4	b3	b2	b1	b0
S	0	1	0	1	0	0	1	1
H	0	1	0	0	1	0	0	0
U	0	1	0	1	0	1	0	1
M	0	1	0	0	1	1	0	1
I	0	1	0	0	1	0	0	1

Maintenant on va faire une parité *paire* horizontalement et une parité *impaire* verticalement comme vu précédemment..

Car	b7	b6	b5	b4	b3	b2	b1	b0	PP
S	0	1	0	1	0	0	1	1	0
H	0	1	0	0	1	0	0	0	0
U	0	1	0	1	0	1	0	1	0
M	0	1	0	0	1	1	0	1	0
I	0	1	0	0	1	0	0	1	1
PI	1	0	1	1	0	1	0	1	

Maintenant en vérifiant tous les bits de parités on peut retrouver une erreur facilement et la corriger

Car	b7	b6	b5	b4	b3	b2	b1	b0	PP
S	0	1	0	1	0	0	1	1	0
H	0	1	0	0	1	0	0	0	0
E	0	1	0	**0**	0	1	0	1	**0**
M	0	1	0	0	1	1	0	1	0
I	0	1	0	0	1	0	0	1	1
PI	1	0	1	**1**	0	1	0	1	

On peut voir ici qu'il y a eu une erreur au niveau du bit 4 sur le 3e caractère car la parité paire de la ligne et la parité impaire de la colonne ne sont plus correct. A cause de ça notre `U` est devenu un `E`.

Mais on sait donc précisément quel bit est la cause de l'erreur et comment le résoudre. En revanche cette méthode nécessite beaucoup de bits de vérification.

Le code de Hamming

Ici on se concentre sur le code de Hamming qui permet de corriger 1 bit en erreur, il existe des variantes plus complexes mais que l'on apprend pas ici.

Création du code de Hamming

Disons que l'on veut stoquer le code *S*, soit `1010011` en binaire.

Tout d'abord on va faire un tableau pour contenir notre code, mais qui va être un peu plus long pour contenir nos bits de parités

11	10	9	8	7	6	5	4	3	2	1

Maintenant on va bloquer tous les emplacements qui sont des puissances de 2 (qui sont nos bits de parités)

11	10	9	8	7	6	5	4	3	2	1
			k4				*k3*		*k2*	*k1*

Ensuite on peut écrire le message à coder dans les bits restants :

11	10	9	8	7	6	5	4	3	2	1
1	0	1	*k4*	0	0	1	*k3*	1	*k2*	*k1*

Maintenant on liste tous les numéros de bits qui valent `1` dans le tableau et on les convertis en binaire

```
11 → 1011
9  → 1001
5  → 0101
3  → 0011
```

Ensuite pour chaque colonne en binaire on va faire une parité de la colonne. Chaque bit de parité trouvé corresponds aux bits de parités dans le tableau.

```
11 → 1011
9  → 1001
5  → 0101
3  → 0011
     ----
     0100 (k4 = 0, k3 = 1, k2 = 0, k1 = 0)
```

On peut donc remplacer ces derniers dans le tableau

11	10	9	8	7	6	5	4	3	2	1
1	0	1	**0**	0	0	1	**1**	1	**0**	**0**

Notre code de Hamming est donc : `10100011100`

Vérification et correction dans un code de Hamming

Maintenant quand on veut vérifier un code reçu, voici comment faire. On va partir du code `10101011100` et on va le vérifier (et si besoin corriger)

Tout d'abord on va placer tout ceci dans un tableau avec les bits numérotés :

11	10	9	8	7	6	5	4	3	2	1
1	0	1	0	1	0	1	1	1	0	0

Maintenant on liste tous les bits qui sont a 1 et on convertis leur position en binaire :

```
11 → 1011
9  → 1001
7  → 0111
5  → 0101
4  → 0100
3  → 0011
```

Ensuite, on va faire la parité de chaque colonne. Le résultat corresponds à la position du bit en erreur en binaire. Si le résultat done `0` partout, alors il n'y a pas d'erreur détectée.

```
11 → 1011
9  → 1001
7  → 0111
5  → 0101
4  → 0100
3  → 0011
     ----
     0111 → 7 (le bit 7 doit donc être inversé)
```

Nous pouvons donc corriger notre code :

11	10	9	8	7	6	5	4	3	2	1
1	0	1	0	**0**	0	1	1	1	0	0

Pour obtenir le message d'origine il suffit de retirer tous les bits qui sont à des positions qui correspondent à des puissances de 2

11	10	9	8	7	6	5	4	3	2	1

1	0	1		0	0	1		1		

Ce qui nous donne donc le message suivant : `1010011` soit `S` en ASCII. Nous retrouvons donc le même résultat que le message encodé au départ □□

Les opérateurs logiques

https://www.youtube-nocookie.com/embed/7mLNb8XIRHw

> Cette matière étant globalement la même que celle vu en math, je vous renvois donc vers ma synthèse de math sur le sujet : (Math) Les connecteurs logiques de base.

Symbole électronique	Nom électronique	Formule mathématique
$ \overline{a} $	`NOT` a	$ \neg a $
$ ab $ ou $ a * b $	a `AND` b	$ a \wedge b $
$ a + b $	a `OR` b	$ a \vee b $
$ a \oplus b $	a `XOR` b	$ a \oplus b $

Ensuite il y a la négations des portes précédentes :

Symbole électronique	Nom électronique	Formule mathématique
$ a \downarrow b $ ou $ \overline {a + b} $	a `NOR` b	$ \neg (a \vee b) $
$ a \uparrow b $ ou $ \overline {ab} $	a `NAND` b	$ \neg (a \wedge b) $
$ \overline {a \oplus b} $ ou $ a \iff b $	a `XNOR` b	$ a \iff b $

Et voici à quoi correspondent ces portes dans des schémas électroniques :

symboles électroniques

Comment construire les portes logiques avec des transistors

Pour en savoir plus, j'ai trouvé 2 vidéos en anglais qui expliquent comment fonctionnent les transistors et les résistances :

- Une vidéo avec animation 3D

- Une vidéo qui explique toutes les portes logiques avec un schéma électronique

2 petites informations pour mieux comprendre les vidéos :

- Un transistor se comporte comme un interrupteur mais activé de manière électronique
- Une résistance crée une différence de tension. Donc dans un circuit fermé, avant la résistance la tension serait de 5V et après elle serait de 0V.

Les formes normales

Les formes normales permettent de représenter toute fonction logique avec uniquement des AND, OR et NOT.

La première forme normale (disjonctive)

1. On construit la table de vérité de la fonction

a	b	xor
0	0	0
0	1	1
1	0	1
1	1	0

2. On se concentre uniquement sur les fois où la fonction vaut 1

a	b	xor
0	1	1
1	0	1

3. On met un opérateur `AND` entre les deux inputs et on remplace le 0 par une négation

> $ \overline{a} b $ | $ a \overline{b} $

4. On sépare les différents résultat de l'étape précédente par des `OR`

> Première forme normale de XOR : $ \overline{a} b + a \overline{b} $

La deuxième forme normale (conjonctive)

1. On construit la table de vérité de la fonction

a	b	xor
0	0	0
0	1	1
1	0	1
1	1	0

2. On se concentre uniquement sur les fois où la fonction vaut 0

a	b	xor
0	0	0
1	1	0

3. On met un opérateur `OR` entre les deux inputs et on rempalce le 1 par une négation

> $a + b$ | $\overline{a} + \overline{b}$

4. On sépare les différents résultats de l'étape précédente par des `AND`

> Deuxième forme normale de XOR : $ (a + b)(\overline{a} + \overline{b}) $

La mémoire cache

https://www.youtube-nocookie.com/embed/K9h6-NiQrMU

Correspondance directe

Voici un exemple d'exercice :

Ligne	Init	18	27	3
0	30			
1	7			
2	0			
3	9			
4	0			
5	0			

Pour la correspondance directe il faut pour chaque colonne, le modulo du nombre de lignes.

Dans l'exemple ici, la première colonne est `18` et il y a 6 lignes. On fait donc $18 \mod 6$ ce qui donne `0`. On met donc 18 dans la ligne 0. Ainsi de suite pour les autres.

Ligne	Init	18	27	3
0	30	**18**		
1	7			
2	0			
3	9		**27**	**3**
4	0			
5	0			

Enfin on complète le tableau en recopiant à chaque fois la case de gauche.

Ligne	Init	18	27	3
0	30	**18**	18	18
1	7	7	7	7

Ligne	Init	18	27	3
2	0	0	0	0
3	9	9	**27**	**3**
4	0	0	0	0
5	0	0	0	0

Complètement associative avec remplacement circulaire

Schéma *Schéma honteusement subtilisé à Melisa*

Celui ci est un peu plus subtile que le précédent, pour chaque colonne il faut :

1. Vérifier si le nombre est déjà dans la colonne précédente, si oui, on recopie juste la colonne
2. Sinon, pour le premier remplacement, il faut mettre le nombre à la première ligne qui est à 0
3. A chaque *remplacement* (et pas colonne), on va placer le nombre sur la ligne juste en dessous
4. On complète le reste des cases par les cases de la colonne précédente (comme précédemment)

Associative par ensemble avec remplacement circulaire

Exercice syllabus

Ici, c'est un mélange des deux précédents, ici il y a plusieurs sets, et chaque set a 2 lignes. Pour faire ceci, pour chaque colonne :

1. Faire le modulo de la colonne par le nombre de sets (exemple avec le premier : $22 \mod 4 = 2$, donc il faudra écrire 22 dans le set 2)
2. Si il y a une ligne vide (0 ou -) parmis le set en question on place le nombre là (exemple avec le premier : la deuxième ligne du set 2 est -, c'est donc là que l'on place notre nombre)
3. Sinon, si le nombre est déjà inscrit dans le set, on réécrit juste la colonne
4. Sinon, on écrit le nombre à la ligne opposée à celle du dernier remplacement (exemple: le nombre 21 a été placé en bas du set 1, 25 (le remplacement suivant) est donc placé en

haut).

Le système RAID

Les disques dur de grande capacité étant très cher. Le système RAID avait à la base été conçu pour créer un « gros disque » (appellé *grappe*), sur base de disque plus petit.

RAID signifie *Redundant Array of Inexpensive Disks*, puis qui fut plus tard renomé en *Redundant Array of Independent Disks*. Car avec l'évolution des technologie, le but des système RAID a changé.

RAID 0

RAID 0

Le but de RAID 0 est de grouper plusieurs disques physique pour créer un volume de plus grande capacité. Cela permet aussi d'avoir des meilleurs performances, en **écriture**, en écrivant sur plusieurs disques en parallèle.

On peut donc aussi avoir un niveau de **lecture** en lisant des morceaux simultanément.

Le niveau RAID 0 n'offre *aucune* tolérence aux pannes, au contraire, il diminue la fiabilité globale car il suffit qu'un seul disque tombe en panne pour que toutes les données soient perdues.

RAID 1

RAID 1

Le but de RAID 1 est d'offrir une tolérance aux pannes en dupliquant les données de chaque disque sur un autre disque (ainsi il y a toujours un nombre pair de disques).

Ce système permet aussi une meilleur performance en **lecture** en pouvait lire simultanément la même donnée sur plusieurs disques.

RAID 2

RAID 2

Ce niveau n'est plus utilisé. il utilisait le code de Hamming pour protéger les données contre la perte d'un disque. Chaque bit d'un octet est stoqué sur un disque différent.

RAID 3

RAID 3

Ce niveau est une version simplifiée du niveau 2. Il fonctionne en ajoutant simplement 1 bit de parité (et donc un disque). Ainsi quand un disque tombe en panne et est remplacé on peut "réparer" les données grâce au bit de parité.

RAID 4

RAID 4

Les données sont écrites bloc par bloc de manière circulaire sur N disques (et non plus bit par bit comme dans les 2 niveaux précédents)

Datapath

> **Attention !**, cette synthèse est encore incomplète. Je vais faire une vidéo et de meilleures explications plus tard.

PC contient une addresse (qui va dans la mémoire d'instructions) Add4 calcule l'instruction suivante L'autre add et ajoute encore pour un branchement Register pour lire et faire des opérations ou écrire l'ALU réalise une opération (comparaison et arithmétique) Sign-extend pour les constantes Mémoire de données (load et store)

Toujours utilisé : PC, mémoire d'instructions, add4

Banque de registre quand l'instruction fait référence à un registre (R dans la greencard) ALU est utilisée si opération (+ - * / > < << >>>) Mémoire de données (M dans la greencard) Sign-extend utilisée si SignExtImmm dans la greencard Shift left 2 et additionneur de droite si branchement (BranchAddr dans greencard)

Signaux actifs ou inactifs

- RegWrite si R à gauche du =
- RegRead si R à droite du =
- MemWrite si M à gauche du =
- MemRead si M à droite du =

Multiplexeurs

- premier : On regarde à gauche du = (si rd alors bas, si rt alors haut)
- deuxiem : si opération de deux registres alors haut. Sinon bas pour constante
- troisie : si pas de branchement: haut. Si branchement: bas. Attnetion il faut connaitre le résultat de la comparaison
- quatrie : si on écrit dans un registre : si ALU → bas. sinon (si vient de mémoire) → haut