

Un livre de Wikilivres.

Fonctionnement d'un ordinateur

Une version à jour et éitable de ce livre est disponible sur Wikilivres,
une bibliothèque de livres pédagogiques, à l'URL :
http://fr.wikibooks.org/wiki/Fonctionnement_d'un_ordinateur

Vous avez la permission de copier, distribuer et/ou modifier ce document selon les termes de la Licence de documentation libre GNU, version 1.2 ou plus récente publiée par la Free Software Foundation ; sans sections inaltérables, sans texte de première page de couverture et sans Texte de dernière page de couverture. Une copie de cette licence est incluse dans l'annexe nommée « Licence de documentation libre GNU ».

Encodage, traitement, décodage

Un ordinateur peut manipuler diverses formes d'information : du texte, de la vidéo, du son, et plein d'autres choses encore. Eh bien, sachez que tout cela est stocké... avec des nombres. Aussi bizarre que cela puisse paraître, un ordinateur n'est qu'une sorte de grosse calculatrice hyper-performante. Et pourtant cela suffit pour permettre d'écouter de la musique, de retoucher des images, de créer des vidéos, etc. Mais dans ce cas, comment faire la correspondance entre ces nombres et du son, du texte, ou toute autre forme d'information ? Et comment fait notre ordinateur pour stocker ces nombres et les manipuler ? Nous allons répondre à ces questions dans ce chapitre.

Le codage de l'information

Toute information présente dans un ordinateur est décomposée en petites informations de base, chacune représentée par un nombre. Par exemple, le texte sera décomposé en caractères (des lettres, des chiffres, ou des symboles). Pareil pour les images, qui sont décomposées en pixels, eux-mêmes codés par un nombre. Même chose pour la vidéo, qui n'est rien d'autre qu'une suite d'images affichées à intervalles réguliers. La façon dont un morceau d'information (lettre ou pixel, par exemple) est représenté avec des nombres est définie par ce qu'on appelle un codage, parfois appelé improprement encodage. Ce codage va attribuer un nombre à chaque morceau d'information. Pour montrer à quoi peut ressembler un codage, on va prendre trois exemples : du texte, une image et du son.

Texte : standard ASCII

Pour coder un texte, il suffit de savoir coder une lettre ou tout autre symbole présent dans un texte normal (on parle de **caractères**). Pour coder chaque caractères avec un nombre, il existe plusieurs codages : l'ASCII, l'Unicode, etc. Le codage le plus ancien, l'ASCII, est intégralement défini par une table de correspondance entre une lettre et le nombre associé : la table ASCII. De nos jours, on utilise des codages de texte plus complexes, afin de gérer plus de caractères. Là où l'ASCII ne code que l'alphabet anglais, les codages actuels comme l'Unicode prennent en compte les caractères chinois, japonais, grecs, etc.

Image

Le même principe peut être appliquée aux images : l'image est décomposée en morceaux de même taille qu'on appelle des **pixels**. Chaque pixel a une couleur qui est codée par un nombre entier. Pour stocker une image dans l'ordinateur, on a besoin de connaître sa largeur, sa longueur et la couleur de chaque pixel. Une image peut donc être représentée dans un fichier par une suite d'entiers : un pour la largeur, un pour la longueur, et le reste pour les couleurs des pixels. Ces entiers sont stockés les uns à la suite des autres dans un fichier. Les pixels sont stockés ligne par ligne, en partant du haut, et chaque ligne est codée de gauche à droite. Les fichiers images actuels utilisent des techniques de codage plus élaborées, permettant notamment décrire une image en utilisant moins de nombres, ce qui prend moins de place dans l'ordinateur.

Son

Pour mémoriser du son, il suffit de mémoriser l'intensité sonore reçue par un microphone à intervalles réguliers. Cette intensité est codée par un nombre entier : si le son est fort, le nombre sera élevé, tandis qu'un son faible se verra attribuer un entier petit. Ces entiers seront rassemblés dans l'ordre de mesure, et stockés dans un fichier son, comme du wav, du PCM, etc. Généralement, ces fichiers sont compressés afin de prendre moins de place.

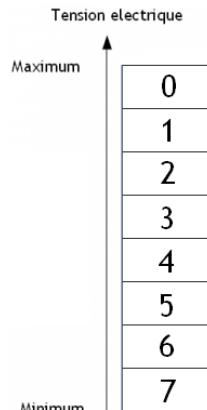
Analogique, numérique et binaire

Dans les grandes lignes, on peut identifier deux grands types de codages. Le **codage analogique** utilise des nombres réels : il code l'information avec des grandeurs physiques (des trucs qu'on peut mesurer par un nombre) comprises dans un intervalle. Un codage analogique a une précision théoriquement infinie : on peut par exemple utiliser toutes les valeurs entre 0 et 5 pour coder une information. Celle-ci peut alors prendre une valeur comme 1, 2,2345646, ou pire... Un calculateur analogique utilise le codage analogique. Dans un monde parfait, il peut faire des calculs avec une précision très fine, et peut même faire certains calculs avec une précision théorique impossible à atteindre avec un calculateur numérique : des dérivées, des intégrations, etc.

Le **codage numérique** va coder des informations en utilisant des nombres entiers, représentés par des suites de chiffres. Pour donner une définition plus générale, le codage numérique utilise qu'un nombre fini de valeurs, contrairement au codage analogique. Cela donnera des valeurs du style : 0, 0,12, 0,24, 0,36, 0,48... jusqu'à 2. Pour les calculateurs numériques, les nombres manipulés sont codés par des suites de chiffres.

Mais peu importe le codage utilisé, celui-ci a besoin d'un support physique, d'une grandeur physique quelconque : le codage analogique a besoin de quelque chose pour mémoriser un nombre, tandis que le codage numérique a besoin de quelque chose pour représenter un chiffre. Et pour être franc, on peut utiliser tout et n'importe quoi. Par exemple, certains calculateurs assez anciens étaient des calculateurs pneumatiques, et utilisaient la pression de l'air pour représenter des chiffres ou nombres : soit le nombre encodé était proportionnel à la pression (codage analogique), soit il existait divers intervalles de pression pour chaque chiffre (codage numérique). De nos jours, on utilise soit l'aimantation d'un support magnétique, soit des tensions.

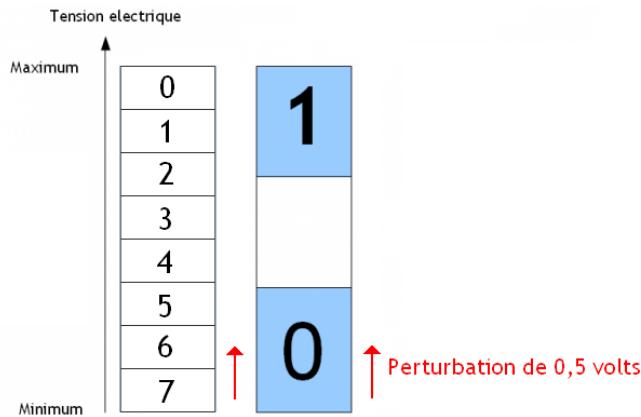
Pour les supports de stockage électroniques, très courants dans nos ordinateurs, le support en question est une **tension électrique**. Ces tensions sont manipulées par différents circuits électroniques plus ou moins sophistiqués. Ces circuits ont besoin d'être alimentés en énergie. Pour cela, notre circuit possédera une tension qui alimentera le circuit en énergie, qui s'appelle la **tension d'alimentation**. Après tout, il faudra bien que notre circuit trouve de quoi fournir une tension de 2, 3, 5 volts : la tension codant un chiffre ne sort pas de nulle part ! De même, on a besoin d'une tension de référence valant zéro volt, qu'on appelle la **masse**, qui sert pour le zéro. En règle générale, le codage utilisé est un codage numérique : si la tension est dans tel intervalle, alors elle code tel chiffre.



De nos jours, tous les appareils électroniques et ordinateurs utilisent un codage numérique ! C'est dû au fait que les calculateurs analogiques sont plus sensibles aux perturbations électromagnétiques (on dit aussi ils ont une faible immunité au bruit). Explication : un signal analogique peut facilement subir des perturbations qui vont changer sa valeur, modifiant directement la valeur des nombres stockés/manipulés. Avec signal numérique, les perturbations ou parasites vont moins perturber le signal numérique. La raison est qu'une variation de tension qui reste dans un intervalle représentant un chiffre ne changera pas sa valeur. Il faut que la variation de tension fasse sortir la tension de l'intervalle pour changer le chiffre.

Codage des nombres

Dans le chapitre précédent, nous avons vu que les ordinateurs actuels utilisent un codage numérique. Cependant, il ne faut pas croire que ceux-ci comptent en décimal. Les ordinateurs n'utilisent que deux chiffres, 0 et 1 : on dit qu'ils comptent en binaire. La raison à cela est simple : cela permet de mieux résister aux perturbations électromagnétiques mentionnées dans le chapitre précédent. À tension d'alimentation égale, les intervalles de chaque chiffre sont plus petits pour un codage décimal : toute perturbation de la tension aura plus de chances de changer un chiffre. Mais avec des intervalles plus grands, un parasite aura nettement moins de chance de modifier la valeur du chiffre codé ainsi. La résistance aux perturbations électromagnétiques est donc meilleure avec seulement deux intervalles.



Ce codage binaire ne vous est peut-être pas familier. Aussi, dans ce chapitre, nous allons apprendre comment coder des nombres en binaire. Nous allons commencer par le cas le plus simple : les nombres positifs. Par la suite, nous aborderons les nombres négatifs. Et nous terminerons par les nombres flottants.

Nombres entiers positifs

Pour coder des nombres entiers positifs, il existe plusieurs méthodes :

- le binaire ;
- le code Gray ;
- le décimal codé binaire.

Binaire

Prenons un nombre écrit en décimal, comme vous en avez l'habitude depuis le primaire. Le chiffre le plus à droite est le chiffre des unités, celui à côté est pour les dizaines, suivi du chiffre des centaines, et ainsi de suite. Dans un tel nombre :

- on utilise une dizaine de chiffres, de 0 à 9 ;
- chaque chiffre est multiplié par une puissance de 10 : 1, 10, 100, 1000, etc. ;
- la position d'un chiffre dans le nombre indique par quelle puissance de 10 il faut le multiplier : le chiffre des unités doit être multiplié par 1, celui des dizaines par 10, celui des centaines par 100, et ainsi de suite.

Exemple avec le nombre 1337 :

$$1337 = 1 \times 1000 + 3 \times 100 + 3 \times 10 + 7 \times 1$$

Pour résumer, un nombre en décimal s'écrit comme la somme de produits, chaque produit multipliant un chiffre par une puissance de 10.

$$1337 = 1 \times 10^3 + 3 \times 10^2 + 3 \times 10^1 + 7 \times 10^0$$

Ce qui peut être fait avec des puissances de 10 peut être fait avec des puissances de 2, 3, 4, 125, etc : n'importe quel nombre entier strictement positif peut servir de base. En binaire, on utilise des puissances de deux : n'importe quel nombre entier peut être écrit sous la forme d'une somme de puissances de 2. En binaire, on n'utilise que deux chiffres, à savoir 0 ou 1 : ces chiffres binaires sont appelés des **bits** (abréviation de Binary Digit).

Pour traduire un nombre binaire en décimal, il faut juste se rappeler que la position d'un bit indique par quelle puissance il faut le multiplier. Ainsi, le chiffre le plus à droite est le chiffre des unités : il doit être multiplié par 1 (2^0). Le chiffre situé immédiatement à gauche du chiffre des unités doit être multiplié par 2 (2^1). Le chiffre encore à gauche doit être multiplié par 4 (2^2), et ainsi de suite. Mathématiquement, on peut dire que le *énième* bit en partant de la droite doit être multiplié par $2^{(n-1)}$. Par exemple, la valeur du nombre noté 1011 en binaire est de $(8 \times 1) + (4 \times 0) + (2 \times 1) + (1 \times 1) = 11$. Le **bit de poids faible** est celui qui est le plus à droite du nombre, alors que le **bit de poids fort** est celui non nul qui est placé le plus à gauche.

La traduction inverse, du décimal au binaire, demande d'effectuer des divisions successives par deux. Les divisions en question sont des divisions euclidiennes, avec un reste et un quotient. En lisant les restes des divisions dans un certain sens, on obtient le nombre en binaire. Voici comment il faut procéder, pour traduire le nombre 34 :

$34/2 = 17$	reste : 0	↑ Sens dans lequel lire les bits
$17/2 = 8$	reste : 1	
$8/2 = 4$	reste : 0	
$4/2 = 2$	reste : 0	
$2/2 = 1$	reste : 0	
$1/2 = 0$	reste : 1	

Résultat : 0100010

Hexadécimal

L'hexadécimal est basé sur le même principe que le binaire, sauf qu'il utilise les 16 chiffres suivants :

Chiffre hexadécimal	Nombre décimal correspondant	Notation binaire
0	0	0
1	1	1
2	2	01
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Afin de différencier les nombres écrits en décimal des nombres hexadécimaux, les nombres hexadécimaux sont souvent suivis par un petit h, indiqué en indice. Si cette notation n'existe pas, des nombres comme 2546 seraient ambigus : on ne sait pas dire sans autre indication s'ils sont écrit en décimal ou en hexadécimal. Par contre, on sait de suite que 2546 est en décimal et que 2546h est en hexadécimal.

Pour convertir un nombre hexadécimal en décimal, il suffit de multiplier chaque chiffre par la puissance de 16 qui lui est attribué. Là encore, la position d'un chiffre indique par quelle puissance celui-ci doit être multiplié : le chiffre le plus à droite est celui des unités, le second chiffre le plus à droite doit être multiplié par 16, le troisième chiffre en partant de la droite doit être multiplié par 256 (16 * 16) et ainsi de suite. La technique pour convertir un nombre décimal vers de l'hexadécimal est similaire à celle utilisée pour traduire un nombre du binaire vers le décimal. On retrouve une suite de divisions successives, mais cette fois-ci les divisions ne sont pas des divisions par 2 : ce sont des divisions par 16.

La conversion entre hexadécimal et binaire est très simple, nettement plus simple que les autres conversions. Pour passer de l'hexadécimal au binaire, il suffit de traduire chaque chiffre en sa valeur binaire, celle indiquée dans le tableau au tout début du paragraphe nommé « Hexadécimal ». Une fois cela fait, il suffit de faire le remplacement. La traduction inverse est tout aussi simple : il suffit de grouper les bits du nombre par 4, en commençant par la droite (si un groupe est incomplet, on le remplit avec des zéros). Il suffit alors de remplacer le groupe de 4 bits par le chiffre hexadécimal qui correspond.

Binary Coded Decimal

Le **Binary Coded Decimal**, abrégé BCD, est une représentation qui mixe binaire et décimal. Elle était utilisée sur les tout premiers ordinateurs pour faciliter le travail des programmeurs. Avec cette représentation, les nombres sont écrits en décimal, comme nous en avons l'habitude dans la vie courante. Simplement, chaque chiffre décimal est directement traduit en binaire : au lieu d'utiliser les symboles 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9, on codera chaque chiffre décimal sur quatre bits, en binaire normal. Par exemple, le nombre 624 donnera : 0110 0010 0100.

On peut remarquer que quelques combinaisons de quatre bits ne sont pas des chiffres valides : c'est le cas des suites de bits qui correspondent à 10, 11, 12, 13, 14 ou 15. Ces combinaisons peuvent être utilisées pour représenter d'autres symboles, comme un signe + ou - afin de gérer les entiers relatifs, ou une virgule pour gérer les nombres non-entiers. Mais sur d'autres ordinateurs, ces combinaisons servaient à représenter des chiffres en double : ceux-ci correspondaient alors à plusieurs combinaisons.

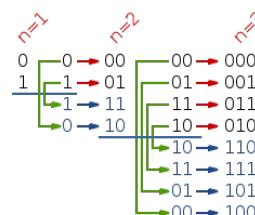
Code Gray

Avec le **code Gray**, deux nombres consécutifs n'ont qu'un seul bit de différence. Pour construire ce code Gray, on peut procéder en suivant plusieurs méthodes, les deux plus connues étant la méthode du miroir et la méthode de l'inversion.

La méthode du miroir est relativement simple. Pour connaître le code Gray des nombres codés sur n bits, il faut :

- partir du code Gray sur n-1 bits ;
- symétriser verticalement les nombres déjà obtenus (comme une réflexion dans un miroir) ;
- rajouter un 0 au début des anciens nombres, et un 1 au début des nouveaux nombres.

Il suffit de connaître le code Gray sur 1 bit pour appliquer la méthode : 0 est codé par le bit 0 et 1 par le bit 1.



Une autre méthode pour construire la suite des nombres en code Gray sur n bits est la méthode de l'inversion. Celle-ci permet de connaître le codage du nombre n à partir du codage du nombre n-1, comme la méthode du dessus. On part du nombre 0, systématiquement codé avec uniquement des zéros. Par la suite, on décide quel est le bit à inverser pour obtenir le nombre suivant, avec la règle suivante :

- si le nombre de 1 est pair, il faut inverser le dernier chiffre.
- si le nombre de 1 est impair, il faut localiser le 1 le plus à droite et inverser le chiffre situé à sa gauche.

Pour vous entraîner, essayez par vous-même avec 2, 3, voire 5.

Nombres entiers négatifs

Passons maintenant aux entiers négatifs en binaire : comment représenter le signe moins (« - ») avec des 0 et des 1 ? Eh bien, il existe plusieurs méthodes :

- la représentation en signe-valeur absolue ;
- la représentation en complément à un ;
- la représentation en complément à deux;
- la représentation par excès ;
- la représentation dans une base négative.

Représentation en signe-valeur absolue

La solution la plus simple pour représenter un entier négatif consiste à coder sa valeur absolue en binaire, et rajouter un bit qui précise si c'est un entier positif ou un entier négatif. Par convention, ce bit de signe vaut 0 si le nombre est positif et 1 s'il est négatif. Mais avec cette méthode, le zéro est codé deux fois : on a un -0, et un +0. Cela pose des problèmes lorsqu'on demande à notre ordinateur d'effectuer des calculs ou des comparaisons avec zéro.

Complément à deux

Pour éviter ces problèmes avec le zéro et les opérations arithmétiques, les ordinateurs actuels utilisent la méthode du **complément à deux**. Avec cette méthode, le zéro n'est codé que par un seul nombre binaire. L'idée derrière cette méthode est de coder un nombre entier négatif par un nombre positif qui se comportera de manière équivalente dans les calculs. Tout ce qu'il faut, c'est que les résultats des calculs effectués avec ce nombre positif et ceux réalisés avec le nombre négatif soient identiques.

Cette technique utilise le fait qu'un ordinateur manipule des entiers de taille fixe, dont le nombre de bits est toujours le même. Avec n bits, on peut coder 2^n valeurs différentes, dont 0, ce qui fait qu'on peut compter de 0 à $2^n - 1$. Si le résultat d'un calcul sort de cet intervalle, il ne peut pas être représenté par l'ordinateur : il se produit ce qu'on appelle un **débordement d'entier**. Face à une telle situation, certains ordinateurs utilisent ce qu'on appelle l'**arithmétique saturée** : ils arrondissent le résultat au plus grand entier supporté par l'ordinateur. La même chose est possible lors d'une soustraction, quand le résultat est inférieur à la plus petite valeur possible : l'ordinateur arrondit au plus petit entier possible.

Mais d'autres ordinateurs ne conservent que les bits de poids faible du résultat : les autres bits sont oubliés. On parle alors d'**arithmétique modulaire**. Par exemple, avec des nombres de 4 bits, l'addition $1111 + 0010$ ne donnera pas 17 (10001), mais 1 (0001). Avec cette arithmétique, il est possible que l'addition de deux nombres donne zéro. Prenons par exemple, toujours avec des nombres de 4 bits, l'addition de $13 + 3 = 16$. Maintenant, regardons ce que donne cette opération en binaire : $1101 + 0011 = 1\ 0000$. En ne gardant que les 4 bits de poids faible, on obtient : $1101 + 0011 = 0000$. En clair, avec ce genre d'arithmétique, $13 + 3 = 0$! On peut aussi reformuler en disant que $13 = -3$, ou encore que $3 = -13$. Et ne croyez pas que ça marche uniquement dans cet exemple : cela se généralise assez rapidement. Pire : ce qui marche pour l'addition marche aussi pour les autres opérations, telles la soustraction ou la multiplication. Un nombre négatif va donc être représenté par un entier positif strictement équivalent dans nos calculs qu'on appelle son complément à deux.

Pour traduire un nombre décimal en complément à deux, la procédure de conversion est différente suivant que le nombre est positif ou négatif. Pour les nombres positifs, il suffit de les traduire en binaire, sans faire quoi que ce soit de plus. C'est pour les nombres négatifs que la procédure est différente. Pour effectuer la conversion d'un nombre négatif, il faut convertir inverser tous les bits (les 0 deviennent des 1, et inversement) avant d'ajouter.

Pour savoir quelle est la valeur décimale d'un entier en complément à deux, on utilise la même procédure que pour traduire un nombre du binaire vers le décimal, à un détail près : le bit de poids fort (celui le plus à gauche) doit être multiplié par une puissance de deux, mais le résultat doit être soustrait au lieu d'être additionné. Au fait, on peut remarquer que le bit de poids fort (le bit le plus à gauche) vaut 1 si le nombre est négatif, et 0 si le nombre représenté est positif.

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Unsigned	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15															
2's Compl.	0 1 2 3 4 5 6 7 -8 -7 -6 -5 -4 -3 -2 -1															

Et ce bit de poids fort est écrasé en cas de débordement d'entier, qui sont possibles en complément à deux et à un. Par exemple, si l'on additionne les nombres positifs 0111 1111 et 0000 0001, le résultat sera 1000 0000, qui est négatif. Pour détecter les débordements, on doit vérifier si les signes sont mathématiquement cohérents : par exemple, si deux nombres de même signe sont ajoutés, un débordement a lieu quand le bit de signe du résultat a le signe opposé.

Dans nos ordinateurs, tous les nombres sont codés sur un nombre fixé et constant de bits. Ainsi, les circuits d'un ordinateur ne peuvent manipuler que des nombres de 4, 8, 12, 16, 32, 48, 64 bits, suivant la machine. Si l'on veut utiliser un entier codé sur 16 bits et que l'ordinateur ne peut manipuler que des nombres de 32 bits, il faut bien trouver un moyen de convertir notre nombre de 16 bits en un nombre de 32 bits, sans changer sa valeur et en conservant son signe. Cette conversion d'un entier en un entier plus grand, qui conserve valeur et signe s'appelle l'extension de signe. L'extension de signe des nombres positifs consiste à remplir les bits de poids fort avec des 0 jusqu'à arriver à la taille voulue : c'est la même chose qu'en décimal, où rajouter des zéros à gauche d'un nombre positif ne changera pas sa valeur. Pour les nombres négatifs, il faut remplir les bits à gauche du nombre à convertir avec des 1, jusqu'à obtenir le bon nombre de bits : par exemple, 1000 0000 (-128 codé sur 8 bits) donnera 1111 1111 1000 0000 après extension de signe sur 16 bits. L'extension de signe d'un nombre codé en complément à 2 se résume donc en une phrase : il faut recopier le bit de poids fort de notre nombre à convertir à gauche de celui-ci jusqu'à atteindre le nombre de bits voulu.

Représentation par excès

La représentation par excès consiste à ajouter un biais aux nombres à encoder afin de les encoder par un entier positif. Pour encoder tous les nombres compris entre $-X$ et Y en représentation par excès, il suffit de prendre la valeur du nombre à encoder, et de lui ajouter un biais égal à X . Ainsi, la valeur $-X$ sera encodée par zéro, et toutes les autres valeurs le seront par un entier positif. Par exemple, prenons des nombres compris entre -127 et 128. On va devoir ajouter un biais égal à 127, ce qui donne :

Valeur avant encodage	Valeur après encodage
-127	0
-126	1
-125	2
...	...
0	127
...	...
127	254
128	255

Représentation en base négative

Enfin, il existe une dernière méthode, assez simple à comprendre. Dans cette méthode, les nombres sont codés non en base 2, mais en base -2. Oui, vous avez bien lu : la base est un nombre négatif. Alors dans les faits, la base -2 est similaire à la base 2 : il y a toujours deux chiffres (0 et 1), et la position dans un chiffre indique toujours par quelle puissance il faut multiplier. Simplement, les puissances utilisées seront des puissances de -2 et non des puissances de 2.

Concrètement, les puissances de -2 sont les suivantes : 1, -2, 4, -8, 16, -32, 64, etc. En effet, un nombre négatif multiplié par un nombre négatif donne un nombre positif, ce qui fait que une puissance sur deux est négative, alors que les autres sont positives. Ainsi, on peut représenter des nombres négatifs, mais aussi des nombres positifs dans une puissance négative. Par exemple, la valeur du nombre noté 11011 en base -2 s'obtient comme suit :

...	-32	16	-8	4	-2	1
...	1	1	1	0	1	1

Sa valeur est ainsi de $(-32 \times 1) + (16 \times 1) + (-8 \times 1) + (4 \times 0) + (-2 \times 1) + (1 \times 1) = -32 + 16 - 8 - 2 + 1 = -25$.

Nombres à virgule

On sait donc comment sont stockés nos nombres entiers dans un ordinateur. Néanmoins, les nombres entiers ne sont pas les seuls nombres que l'on utilise au quotidien : il nous arrive d'en utiliser à virgule. Notre ordinateur n'est pas en reste : il est lui aussi capable de manipuler de tels nombres. Il existe deux méthodes pour coder des nombres à virgule en binaire :

- La virgule fixe ;
- La virgule flottante.

Virgule fixe

La méthode de la virgule fixe consiste à émuler nos nombres à virgule à partir de nombres entiers. Un nombre à virgule fixe est codé par un nombre entier proportionnel au nombre à virgule fixe. Pour obtenir la valeur de notre nombre à virgule fixe, il suffit de diviser l'entier servant à le représenter par le facteur de proportionnalité. Par exemple, pour coder 1,23 en virgule fixe, on peut choisir comme « facteur de conversion » 1000, ce qui donne l'entier 1230.

Généralement, les informaticiens utilisent une puissance de deux comme facteur de conversion, pour simplifier le calcul. Ainsi, comme pour les chiffres situés à gauche de la virgule, chaque bit situé à droite de la virgule doit être multiplié par une puissance de deux. La différence, c'est que les chiffres situés à droite de la virgule sont multipliés par une puissance négative de deux.

Nombres flottants

Les nombres à virgule fixe ont aujourd'hui été remplacés par les **nombres à virgule flottante**, où le nombre de chiffres après la virgule est variable. De nos jours, les ordinateurs utilisent un standard pour le codage des nombres à virgule flottante : la norme IEEE 754. Avec cette norme, l'écriture d'un nombre flottant est basée sur son écriture scientifique. Pour rappel, en décimal, l'écriture scientifique d'un nombre consiste à écrire celui-ci comme un produit entre un nombre et une puissance de 10, de la forme $a \times 10^e$ (Exposant). Le nombre a ne possède qu'un seul chiffre à gauche de la virgule : on peut toujours trouver un exposant tel que ce soit le cas. En clair, en base 10, sa valeur est comprise entre 1 (inclus) et 10 (exclu). En binaire, c'est la même chose, mais avec une puissance de deux. L'écriture scientifique binaire d'un nombre consiste à écrire celui-ci sous la forme $a \times 2^e$. Le nombre a ne possède toujours qu'un seul chiffre à gauche de la virgule, comme en base 10. Vu qu'en binaire, seuls deux chiffres sont possibles (0 et 1), le chiffre de a situé à gauche de la virgule est soit un zéro ou un 1. Pour stocker cette écriture scientifique, il faut donc stocker :

- la partie fractionnaire du nombre a , qu'on appelle la mantisse ;
- l'exposant de la puissance de deux ;
- un bit de signe.

Le bit à gauche de la virgule vaut 1, sauf dans quelques rares exceptions que nous aborderons plus tard. On verra que ce bit peut se déduire en fonction de l'exposant utilisé pour encoder le nombre à virgule, ce qui lui vaut le nom de **bit implicite**. L'exposant peut être aussi bien positif que négatif (pour permettre de coder des nombres très petits), et est encodé en représentation par excès sur n bits avec un biais égal à $2^{(n-1)} - 1$.

Les flottants doivent être stockés dans la mémoire d'une certaine façon, standardisée par la norme. Cette norme va (entre autres) définir quatre types de flottants différents, qui pourront stocker plus ou moins de valeurs différentes.

Format	Nombre de bits utilisés pour coder un flottant	Nombre de bits de l'exposant	Nombre de bits pour la mantisse	Décalage
Simple précision	32	8	23	127
Double précision	64	11	52	1023
Double précision étendue	80 ou plus	15 ou plus	64 ou plus	16383 ou plus

IEEE754 impose aussi le support de certains nombres flottants spéciaux qui servent notamment à stocker des valeurs comme l'infini. Commençons notre revue des flottants spéciaux par les **dénormaux**, aussi appelés flottants dénormalisés. Ces flottants ont une particularité : leur bit implicite vaut 0. Ces dénormaux sont des nombres flottants où l'exposant est le plus possible. Le zéro est un dénormal particulier. Au fait, remarquez que le zéro est codé deux fois à cause du bit de signe : on se retrouve avec un -0 et un +0.

Bit de signe	Exposant	Mantisse
0 ou 1	0	Mantisse différente de zéro (dénormal strict) ou égale à zéro (zéro)

Fait étrange, la norme IEEE754 permet de représenter **l'infini**, aussi bien en positif qu'en négatif. Celui-ci est codé en mettant l'exposant à sa valeur maximale et la mantisse à zéro. Et le pire, c'est qu'on peut effectuer des calculs sur ces flottants infinis. Mais cela a peu d'utilité.

Bit de signe	Exposant	Mantisse
0 ou 1	Valeur maximale	Mantisse égale à zéro

Mais malheureusement, l'invention des flottants infinis n'a pas réglé tous les problèmes. Par exemple, quel est le résultat de $\infty - \infty$? Et pour $\infty - \infty$? Ou encore 00 ? Autant prévenir tout de suite : mathématiquement, on ne peut pas savoir quel est le résultat de ces opérations. Pour pouvoir résoudre ces calculs, il a fallu inventer un nombre flottant qui signifie « je ne sais pas quel est le résultat de ton calcul pourri ». Ce nombre, c'est **NaN**. NaN est l'abréviation de Not A Number, ce qui signifie : n'est pas un nombre. Ce NaN a un exposant dont la valeur est maximale, mais une mantisse différente de zéro. Pour être plus précis, il existe différents types de NaN, qui diffèrent par la valeur de leur mantisse, ainsi que par les effets qu'ils peuvent avoir. Malgré son nom explicite, on peut faire des opérations avec NaN, mais cela ne sert pas vraiment à grand chose : une opération arithmétique appliquée avec un NaN aura un résultat toujours égal à NaN.

Bit de signe	Exposant	Mantisse
0 ou 1	Valeur maximale	Mantisse différente de zéro

La norme impose aussi une gestion des arrondis ou erreurs, qui arrivent lors de calculs particuliers. En voici la liste :

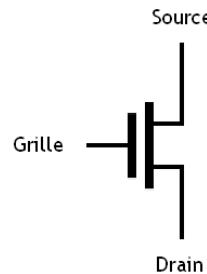
- Invalid operation : opération qui produit un NAN ;
- Overflow : résultat trop grand pour être stocké dans un flottant. Le plus souvent, on traite l'erreur en arrondissant le résultat vers Image non disponible ;
- Underflow : pareil, mais avec un résultat trop petit. Le plus souvent, on traite l'erreur en arrondissant le résultat vers 0 ;
- Division par zéro. La réponse la plus courante est de répondre + ou - l'infini ;
- Inexact : le résultat ne peut être représenté par un flottant et on doit l'arrondir. Pour éviter que des ordinateurs différents utilisent des méthodes d'arrondis différentes, on a décidé de normaliser les calculs sur les nombres flottants et les méthodes d'arrondis. Pour cela, la norme impose le support de quatre modes d'arrondis :
 - Arrondir vers + l'infini ;
 - vers - l'infini ;
 - vers zéro ;
 - vers le nombre flottant le plus proche.

Les circuits combinatoires

Grâce au chapitre précédent, on sait enfin comment sont représentées nos données les plus simples avec des bits. On n'est pas encore allés bien loin : on ne sait pas comment représenter des bits dans notre ordinateur ou les modifier, les manipuler, ni faire quoi que ce soit avec. On sait juste transformer nos données en paquets de bits (et encore, on ne sait vraiment le faire que pour des nombres entiers, des nombres à virgule et du texte...). C'est pas mal, mais il reste du chemin à parcourir ! Rassurez-vous, ce chapitre est là pour corriger ce petit défaut. On va vous expliquer comment représenter des bits dans un ordinateur et quels traitements élémentaires notre ordinateur va effectuer sur nos bits. Et on va voir que tout cela se fait avec de l'électricité ! :diable:

Transistors

Nos ordinateurs sont fabriqués avec des composants électroniques que l'on appelle des **transistors**, reliés entre eux pour former des circuits plus ou moins compliqués. Pour donner un exemple, sachez que les derniers modèles de processeurs peuvent utiliser près d'un milliard de transistors. Il existe différents types de transistors, et tous les transistors de nos ordinateurs sont des transistors dits CMOS. Ce sont des composants reliés à trois morceaux de « fil » conducteur que l'on appelle **broches**. On peut appliquer une tension électrique sur ces broches, qui peut représenter soit 0 soit 1. On a donné un nom à chaque broche pour mieux les repérer, nom qui est indiqué sur le schéma ci-dessous.

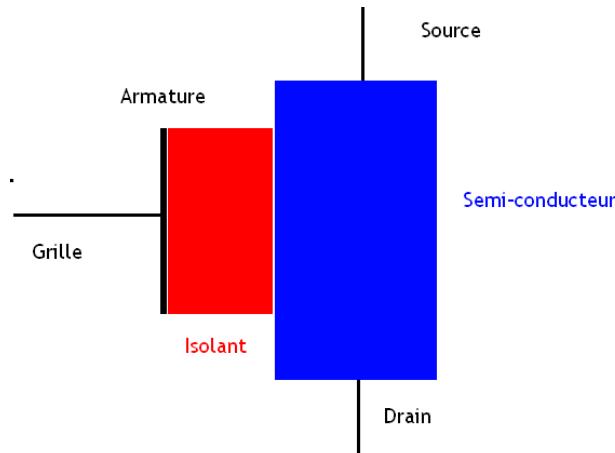


Dans les grandes lignes, un transistor est un interrupteur commandé par sa grille. Appliquez la tension adéquate et la liaison entre la source et le drain se comportera comme un interrupteur fermé. Mettez la grille à une autre valeur et cette liaison se comportera comme un interrupteur ouvert. Il existe deux types de transistors CMOS, qui diffèrent entre eux par le bit qu'il faut mettre sur la grille pour les ouvrir/fermer :

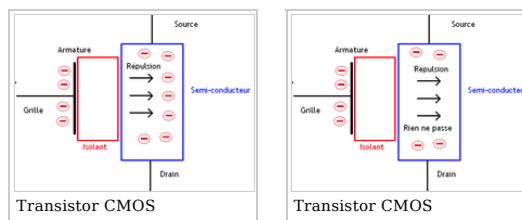
- les transistors NMOS qui s'ouvrent lorsqu'on envoie un zéro sur la grille et se ferment si la grille est à un ;
- et les PMOS se ferment lorsque la grille est à zéro, et s'ouvrent si la grille est à un.

Anatomie d'un transistor CMOS

À l'intérieur du transistor, on trouve simplement une plaque en métal reliée à la grille appelée l'armature, un bout de semi-conducteur entre la source et le drain, et un morceau d'isolant entre les deux.



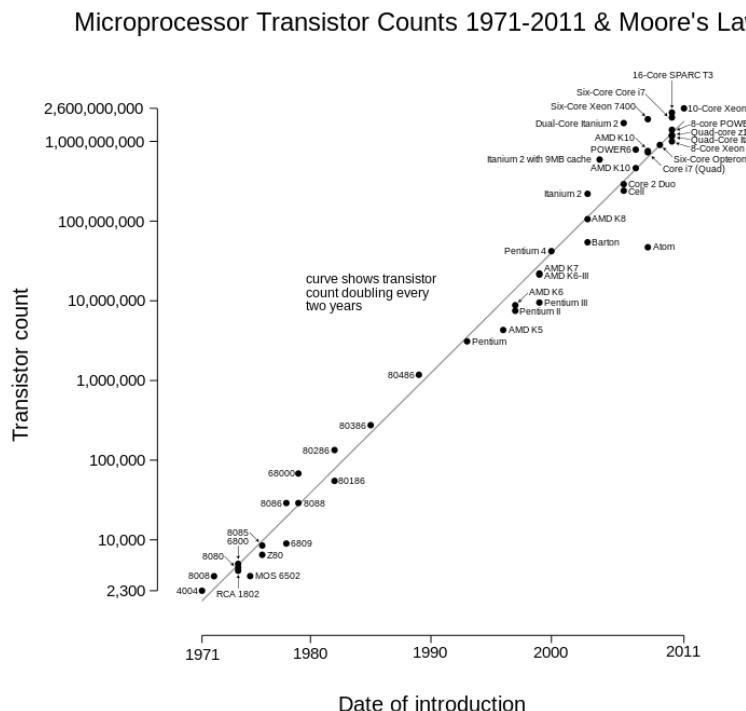
Si on laisse la grille tranquille, l'armature est vide d'électrons. Si on place une tension entre la source et le drain, un courant électrique va passer et des électrons vont circuler de la source vers le drain : un courant va s'établir. Maintenant, si on place une tension sur la grille, des électrons vont s'accumuler dans l'armature jusqu'à ce que la grille soit remplie. Ces électrons, chargés négativement, vont donc repousser les électrons qui circulent entre la source et le drain, gênant leur circulation : le courant va diminuer. À partir d'une certaine tension, les électrons stockés dans la grille vont tellement repousser les électrons de la source que ceux-ci ne traverseront plus du tout la liaison entre la source et le drain : le courant ne passe plus.



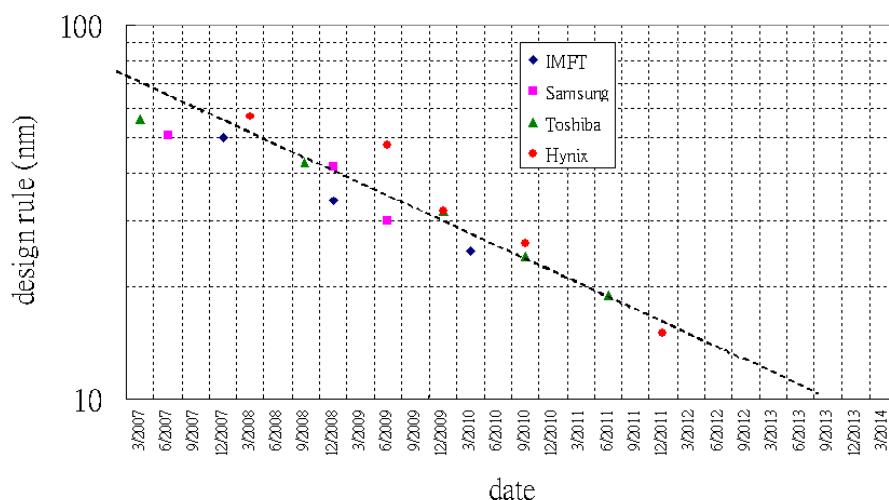
Loi de Moore

Les composants électroniques contiennent un grand nombre de transistors. Par exemple, les derniers modèles de processeurs peuvent utiliser près

d'un milliard de transistors. Cette orgie de transistors permet d'ajouter des fonctionnalités aux composants électroniques. C'est notamment ce qui permet aux processeurs récents d'intégrer plusieurs coeurs, une carte graphique, etc. En 1965, le cofondateur de la société Intel, spécialisée dans la conception de mémoires et de processeurs, a affirmé que la quantité de transistors présents dans un processeur doublait tous les 18 mois : c'est la **première loi de Moore**. En 1975, il réévalua cette affirmation : ce n'est pas tous les 18 mois que le nombre de transistors d'un processeur double, mais tous les 2 ans. Cette nouvelle version, appelée la **seconde loi de Moore**, est toujours valable de nos jours.



Cela n'aurait pas été possible sans le développement de la miniaturisation, qui permet de rendre les transistors plus petits. Il faut savoir que les circuits imprimés sont fabriqués à partir d'une plaque de silicium pur, un wafer, sur laquelle on vient graver le circuit imprimé. Les transistors sont donc répartis sur une surface plane. Ils ont souvent une largeur et une longueur qui sont très proches. Pour simplifier, la taille des transistors est aussi appelée la **finesse de gravure**. Celle-ci s'exprime le plus souvent en nanomètres. La loi de Moore nous donne des indications sur l'évolution de la finesse de gravure dans le temps. Doubler le nombre de transistors signifie qu'on peut mettre deux fois plus de transistors sur une même surface : la surface occupée par un transistor a été divisée par deux. Ainsi, la finesse de gravure est divisée par la racine carrée de deux, environ 1,4, tous les deux ans. Une autre formulation consiste à dire que la finesse de gravure est multipliée par 0,7 tous les deux ans, soit une diminution de 30 % tous les deux ans. Néanmoins, la loi de Moore n'est pas vraiment une loi gravée dans le marbre. Si celle-ci a été respectée jusqu'à présent, c'est avant tout grâce aux efforts des fabricants de processeurs, qui ont tenté de la respecter pour des raisons commerciales. Vendre des processeurs toujours plus puissants, avec de plus en plus de transistors est en effet gage de progression technologique autant que de nouvelles ventes.



Il arrivera un moment où les transistors ne pourront plus être miniaturisés, et ce moment approche ! Quand on songe qu'en 2016 certains transistors ont une taille proche d'une vingtaine ou d'une trentaine d'atomes, on se doute que la loi de Moore n'en a plus pour très longtemps. Et la progression de la miniaturisation commence déjà à montrer des signes de faiblesses. Le 23 mars 2016, Intel a annoncé que pour ses prochains processeurs, le doublement du nombre de transistors n'aurait plus lieu tous les deux ans, mais tous les deux ans et demi. Cet acte de décès de la loi de Moore n'a semble-t-il pas fait grand bruit, et les conséquences ne se sont pas encore faites sentir dans l'industrie. Au niveau technique, on peut facilement prédire que la course au nombre de coeurs a ses jours comptés.

On estime que la limite en terme de finesse de gravure sera proche des 5 à 7 nanomètres : à cette échelle, le comportement des électrons suit les

lois de la physique quantique et leur mouvement devient aléatoire, perturbant fortement le fonctionnement des transistors au point de les rendre inutilisables. Et cette limite est proche : des finesse de gravure de 10 nanomètres sont déjà disponibles chez certaines fondeurs comme TSMC. Autant dire que si la loi de Moore est respectée, la limite des 5 nanomètres sera atteinte dans quelques années, à peu-près vers l'année 2020. Ainsi, nous pourrons vivre la fin d'une ère technologique, et en voir les conséquences. Les conséquences économiques sur le secteur du matériel promettent d'être assez drastiques, que ce soit en terme de concurrence ou en terme de réduction de l'innovation.

Quant cette limite sera atteinte, l'industrie sera face à une impasse. Le nombre de coeurs ou la micro-architecture des processeurs ne pourra plus profiter d'une augmentation du nombre de transistors. Et les recherches en terme d'amélioration des micro-architectures de processeurs sont au point mort depuis quelques années. La majeure partie des optimisations matérielles récemment introduites dans les processeurs sont en effet connues depuis fort longtemps (par exemple, le premier processeur superscalaire à exécution dans le désordre date des années 1960), et ne sont améliorables qu'à la marge. Quelques équipes de recherche travaillent cependant sur des architectures capables de révolutionner l'informatique. Le calcul quantique ou les réseaux de neurones matériels sont une première piste, mais qui ne donneront certainement de résultats que dans des marchés de niche. Pas de quoi rendre un processeur de PC plus rapide.

Portes logiques

Ces transistors sont rassemblés dans ce qu'on appelle des **portes logiques**, c'est-à-dire des circuits qui possèdent des sorties et des entrées sur lesquelles on va placer ou récupérer des bits. Les entrées ne sont rien d'autre que des morceaux de « fil » conducteur sur lesquels on envoie un bit (une tension). À partir de là, le circuit électronique va réagir et déduire le bit à placer sur chaque sortie. Tous les composants d'un ordinateur sont fabriqués avec ce genre de circuits.

Sur les schémas qui vont suivre, les entrées des portes logiques seront à gauche et les sorties à droite !

La porte NON

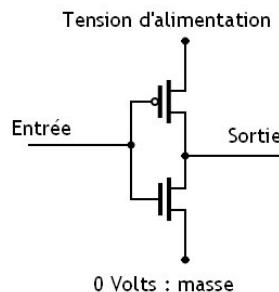
La première porte fondamentale est la porte NON, qui agit sur un seul bit : la sortie d'une porte NON est exactement le contraire de l'entrée.



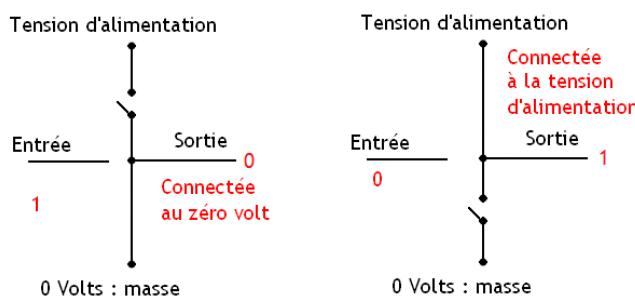
Pour simplifier la compréhension, je vais rassembler les états de sortie en fonction des entrées pour chaque porte logique dans un tableau que l'on appelle table de vérité.

Entrée	Sortie
0	1
1	0

Cette porte est fabriquée avec seulement deux transistors, comme indiqué ci-dessous.



Si on met un 1 en entrée de ce circuit, le transistor du haut va fonctionner comme un interrupteur ouvert, et celui du bas comme un interrupteur fermé : la sortie est reliée au zéro volt, et vaut donc 0. Inversement, si on met un 0 en entrée de ce petit montage électronique, le transistor du bas va fonctionner comme un interrupteur ouvert, et celui du haut comme un interrupteur fermé : la sortie est reliée à la tension d'alimentation, et vaut donc 1.



La porte ET

La porte ET possède plusieurs entrées, mais une seule sortie. Cette porte logique met sa sortie à 1 quand toutes ses entrées valent 1. Dans le cas le plus simple, une porte ET possède deux entrées.



Entrée 1	Entrée 2	Sortie
0	0	0
0	1	0
1	0	0
1	1	1

Certaines portes ET ont plus de deux entrées, et peuvent en avoir 3, 4, 5, 6, 7, etc. Là encore, leur sortie ne vaut 1 que si toutes les entrées valent 1 : dans le cas contraire, la sortie de la porte ET vaut 0. Dit autrement, si une seule entrée vaut 0, la sortie de la porte ET vaut 0.

Porte NAND

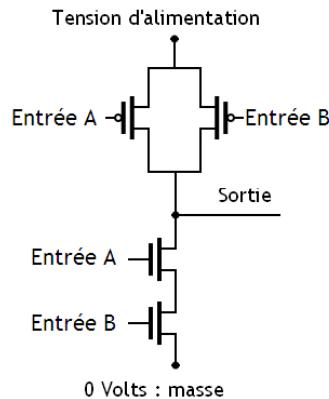
La porte NAND donne l'exact inverse de la sortie d'une porte ET. En clair, sa sortie ne vaut 1 que si au moins une entrée est nulle. Dans le cas contraire, si toutes les entrées sont à 1, la sortie vaut 0. Dans le cas le plus simple, une porte NAND à deux entrées. Certaines portes NAND ont plus de deux entrées : elles peuvent en avoir 3, 4, 5, 6, 7, etc. Là encore, leur sortie ne vaut 1 que si au moins une entrée est nulle : dans le cas contraire, la sortie de la porte ET vaut 0. Dit autrement, si toutes les entrées sont à 1, la sortie vaut 0.

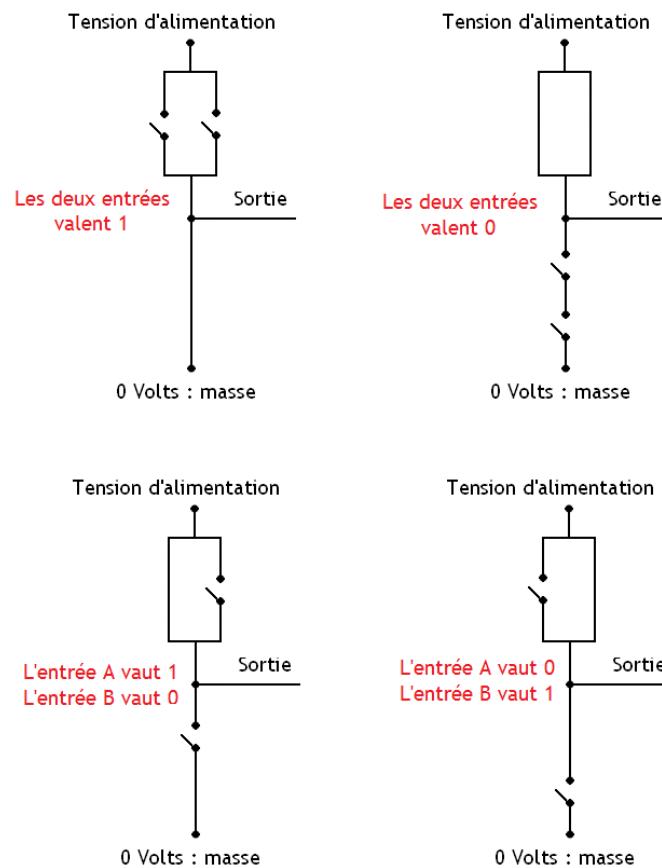
Entrée 1	Entrée 2	Sortie
0	0	1
0	1	1
1	0	1
1	1	0



Au fait, si vous regardez le schéma de la porte NAND, vous verrez que son symbole est presque identique à celui d'une porte ET : seul le petit rond sur la sortie de la porte a été rajouté. Il s'agit d'une sorte de raccourci pour schématiser une porte NON.

Voici en exclusivité comment créer une porte NAND à deux entrées avec des transistors CMOS !





La porte OU

La porte OU est une porte dont la sortie vaut 1 si et seulement si au moins une entrée vaut 1. Dit autrement, sa sortie est à 0 si toutes les entrées sont à 0. Dans le cas le plus simple, la porte OU possède deux entrées, ainsi qu'une seule sortie. Cette porte logique met sa sortie à 1 quand au moins une de ses entrées vaut 1. Certaines portes OU ont plus de deux entrées. Là encore, leur sortie est à 0 si et seulement si toutes les entrées sont à 0 : si une seule entrée est à 1, alors la sortie vaut 1.



Entrée 1	Entrée 2	Sortie
0	0	0
0	1	1
1	0	1
1	1	1

Porte NOR

La porte NOR donne l'exact inverse de la sortie d'une porte OU. Là encore, il en existe une version avec deux entrées, et des versions avec plus de deux entrées. Les tableaux et symboles qui suivent sont ceux d'une porte NOR à deux entrées.

Entrée 1	Entrée 2	Sortie
0	0	1
0	1	0
1	0	0
1	1	0



Implémenter une porte NOR à deux entrées avec des transistors CMOS ressemble à ce qu'on a fait pour la porte NAND.

[[File:Porte NOR fabriquée avec des transistors. 02.png|centre|Porte NOR fabriquée avec des transistors. [[File:Porte NOR fabriquée avec des transistors - Fonctionnement.png|centre|Porte NOR fabriquée avec des transistors.

Porte XOR

Avec une porte OU, deux ET et deux portes NON, on peut créer une porte nommée XOR. Cette porte est souvent appelée porte OU exclusif. Sa sortie est à 1 quand les deux bits placés sur ses entrées sont différents, et vaut 0 sinon.



Entrée 1	Entrée 2	Sortie
0	0	0
0	1	1
1	0	1
1	1	0

Porte NXOR

La porte XOR possède une petite sœur : la NXOR. Sa sortie est à 1 quand les deux entrées sont identiques, et vaut 0 sinon (elle est équivalente à une porte XOR suivie d'une porte NON).



Entrée 1	Entrée 2	Sortie
0	0	1
0	1	0
1	0	0
1	1	1

Les circuits combinatoires

Nous allons maintenant utiliser des portes logiques pour créer des circuits plus compliqués : les **circuits combinatoires**. Ces circuits font comme tous les autres circuits : ils prennent des données sur leurs entrées, et fournissent un résultat en sortie. Le truc, c'est que ce qui est fourni en sortie ne dépend que du résultat sur les entrées, et de rien d'autre (ce n'est pas le cas pour tous les circuits). Pour donner quelques exemples, on peut citer les circuits qui effectuent des additions, des multiplications, ou d'autres opérations arithmétiques du genre.

Nous allons donner quelques exemples de circuits assez fréquents dans un ordinateur et voir comment construire ceux-ci avec des portes logiques. Les circuits qui nous allons présenter sont utilisés dans les mémoires, ainsi que dans certains circuits de calcul. Il est important de bien mémoriser ces circuits, ainsi que la procédure pour les concevoir : nous en aurons besoin dans la suite du cours.

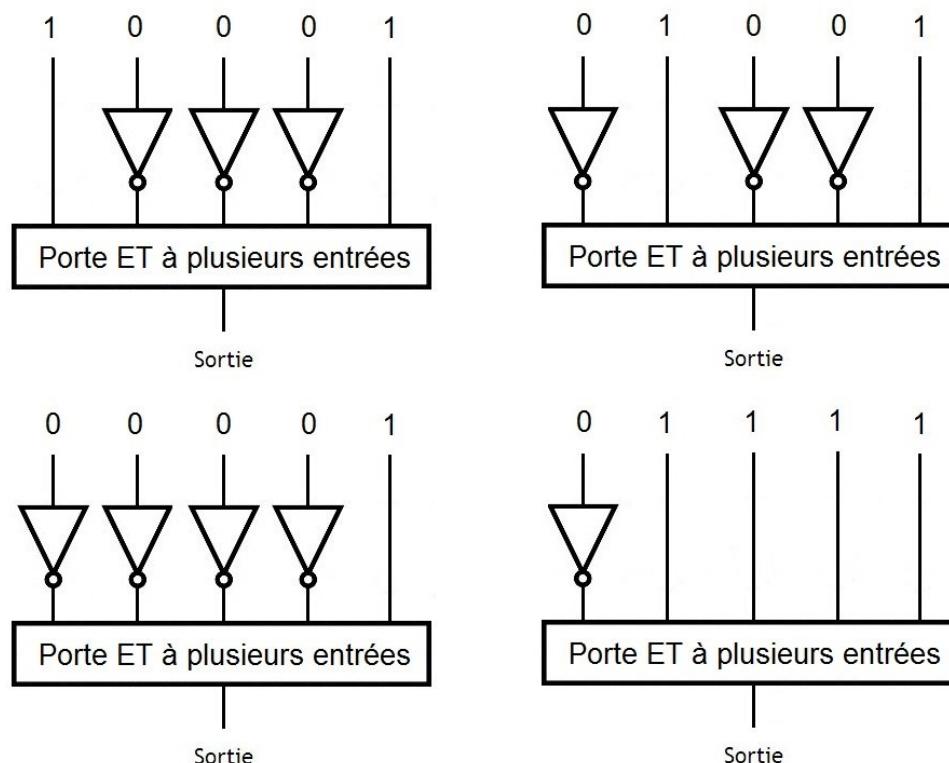
Comparateur

Pour commencer, nous allons voir le comparateur, un circuit qui possède plusieurs entrées et une seule sortie : on lui envoie un nombre codé sur plusieurs bits en entrée, et le circuit vérifie que ce nombre est égal à une certaine constante (2, 3, 5, 8, ou tout autre nombre) qui dépend du circuit :

- si le nombre en entrée est égal à cette constante, la sortie vaut 1 ;
- sinon, la sortie est mise à 0.

Ainsi, on peut créer un circuit qui mettra sa sortie à 1 uniquement si on envoie le nombre 5 sur ses entrées. Ou comme autre exemple, créer un circuit qui met sa sortie à 1 uniquement quand l'entrée correspondent au nombre 126. Et ainsi de suite : tout nombre peut servir de constante à vérifier.

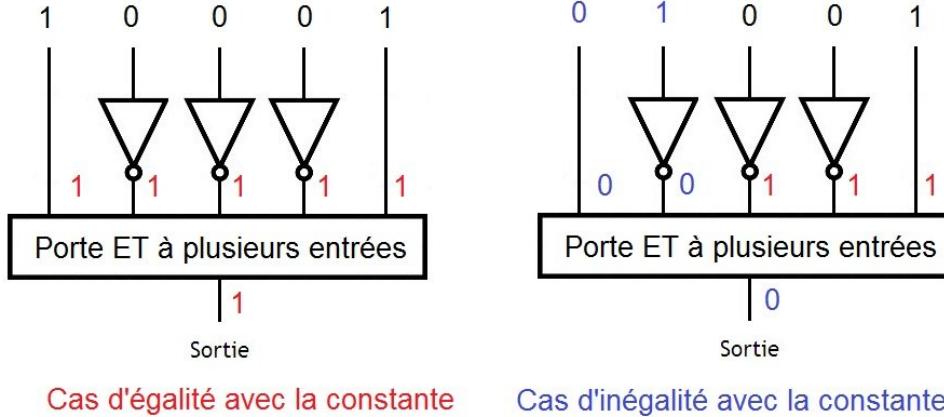
Tout circuit de ce type est systématiquement composé de deux couches de portes logiques : une couche de portes NON et une porte ET à plusieurs entrées. Créer un tel circuit se fait en trois étapes. En premier lieu, il faut convertir la constante à vérifier en binaire : dans ce qui suit, nous nommerons cette constante k. En second lieu, il faut créer la couche de portes NON. Pour cela, rien de plus simple : on place des portes NON pour les entrées de la constante k qui sont à 0, et on ne met rien pour les bits à 1. Par la suite, on place une porte ET à plusieurs entrées à la suite de la couche de portes NON.



Pour comprendre pourquoi on procède ainsi, il faut simplement regarder ce que l'on trouve en sortie de la couche de portes NON :

- si on envoie la constante, tous les bits à 0 seront inversés alors que les autres resteront à 1 : on se retrouve avec un nombre dont tous les bits sont à 1 ;
- si on envoie un autre nombre, soit certains 0 du nombre en entrée ne seront pas inversés, ou alors des bits à 1 le seront : il y aura au moins un bit à 0 en sortie de la couche de portes NON.

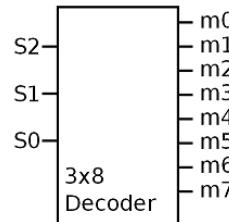
Ainsi, on sait que le nombre envoyé en entrée est égal à la constante k si et seulement si tous les bits sont à 1 en sortie de la couche de portes NON. Dit autrement, la sortie du circuit doit être à 1 si et seulement si tous les bits en sortie des portes NON sont à 1 : il ne reste plus qu'à trouver un circuit qui prenne ces bits en entrée et ne mette sa sortie à 1 que si tous les bits d'entrée sont à 1. Il existe une porte logique qui fonctionne ainsi : il s'agit de la porte ET à plusieurs entrées.



Décodeur

Nous allons poursuivre avec le décodeur, un composant :

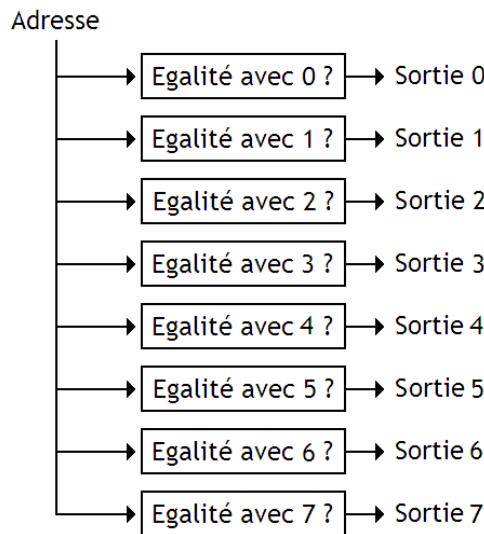
- qui contient une entrée sur laquelle on envoie un nombre n de bits ;
- qui contient 2^n sorties ;
- où les sorties sont numérotées en partant de zéro ;
- où on ne peut sélectionner qu'une seule sortie à la fois : une seule sortie devra être placée à 1, et toutes les autres à zéro ;
- et où deux nombres d'entrée différents devront sélectionner des sorties différentes : la sortie de notre contrôleur qui sera mise à 1 sera différente pour deux nombres différents placés sur son entrée.



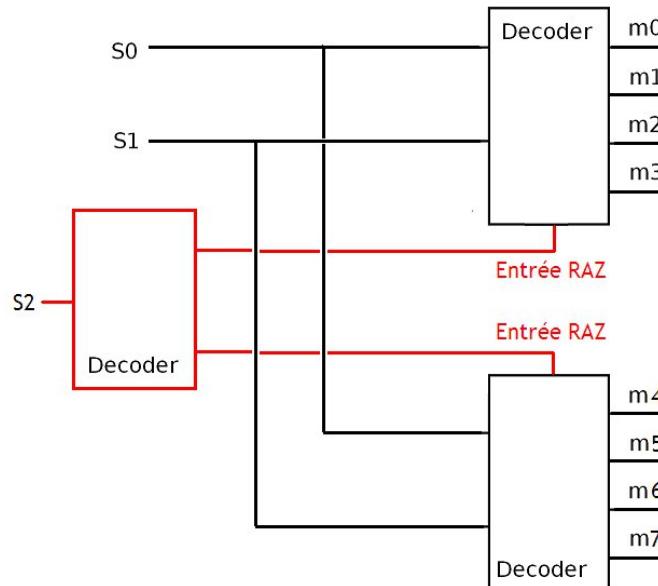
Le fonctionnement d'un décodeur est très simple :

- il prend sur son entrée un nombre entier x codé en binaire ;
- positionne à 1 la sortie numérotée x ;
- et positionne à zéro toutes les autres sorties.

En réfléchissant bien, on sait qu'on peut déduire la sortie assez facilement en fonction de l'entrée : si l'entrée du décodeur vaut N, la sortie mise à 1 sera la sortie N. Bref, déduire quand mettre à 1 la sortie N est facile : il suffit de comparer l'entrée avec N. Si l'adresse vaut N, on envoie un 1 sur la sortie, et on envoie un zéro sinon. Pour cela, j'ai donc besoin d'un comparateur pour chaque sortie, et le tour est joué.

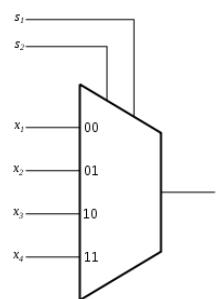


Comme autre méthode, on peut créer un décodeur en assemblant plusieurs décodeurs plus simples, nommés sous-décodeurs. Ces sous-décodeurs sont des décodeurs normaux, auxquels on a ajouté une entrée RAZ, qui permet de mettre à zéro toutes les sorties : si on met un 0 sur cette entrée, toutes les sorties passent à 0, alors que le décodeur fonctionne normalement sinon. Construire un décodeur demande suffisamment de sous-décodeurs pour combler toutes les sorties. Si on utilise des sous-décodeurs à n entrées, ceux-ci prendront en entrée les n bits de poids faible de l'entrée du décodeur que l'on souhaite construire (le décodeur final). Dans ces conditions, les n décodeurs auront une de leur sortie à 1. Pour que le décodeur final se comporte comme il faut, il faut désactiver tous les sous-décodeurs, sauf un avec l'entrée RAZ. Pour commander les n bits RAZ des sous-décodeurs, il suffit d'utiliser un décodeur qui est commandé par les bits de poids fort du décodeur final.



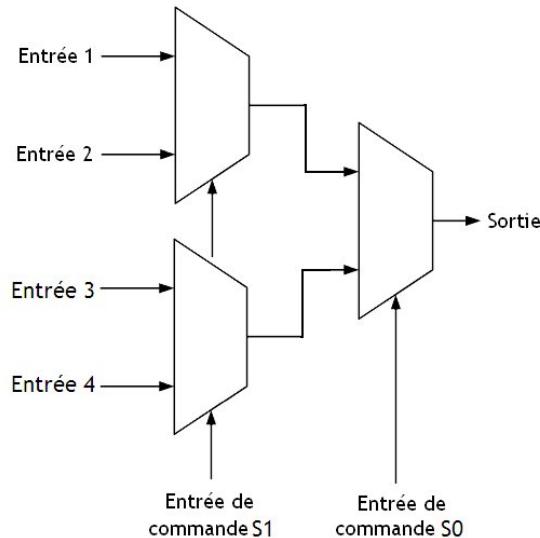
Multiplexeur

Les décodeurs ont des cousins : les multiplexeurs. Ces multiplexeurs sont des composants qui possèdent un nombre variable d'entrées et une sortie. Le rôle d'un multiplexeur est de recopier le contenu d'une des entrées sur sa sortie. Bien sûr, il faut bien choisir l'entrée qu'on veut recopier sur la sortie : pour cela, notre multiplexeur contient une entrée de commande qui permet de spécifier quelle entrée doit être recopiée.



On peut concevoir des multiplexeurs à plus de deux entrées en prenant deux multiplexeurs plus simples, et en ajoutant un multiplexeur 2 vers 2

sur leurs sorties respectives. Le multiplexeur final se contente de sélectionner une sortie parmi les deux sorties des multiplexeurs précédents, qui ont déjà effectué une sorte de présélection.

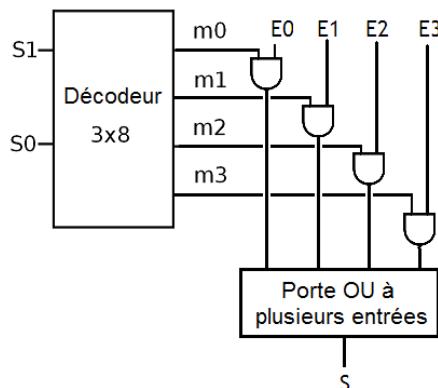


Il existe toutefois une manière bien plus simple pour créer des multiplexeurs : il suffit d'utiliser un décodeur, quelques portes OU, et quelques portes ET. L'idée est de :

- sélectionner l'entrée à recopier sur la sortie ;
- mettre les autres entrées à zéro ;
- faire un OU entre toutes les entrées : vu que toutes les entrées non-sélectionnées sont à zéro, la sortie de la porte OU aura la même valeur que l'entrée sélectionnée.

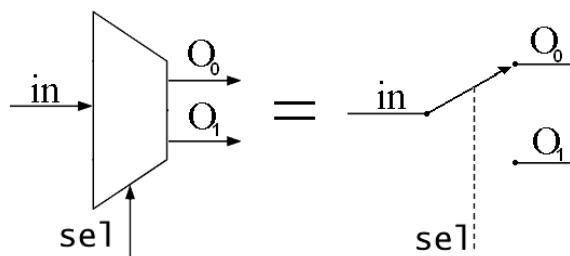
Pour sélectionner l'entrée adéquate du multiplexeur, on utilise un décodeur : si la sortie n du décodeur est à 1, alors l'entrée numéro n du multiplexeur sera recopiée sur sa sortie. Dans ces conditions, l'entrée de commande du multiplexeur correspond à l'entrée du décodeur. Pour mettre à zéro les entrées non-sélectionnées, on ajoute une porte ET par entrée : vu que $a.0=0$ et $a.1=a$, la porte ET :

- recopie l'entrée du multiplexeur si le décodeur sort un 1 ;
- met à zéro l'entrée si le décodeur sort un 0.

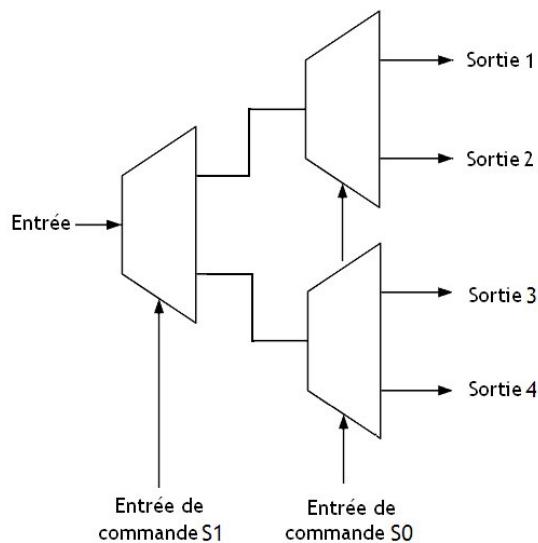


Démultiplexeur

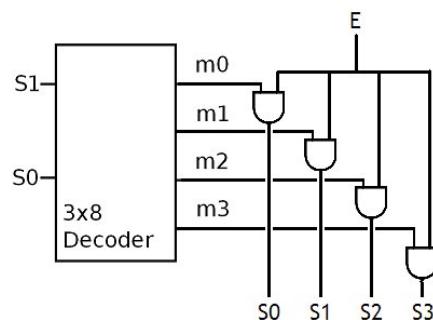
Après avoir vu le multiplexeur, il est temps de voir de démultiplexeur. Comme le nom l'indique, le démultiplexeur fait l'exact inverse du multiplexeur. Son rôle est de recopier, sur une des sorties, ce qu'il y a sur l'entrée. Évidemment, la sortie sur laquelle on recopie l'entrée est choisie parmi toutes les entrées possibles. Pour cela, le démultiplexeur possède une entrée de commande.



Ce qui a été fait pour les multiplexeurs peut aussi s'adapter aux démultiplexeurs : il est possible de créer des démultiplexeurs en assemblant des démultiplexeurs 1 vers 2. Évidemment, le même principe s'applique à des démultiplexeurs plus complexes : il suffit de rajouter des couches.

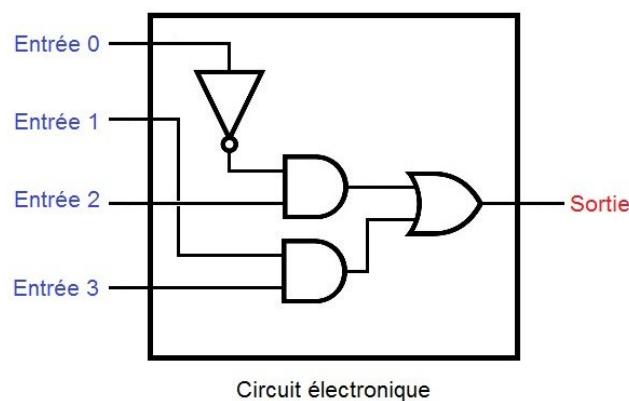


Un démultiplexeur peut aussi se fabriquer en utilisant un décodeur et quelques portes ET. Pour résumer, on utilise un décodeur pour sélectionner la sortie sur laquelle recopier l'entrée. L'entrée doit alors : soit être recopiée si la sortie est sélectionnée, soit mise à zéro. Pour cela, on utilise une porte ET entre la sortie de sélection du décodeur et l'entrée.



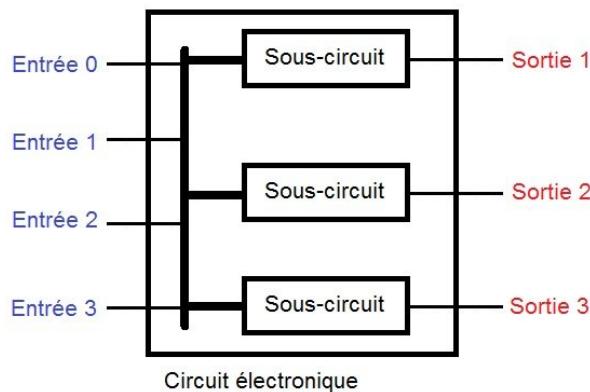
Créer ses propres circuits

On peut se demander s'il existe un moyen pour créer n'importe quel circuit. Eh bien c'est le cas : il existe des méthodes et procédures assez simples qui permettent à n'importe qui de créer n'importe quel circuit combinatoire. Nous allons voir comment créer des circuits combinatoires à plusieurs entrées, mais à une seule sortie. Pour simplifier, on peut considérer que les bits envoyés en entrée sont un nombre, et que le circuit calcule un bit à partir du nombre envoyé en entrée.



Circuit électrique

C'est à partir de circuits de ce genre que l'on peut créer des circuits à plusieurs sorties : il suffit d'assembler plusieurs circuits à une sortie. La méthode pour ce faire est très simple : chaque sortie est calculée indépendamment des autres, uniquement à partir des entrées. Ainsi, pour chaque sortie du circuit, on crée un circuit à plusieurs entrées et une sortie : ce circuit déduit quoi mettre sur cette sortie à partir des entrées. En assemblant ces circuits à plusieurs entrées et une sortie, on peut ainsi calculer toutes les sorties.



Tables de vérité

En premier lieu, il faut décrire ce que fait le circuit. Et pour cela, il suffit de lister la valeur de chaque sortie pour toute valeur possible en entrée : on obtient alors la table de vérité du circuit. Pour créer cette table de vérité, il faut commencer par lister toutes les valeurs possibles des entrées dans un tableau, et écrire à côté les valeurs des sorties qui correspondent à ces entrées. Cela peut être assez long : pour un circuit ayant n entrées, ce tableau aura 2^n lignes.

Le premier exemple sera très simple. Le circuit que l'on va créer sera un inverseur commandable, qui fonctionnera soit comme une porte NON, soit se contentera de recopier le bit fourni en entrée. Pour faire le choix du mode de fonctionnement (inverseur ou non), un bit de commande dira s'il faut que le circuit inverse ou non l'autre bit d'entrée :

- quand le bit de commande vaut zéro, l'autre bit est recopié sur la sortie ;
- quand il vaut 1, le bit de sortie est égal à l'inverse du bit d'entrée (pas le bit de commande, l'autre).

La table de vérité obtenue est celle d'une porte XOR :

Entrées	Sortie
00	0
01	1
10	1
11	0

Pour donner un autre exemple, on va prendre un circuit calculant le bit de parité d'un nombre. Ce bit de parité est un bit qu'on ajoute aux données à stocker afin de détecter des erreurs de transmission ou d'éventuelles corruptions de données. Le but d'un bit de parité est que le nombre de bits à 1 dans le nombre à stocker, bit de parité inclus, soit toujours un nombre pair. Ce bit de parité vaut : zéro si le nombre de bits à 1 dans le nombre à stocker (bit de parité exclu) est pair et 1 si ce nombre est impair. Détecter une erreur demande de compter le nombre de 1 dans le nombre stocké, bit de parité inclus : si ce nombre est impair, on sait qu'un nombre impair de bits a été modifié. Dans notre cas, on va créer un circuit qui calcule le bit de parité d'un nombre de 3 bits.

Entrées	Sortie
000	0
001	1
010	1
011	0
100	1
101	0
110	0
111	1

Pour ce dernier exemple, nous allons prendre en entrée un nombre de 3 bits. Le but du circuit à concevoir sera de déterminer le bit majoritaire dans ce nombre : celui-ci contient-il plus de 1 ou de 0 ? Par exemple :

- le nombre 010 contient deux 0 et un seul 1 : le bit majoritaire est 0 ;
- le nombre 011 contient deux 1 et un seul 0 : le bit majoritaire est 1 ;
- le nombre 000 contient trois 0 et aucun 1 : le bit majoritaire est 0 ;
- le nombre 110 contient deux 1 et un seul 0 : le bit majoritaire est 1 ;
- etc.

Entrées	Sortie
000	0
001	0
010	0
011	1
100	0
101	1
110	1
111	1

Lister les entrées de table qui valident l'entrée

Maintenant que l'on a la table de vérité, il faut lister les valeurs en entrée pour lesquelles la sortie vaut 1. Quand l'entrée du circuit (qui fait n bits, pour rappel) correspond à une de ces valeurs, le circuit sortira un 1. Ainsi, pour savoir si il faut mettre un 1 en sortie, il suffit de vérifier que l'entrée est égale à une de ces valeurs. Et on a vu précédemment un circuit qui peut faire cette comparaison : le comparateur vu plus haut peut

vérifier que ses entrées sont égales à une valeur.

Par exemple, pour un circuit dont la sortie est à 1 si son entrée vaut 0000, 0010, 0111 ou 1111, il suffit d'utiliser :

- un comparateur qui vérifie si l'entrée vaut 0000 ;
- un comparateur qui vérifie si l'entrée vaut 0010 ;
- un comparateur qui vérifie si l'entrée vaut 0111 ;
- et un comparateur qui vérifie si l'entrée vaut 1111.

Reste à combiner les sorties de ces comparateurs pour obtenir une seule sortie, ce qui est fait en utilisant un circuit relativement simple. On peut remarquer que la sortie du circuit est à 1 si un seul comparateur a sa sortie à 1. Or, on connaît un circuit qui fonctionne comme cela : la porte OU à plusieurs entrées. En clair, on peut créer tout circuit avec seulement des comparateurs et une porte OU à plusieurs entrées.

Pour l'exemple, nous allons reprendre le circuit de calcul d'inverseur commandable, vu plus haut.

Entrées	Sortie
00	0
01	1
10	1
11	0

Listons les lignes de la table où la sortie vaut 1.

Entrées	Sortie
01	1
10	1

Pour ce circuit, la sortie vaut 1 si et seulement si l'entrée du circuit vaut 01 ou 10. Dans ce cas, on doit créer deux comparateurs qui vérifient si leur entrée vaut respectivement 01 et 10. Une fois ces deux comparateurs créés, il faut ajouter la porte OU.

Établir l'équation du circuit (facultatif)

Les deux étapes précédentes sont les seules réellement nécessaires : quelqu'un qui sait créer un comparateur (ce qu'on a vu plus haut), devrait pouvoir s'en sortir. Néanmoins, il peut être utile d'écrire un circuit non sous forme d'un schéma, mais sous forme d'équations logiques. Mais attention : il ne s'agit pas des équations auxquelles vous êtes habitués. Ces équations logiques ne font que travailler avec des 1 et des 0, et n'effectuent pas d'opérations arithmétiques mais seulement des ET, des OU, et des NON. Ces équations peuvent ainsi être traduites en circuit, contrairement aux tables de vérité. Voici résumé dans ce tableau les différentes opérations, ainsi que leur notation. a et b sont des bits.

NON a	\bar{a}
a ET b	$a.b$
a OU b	$a+b$
a XOR b	$a \oplus b$

Avec ce petit tableau, vous savez comment écrire des équations logiques... Enfin presque, il ne faut pas oublier le plus important : les parenthèses, pour éviter quelques ambiguïtés. C'est un peu comme avec des équations normales : $(a \times b) + c$ donne un résultat différent de $a \times (b + c)$. Avec nos équations logiques, on peut trouver des situations similaires : par exemple, $(a.b) + c$ est différent de $a.(b+c)$.

Reste à savoir comment transformer une table de vérité en équations logiques, et enfin en circuit. Pour cela, il n'y a pas trente-six solutions : on va écrire une équation logique qui permettra de calculer la valeur (0 ou 1) d'une sortie en fonction de toutes les entrées du circuit. Et on fera cela pour toutes les sorties du circuit que l'on veut concevoir. Pour ce faire, on peut utiliser ce qu'on appelle la méthode des minterms, qui est strictement équivalente à la méthode vue au-dessus. Elle permet de créer un circuit en quelques étapes simples :

- lister les lignes de la table de vérité pour lesquelles la sortie vaut 1 (comme avant) ;
- écrire l'équation logique pour chacune de ces lignes (qui est celle d'un comparateur) ;
- faire un OU entre toutes ces équations logiques, en n'oubliant pas de les entourer par des parenthèses.

Pour écrire l'équation logique d'une ligne, il faut simplement :

- lister toutes les entrées de la ligne ;
- faire un NON sur chaque entrée à 0 ;
- et faire un ET avec le tout.

Vous remarquerez que la succession d'étapes précédente permet de créer un comparateur qui vérifie que l'entrée est égale à la valeur sur la ligne sélectionnée.

Pour illustrer le tout, on va reprendre notre exemple avec le bit de parité. La première étape consiste donc à lister les lignes de la table de vérité dont la sortie est à 1.

Entrées	Sortie
001	1
010	1
100	1
111	1

On a alors :

- la première ligne où l'entrée vaut 001 : son équation logique vaut $\bar{e}_2 \cdot \bar{e}_1 \cdot e_0$;
- la seconde ligne où l'entrée vaut 010 : son équation logique vaut $\bar{e}_2 \cdot e_1 \cdot \bar{e}_0$;
- la troisième ligne où l'entrée vaut 100 : son équation logique vaut $e_2 \cdot \bar{e}_1 \cdot \bar{e}_0$;
- la quatrième ligne où l'entrée vaut 111 : son équation logique vaut $e_2 \cdot e_1 \cdot e_0$.

On a alors obtenu nos équations logiques. Reste à faire un OU entre toutes ces équations, et le tour est joué !

Nous allons maintenant montrer un deuxième exemple, avec le circuit de calcul du bit majoritaire vu juste au-dessus. Première étape, lister les lignes de la table de vérité dont la sortie vaut 1 :

Entrées	Sortie
011	1
101	1
110	1
111	1

Seconde étape, écrire les équations de chaque ligne. Essayez par vous-même, avant de voir la solution ci-dessous.

- Pour la première ligne, l'équation obtenue est : $\bar{e}_2 \cdot e_1 \cdot e_0$.
- Pour la seconde ligne, l'équation obtenue est : $e_2 \cdot \bar{e}_1 \cdot e_0$.
- Pour la troisième ligne, l'équation obtenue est : $e_1 \cdot e_2 \cdot \bar{e}_0$.
- Pour la quatrième ligne, l'équation obtenue est : $e_2 \cdot e_1 \cdot \bar{e}_0$.

Il suffit ensuite de faire un OU entre les équations obtenues au-dessus.

Simplifier un circuit

On sait maintenant obtenir les équations logiques d'un circuit. Mais il serait sympathique de pouvoir simplifier ces équations, pour obtenir un circuit utilisant moins de portes logiques.

Algèbre de Boole

Pour simplifier notre équation, on peut utiliser certaines propriétés mathématiques simples pour factoriser ou développer comme on le ferait avec une équation mathématique normale. Ces propriétés forment ce qu'on appelle l'algèbre de Boole.

Commutativité	$a + b = b + a$ $a \cdot b = b \cdot a$ $a \oplus b = b \oplus a$
Associativité	$(a + b) + c = a + (b + c)$ $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
Distributivité	$(a + b) \cdot c = (c \cdot b) + (c \cdot a)$ $(a \cdot b) + c = (c + b) \cdot (c + a)$
Idempotence	$a \cdot a = a$ $a + a = a$
Élément nul	$a \cdot 0 = 0$ $a + 1 = 1$
Élément Neutre	$a \cdot 1 = a$ $a + 0 = a$ $a \oplus 0 = a$
Loi de De Morgan	$\bar{a} + \bar{b} = \bar{a \cdot b}$ $\bar{a \cdot b} = \bar{a} + \bar{b}$
Complémentarité	$\bar{\bar{a}} = a$ $a \oplus 1 = \bar{a}$ $a + \bar{a} = 1$ $a \oplus \bar{a} = 1$ $a \oplus a = 0$ $a \cdot \bar{a} = 0$

Comme premier exemple, nous allons travailler sur cette équation : $(\bar{e}_2 \cdot e_1 \cdot e_0) + (e_2 \cdot e_1 \cdot e_0)$. On peut la simplifier en trois étapes :

- Appliquer la règle de distributivité du ET sur le OU pour factoriser le facteur $e_1 \cdot e_0$, ce qui donne $(e_2 + \bar{e}_2) \cdot e_1 \cdot e_0$;
- Appliquer la règle de complémentarité sur le terme entre parenthèses $(e_2 + \bar{e}_2)$, ce qui donne $1 \cdot e_1 \cdot e_0$
- Et enfin, utiliser la règle de l'élément neutre du ET, qui nous dit que $a \cdot 1 = a$, ce qui donne : $e_1 \cdot e_0$.

En guise de second exemple, nous allons simplifier $(\bar{e}_2 \cdot e_1 \cdot e_0) + (e_2 \cdot \bar{e}_1 \cdot e_0)$. Cela se fait en suivant les étapes suivantes :

- Factoriser e_0 , ce qui donne : $(e_0 \cdot \bar{e}_2 \cdot e_1) + (e_2 \cdot \bar{e}_1 \cdot e_0)$;
- Utiliser la règle du XOR qui dit que $a \oplus b = b \oplus a$, ce qui donne $(e_2 \oplus e_1) \cdot e_0$.

Tableaux de Karnaugh

Il existe d'autres méthodes pour simplifier nos circuits. Les plus connues étant les tableaux de Karnaugh. Les tableaux de Karnaugh se basent sur un tableau (comme son nom l'indique). La simplification des équations avec un tableau de Karnaugh demande plusieurs étapes.

D'abord, il faut créer une table de vérité pour chaque bit de sortie du circuit à simplifier, qu'on utilise pour construire ce tableau. La première étape consiste à obtenir un tableau plus ou moins carré à partir d'une table de vérité, organisé en lignes et colonnes. Si on a n variables, on crée deux paquets avec le même nombre de variables (à une variable près pour un nombre impair de variables). Par exemple, supposons que j'aie quatre variables : a , b , c et d . Je peux créer deux paquets en regroupant les quatre variables comme ceci : ab et cd . Ou encore comme ceci : ac et bd . Il arrive que le nombre de variables soit impair : dans ce cas, il y a aura un paquet qui aura une variable de plus.

Ensuite, pour le premier paquet, on place les valeurs que peut prendre ce paquet sur la première ligne. Pour faire simple, considérez ce paquet de variables comme un nombre, et écrivez toutes les valeurs que peut prendre ce paquet en binaire. Rien de bien compliqué, mais ces variables doivent être encodées en code Gray : on ne doit changer qu'un seul bit en passant d'une ligne à sa voisine. Pour le second paquet, faites pareil, mais avec les colonnes. Là encore, les valeurs doivent être codées en code Gray.

AB	00	01	11	10
CD	10			
	11			
	01			
	00			

Tableau de Karnaugh à quatre variables.

Pour chaque ligne et chaque colonne, on prend les deux paquets : ces deux paquets sont avant tout des rassemblements de variables, dans lesquels chacune a une valeur bien précise. Ces deux paquets précisent ainsi les valeurs de toutes les entrées, et correspondent donc à une ligne dans la table de vérité. Sur cette ligne, on prend le bit de la sortie, et on le place à l'intersection de la ligne et de la colonne. On fait cela pour chaque case du tableau, et on le remplit totalement.

Troisième étape de l'algorithme : faire des regroupements. Par regroupement, on veut dire que les 1 dans le tableau doivent être regroupés en paquets de 1, 2, 4, 8, 16, 32, etc. Le nombre de 1 dans un paquet doit TOUJOURS être une puissance de deux. De plus, ces regroupements doivent obligatoirement former des rectangles dans le tableau de Karnaugh. De manière générale, il vaut mieux faire des paquets les plus gros possible, afin de simplifier l'équation au maximum.

A	A	\bar{A}	\bar{A}	
B	1	1	1	1
\bar{B}	0	0	1	1
C	\bar{C}	\bar{C}	\bar{C}	C

Exemple de regroupement valide.

A	A	\bar{A}	\bar{A}	
B	1	1	1	
\bar{B}			1	
C	\bar{C}	\bar{C}	\bar{C}	C

Exemples de regroupement invalide.

Il faut noter que les regroupements peuvent se recouvrir. Non seulement c'est possible, mais c'est même conseillé : cela permet d'obtenir des regroupements plus gros. De plus, ces regroupements peuvent passer au travers des bords du tableau : il suffit de les faire revenir de l'autre côté. Et c'est possible aussi bien pour les bords horizontaux (gauche et droite) que pour les bords verticaux (haut et bas). Le même principe peut s'appliquer aux coins.

AB	00	01	11	10
CD	10	1	1	1
	11	1	1	1
	01	1	1	1
	00	1	1	1

Regroupements par les bords du tableau de Karnaugh, avec recouvrement.

Trouver l'équation qui correspond à un regroupement est un processus en plusieurs étapes, que nous illustrerons dans ce qui va suivre. Ce processus demande de :

- trouver la variable qui ne varie pas dans les lignes et colonnes attribuées au regroupement ;
- inverser la variable si celle-ci vaut toujours zéro dans le regroupement ;
- faire un ET entre les variables qui ne varient pas.
- faire un OU entre les équations de chaque regroupement, et on obtient l'équation finale de la sortie.

Les circuits séquentiels

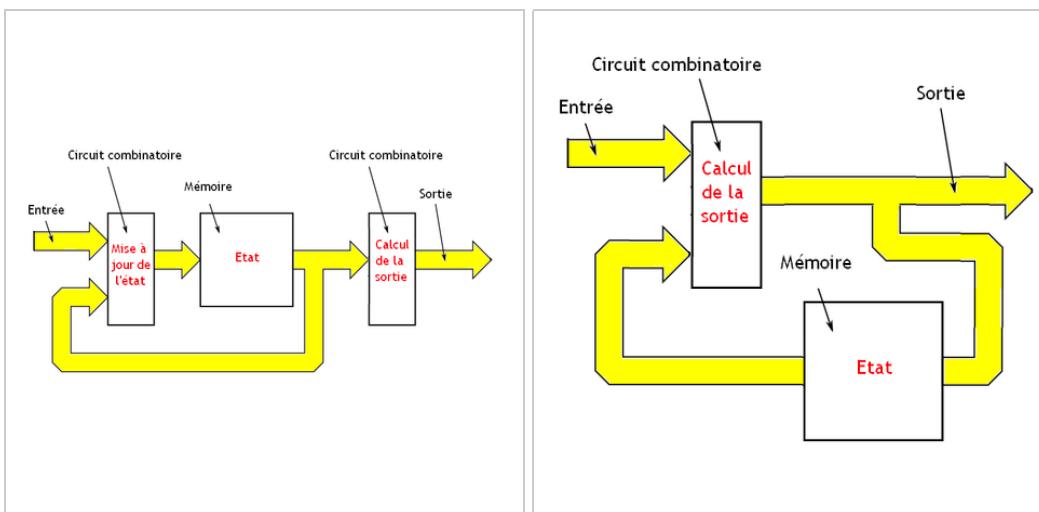
Avec les circuits combinatoires, on sait traiter et manipuler de l'information. Il nous manque encore une chose : la mémoriser. La valeur de la sortie de ces circuits ne dépend que de l'entrée et pas de ce qui s'est passé auparavant : les circuits combinatoires n'ont pas de mémoire. Pour répondre à ce besoin, les électroniciens ont inventé des **circuits séquentiels** qui possèdent une capacité de mémorisation. L'ensemble des informations mémorisées dans un circuit séquentiel forme ce qu'on appelle l'état du circuit. Pour mémoriser cet état, un circuit doit posséder des composants pour stocker un ou plusieurs bits : ce sont des mémoires.

Les **mémoires** d'un circuit séquentiel permettent de mémoriser des informations, que l'on peut récupérer plus tard. Ainsi, on peut parfaitement consulter une mémoire pour récupérer tout ou partie de son contenu : cette opération est ce qu'on appelle une opération de lecture. Mais évidemment, on peut aussi ranger des informations dans la mémoire ou les modifier : on effectue alors une opération d'écriture. Généralement, les informations lues sont disponibles sur la sortie de la mémoire, tandis que l'information à écrire dans une mémoire est envoyée sur son entrée (le circuit effectuant l'écriture sous certaines conditions qu'on verra plus tard).

De manière générale, il se peut que l'état mémorisé dans la mémoire ne dépende pas que de la donnée envoyée sur l'entrée, mais aussi de l'état antérieur de la mémoire. Pour cela, le circuit contient un circuit combinatoire qui calcule le nouvel état en fonction de l'ancien état et des valeurs des entrées. Un circuit séquentiel peut ainsi être découpé en deux morceaux : des mémoires qui stockent l'état du circuit, et des circuits combinatoires pour mettre à jour l'état du circuit et sa sortie. Cette mise à jour de l'état du circuit dépend de l'entrée mais aussi de l'ancien état. Suivant la méthode utilisée pour déterminer la sortie en fonction de l'état, on peut classer les circuits séquentiels en deux catégories :

- les **automates de Moore**, où la sortie ne dépend que de l'état mémorisé ;
- et les **automates de Mealy**, où la sortie dépend de l'état du circuit et de ses entrées.

Ces derniers ont tendance à utiliser moins de portes logiques que les automates de Moore.



Automate de Moore.

Automate de Mealy.

Bascules : des mémoires de 1 bit

On vient de voir que la logique séquentielle se base sur des circuits combinatoires auxquels on ajoute des mémoires. Pour le moment, on sait créer des circuits combinatoires, mais on ne sait pas faire des mémoires. Pourtant, on a déjà tout ce qu'il faut : avec nos portes logiques, on peut créer des circuits capables de mémoriser un bit. Ces circuits sont ce qu'on appelle des **bascules**, ou flip-flop. Une solution pour créer une bascule consiste à boucler la sortie d'un circuit sur son entrée, de façon à ce que la sortie rafraîchisse le contenu de l'entrée en permanence. Un circuit séquentiel contient toujours au moins une entrée reliée sur une sortie, contrairement aux circuits combinatoires, qui ne contiennent jamais la moindre boucle ! On peut grossièrement classer les bascules en quelques grands types principaux : les bascules RS, les bascules JK et les bascules D. Nous ne parlerons pas des bascules JK dans ce qui va suivre.

Bascules RS

Les bascules RS possèdent :

- une sortie pour récupérer le bit mémorisé; avec éventuellement une autre sortie qui fournit l'inverse de ce bit ;
- deux entrées qui permettent de le modifier : une entrée permet de le mettre à 0, tandis que l'autre le met à 1.

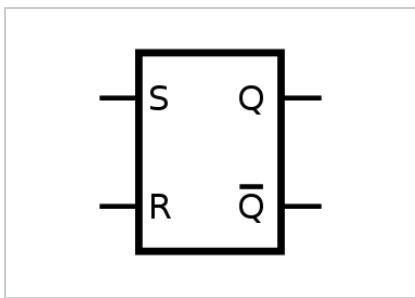
On classe ces bascules RS suivant ce qu'il faut mettre sur les entrées pour modifier le bit mémorisé, ce qui permet de distinguer les bascules RS à NOR des bascules RS à NAND.

Les **bascules RS à NOR** comportent deux entrées R et S et une sortie Q, sur laquelle on peut lire le bit stocké. Le principe de ces bascules est assez simple :

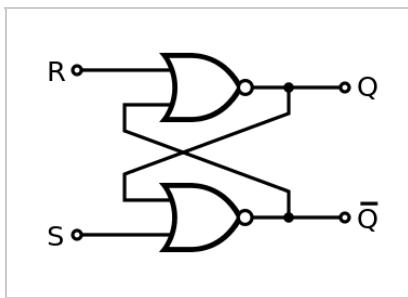
- si on met un 1 sur l'entrée R et un 0 sur l'entrée S, la bascule mémorise un zéro ;
- si on met un 0 sur l'entrée R et un 1 sur l'entrée S, la bascule mémorise un un ;
- si on met un zéro sur les deux entrées, la sortie Q sera égale à la valeur mémorisée juste avant.

Pour vous rappeler de ceci, sachez que les entrées de la bascule ne sont nommées ainsi par hasard : R signifie Reset (qui signifie mise à zéro en anglais) et S signifie Set (qui veut dire mise à un en anglais). Petite remarque : si on met un 1 sur les deux entrées, on ne sait pas ce qui arrivera sur ses sorties. Après tout, quelle idée de mettre la bascule à 1 en même temps qu'on la met à zéro !

Entrée Reset	Entrée Set	Sortie Q
0	0	Bit mémorisé par la bascule
0	1	1
1	0	0
1	1	Interdit



Interface d'une bascule RS à NOR.

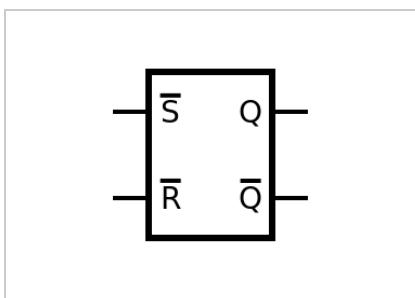


Circuit d'une bascule RS à NOR.

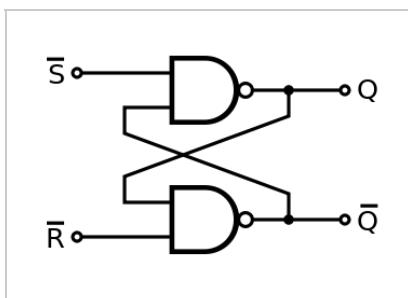
Les **bascules RS à NAND** utilisent des portes NAND pour créer une bascule. Cela n'a pas d'avantages, mais c'est une possibilité comme une autre. Ces bascules fonctionnent différemment de la bascule précédente :

- si on met un 0 sur l'entrée R et un 1 sur l'entrée S, la bascule mémorise un 0 ;
- si on met un 1 sur l'entrée R et un 0 sur l'entrée S, la bascule mémorise un 1 ;
- si on met un 1 sur les deux entrées, la sortie Q sera égale à la valeur mémorisée juste avant.

Entrée Reset	Entrée Set	Sortie Q
0	0	Interdit
0	1	1
1	0	0
1	1	Bit mémorisé par la bascule

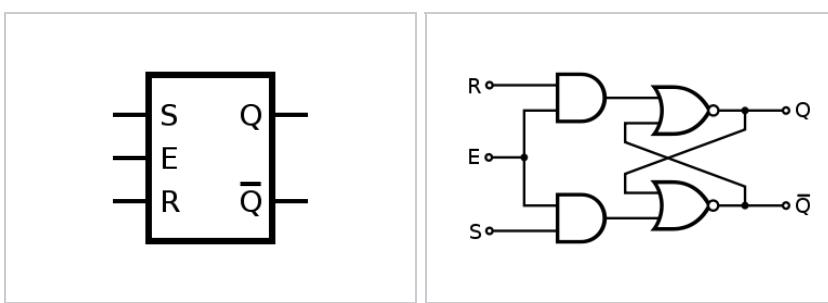


Interface d'une bascule RS à NAND.



SR Flip-flop Diagram.

Dans la bascule RS à NAND du dessus, le bit mémorisé change dès que l'on envoie un bit à 1 sur une des deux entrées R et S. On verra plus tard qu'il peut être utile d'autoriser ou d'interdire cette modification dans certains cas. Dans ces conditions, on peut faire en sorte de créer une bascule où l'on pourrait « activer » ou « éteindre » les entrées R et S à volonté. Cette bascule mémoriserait un bit et aurait toujours des entrées R et S. Mais ces entrées ne fonctionneront que si l'on autorise la bascule à prendre en compte ses entrées. Pour cela, il suffit de rajouter une entrée E à notre circuit. Suivant la valeur de cette entrée, l'écriture dans la bascule sera autorisée ou interdite. Si l'entrée E vaut zéro, alors tout ce qui se passe sur les entrées R et S ne fera rien : la bascule conservera le bit mémorisé, sans le changer. Par contre, si l'entrée E vaut 1, alors les entrées R et S feront ce qu'il faut et la bascule fonctionnera comme une bascule RS normale. On peut aussi faire la même chose, mais avec une bascule RS à NOR, mais le circuit n'est alors pas tout à fait le même. Dans tous les cas, on obtient alors une **bascule RS à entrée Enable**. Pour créer un tel circuit, rien de plus simple : nous allons ajouter un circuit avant les entrées R et S, qui inaktivera celles-ci si l'entrée E vaut zéro. La table de vérité de ce circuit est identique à celle d'une simple porte ET. Le circuit obtenu est donc celui-ci :

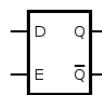


Bascule RS à entrée Enable.

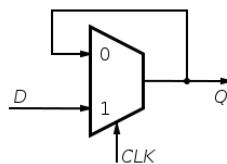
Circuit d'une bascule RS à entrée Enable.

Bascule D

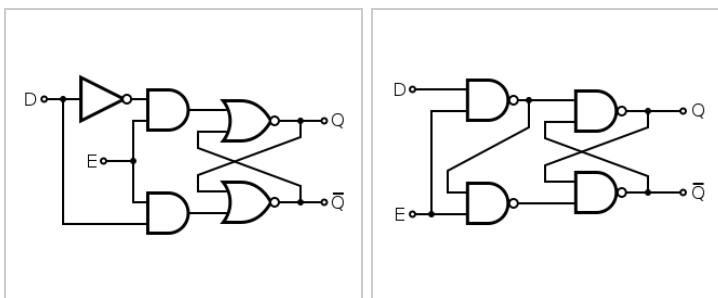
Les bascules D sont différentes des bascules RS, même si elles ont deux entrées. La différence tient dans ce que l'on doit mettre sur les entrées pour mémoriser un bit. Le bit à mémoriser est envoyé directement sur une des entrées, notée D : la bascule a directement connaissance du bit à mémoriser. L'autre entrée, l'entrée Enable, permet d'indiquer quand la bascule doit mettre son contenu à jour : elle permet d'autoriser ou d'interdire les écritures dans la bascule. Ainsi, tant que cette entrée Enable reste à 0, le bit mémorisé par la bascule reste le même, peu importe ce qu'on met sur l'entrée D : il faut que l'entrée Enable passe à 1 pour que l'entrée soit recopiée dans la bascule et mémorisée.



On peut créer une bascule D avec un simple multiplexeur. L'idée est très simple. Quand l'entrée Enable est à 0, la sortie du circuit est bouclée sur l'entrée : le bit mémorisé, qui était présent sur la sortie, est alors renvoyé en entrée, formant une boucle. Cette boucle reproduit en permanence le bit mémorisé. Par contre, quand l'entrée Enable vaut 1, la sortie du multiplexeur est reliée à l'entrée D. Ainsi, ce bit est alors renvoyé sur l'autre entrée : les deux entrées du multiplexeur valent le bit envoyé en entrée, mémorisant le bit dans la bascule.



On peut aussi construire une bascule D à partir d'une simple bascule RS à entrée Enable : il suffit d'ajouter un circuit qui déduise quoi mettre sur les entrées R et S suivant la valeur sur D. On peut alors remarquer que l'entrée R est toujours égale à l'inverse de D, alors que S est toujours strictement égale à D. On obtient alors le circuit suivant. On peut aussi faire la même chose, mais avec la bascule RS à NAND.



Bascule D à NOR.

Bascule D à NAND.

Registres : des mémoires de plusieurs bits

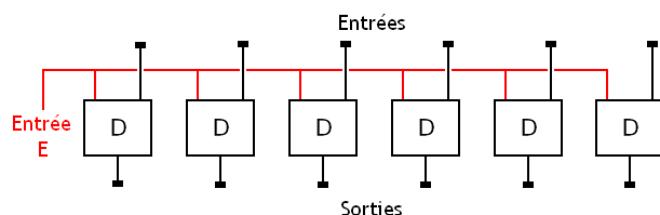
On vient de voir comment créer des bascules, des circuits capables de mémoriser un seul bit. Il se trouve que l'on peut assembler des bascules pour créer des circuits capables de mémoriser plusieurs bits : ces circuits sont appelés des registres.

Registres simples

Les registres les plus simples sont capables de mémoriser un nombre, codé sur une quantité fixe de bits. On peut à tout moment récupérer le nombre mémorisé dans le registre : on dit alors qu'on effectue une lecture. On peut aussi mettre à jour le nombre mémorisé dans le registre, le remplacer par un autre : on dit qu'on effectue une écriture. Ainsi, les registres possèdent :

- des sorties de lecture, sur lesquelles on peut récupérer/lire le nombre mémorisé ;
- des entrées d'écriture, sur lesquelles on envoie le nombre à mémoriser (celui qui remplacera le contenu du registre) ;
- et une entrée Enable, qui a le même rôle que pour une bascule.

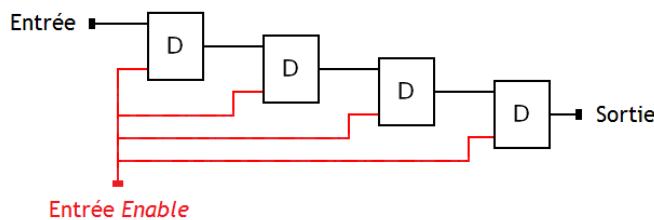
Si l'entrée Enable est à 0, le registre n'est pas mis à jour : on peut mettre n'importe quelle valeur sur les entrées, le registre n'en tiendra pas compte et ne remplacera pas son contenu par ce qu'il y a sur l'entrée. La mise à jour, l'écriture dans un registre n'a lieu que si l'entrée Enable est mise à 1. Pour résumer, l'entrée Enable sert donc à indiquer au registre si son contenu doit être mis à jour quand une écriture a lieu. Ainsi, un registre est composé de plusieurs bascules qui sont toutes mises à jour en même temps : pour cela, toutes les entrées Enable sont reliées au même signal, à la même entrée de commande.



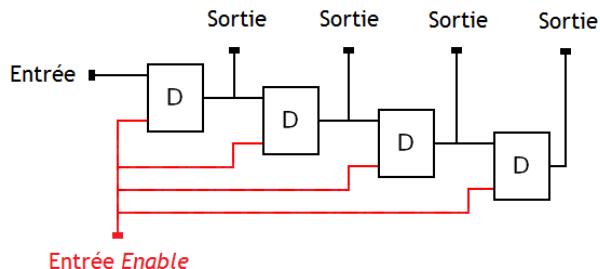
Registres à décalage

Certains registres sont toutefois plus complexes. On peut notamment citer les registres à décalage, des registres dont le contenu est décalé d'un cran vers la gauche ou la droite sur commande.

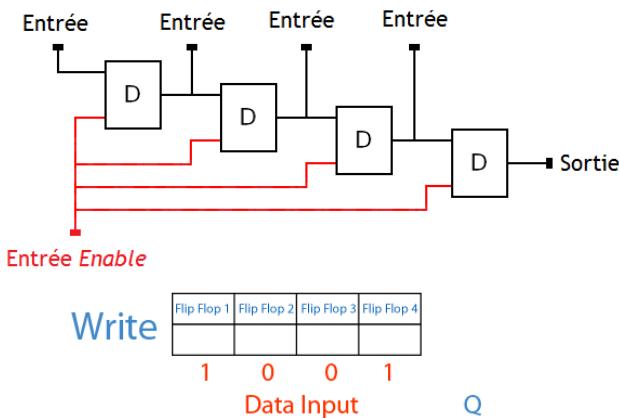
Avec les registres à **entrée et sortie série**, on peut mettre à jour un bit à la fois, de même qu'on ne peut en récupérer qu'un à la fois. Ces registres servent essentiellement à mettre en attente des bits tout en gardant leur ordre : un bit envoyé en entrée ressortira sur la sortie après plusieurs commandes de mise à jour sur l'entrée Enable.



Les registres à décalage à **entrée série et sortie parallèle** sont similaires aux précédents : on peut ajouter un nouveau bit en commandant l'entrée Enable et les anciens bits sont alors décalés d'un cran. Par contre, on peut récupérer (lire) tous les bits en une seule fois. Ils permettent notamment de reconstituer un nombre qui est envoyé bit par bit sur un fil (un bus série).



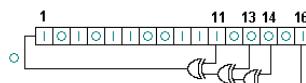
Enfin, il reste les registres à **entrée parallèle et sortie série**. Ces registres sont utiles quand on veut transmettre un nombre sur un fil : on peut ainsi envoyer les bits un par un. On initialise les bascules, avant de déconnecter les entrées : les bits se propageront alors de bascule en bascule vers la sortie à chaque front ou signal sur l'entrée Enable.



Registres à décalage à rétroaction linéaire

Les **registres à décalage à rétroaction linéaire** sont des registres à décalage un peu bidouillés. Avec eux, le bit qui rentre dans le nombre n'est pas fourni sur une entrée, mais est calculé en fonction du contenu du registre par un circuit combinatoire. La fonction qui permet de calculer le bit en sortie est assez spéciale. Dans le cas le plus simple, on dit qu'elle est linéaire, ce qui veut dire que le bit de sortie se calcule à partir en multipliant les bits d'entrée par 0 ou 1, et en additionnant le tout. En clair, ce bit de sortie se calcule par une formule du style :

$0*a_3 + 1*a_2 + 1*a_1 + 0*a_0$ (on ne garde que le bit de poids faible du résultat). Penchons-nous un peu sur cette addition qui ne garde que le bit de poids faible : je ne sais pas si vous avez remarqué, mais il s'agit ni plus ni moins que d'un calcul de parité paire. En effet, si on additionne N bits, le bit de poids faible vaut zéro pour un nombre pair, et 1 pour un nombre impair. Le circuit combinatoire chargé de calculer le bit de résultat est donc un circuit qui calcule la parité de la somme des bits choisis. Pour cela, il suffit d'effectuer une série de XOR entre tous les bits à additionner.



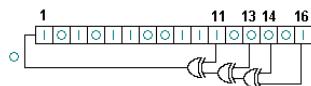
Il existe une variante de ce genre de registre, qui modifie légèrement son fonctionnement. Il s'agit des **registres à décalages à rétroaction affine**. Avec ces registres, la fonction qui calcule le bit de résultat n'est pas linéaire, mais affine. En clair, ce bit de sortie se calcule par une formule du style : $0*a_3 + 1*a_2 + 1*a_1 + 0*a_0 + 1$. Notez le + 1 à la fin de la formule : c'est la seule différence. Avec ce genre de registre, le bit de résultat est donc calculé en faisant le calcul d'un bit d'imparité de certains (ou de la totalité) des bits du registre. Un tel circuit est donc composé de portes NXOR, comparé à son comparase linéaire, composé à partir de portes XOR. Petite remarque : si je prends un registre à rétroaction linéaire et un registre à rétroaction affine avec les mêmes coefficients sur les mêmes bits, le résultat du premier sera égal à l'inverse de l'autre.

Les registres à décalage à rétroaction précédents sont appelés des registres à **rétroaction linéaire de Fibonacci**. Il existe un deuxième type de registres à décalage à rétroaction : les **registres à décalage à rétroaction de Gallois**. Ceux-ci sont un peu l'inverse des registres à décalages à rétroaction de Fibonacci. Dans ces derniers, on prenait plusieurs bits du registre à décalage pour en déduire un seul bit. Avec les registres à décalage à rétroaction de Gallois, c'est l'inverse :

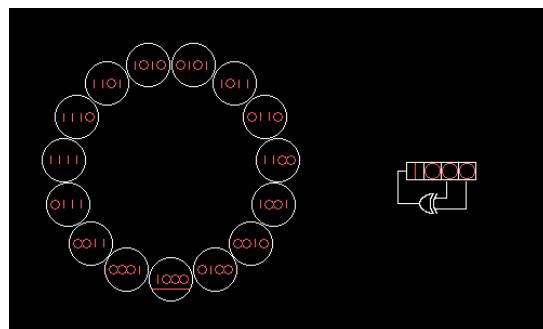
- on prend le bit qui sort du nombre lors d'un décalage ;
- et on en déduit plusieurs bits à partir d'un circuit combinatoire ;
- et on fait rentrer ces bits à divers endroits bien choisis de notre registre à décalage.

Bien sûr, la fonction qui calcule des différents bits à partir du bit d'entrée conserve les mêmes propriétés que celle utilisée pour les registres à décalages à rétroaction linéaire : elle est affine ou linéaire, et se calcule avec uniquement des portes XOR pour les fonctions linéaires, ou NXOR.

pour les fonctions affines.



Ces registres à décalage à rétroaction servent surtout pour fabriquer des suites de nombres aléatoires. Mais il ne s'agit pas de « vrai » aléatoire, vu qu'un tel circuit est déterministe : pour le même résultat en entrée, il donnera toujours le même résultat en sortie. De plus, ce registre ne peut contenir qu'un nombre fini de valeurs, ce qui fait qu'il finira donc par repasser par une valeur qu'il aura déjà parcourue. Lorsque cela arrive, son fonctionnement se reproduit à l'identique comparé à son passage antérieur, vu que le circuit est déterministe. Un tel registre finit donc par faire des cycles. La période d'un de ces cycle dépend fortement de la fonction utilisée pour calculer le bit de sortie, des bits choisis, etc. Dans le meilleur des cas, le registre à décalage à rétroaction passera par toutes les valeurs que le registre peut prendre, à l'exception d'une (suivant le registre, le zéro ou sa valeur maximale sont interdits). Si un registre à rétroaction linéaire passe par zéro (ou sa valeur maximale), il y reste bloqué définitivement. Cela vient du fait que $x \cdot An + Y \cdot An - 1 + \dots + Z \cdot a_0$ donne toujours zéro. Le même raisonnement peut être tenu pour les registres à rétroaction affine, sauf que cette fois-ci, le raisonnement ne marche qu'avec la valeur maximale stockable dans le registre. Tout le challenge consiste donc à trouver quels sont les registres à rétroaction dont la période est maximale.

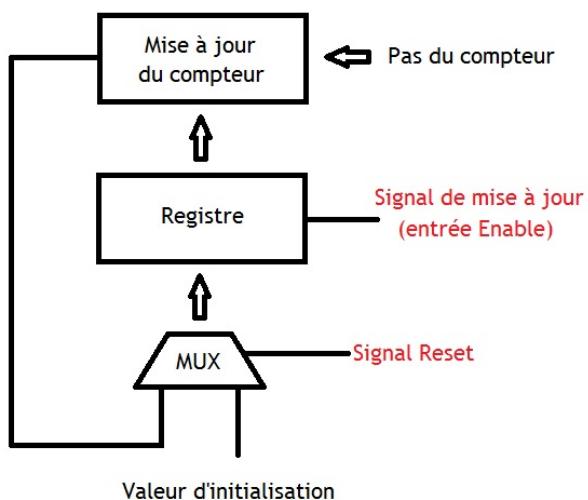


Compteurs et décompteurs : des circuits qui comptent

Les compteurs/décompteurs sont des circuits électroniques qui mémorisent un nombre qu'ils mettent à jour régulièrement. Cette mise à jour augmente ou diminue le compteur d'une quantité fixe, appelée le pas du compteur. Les compteurs augmentent le contenu du compteur d'une quantité fixe à chaque mise à jour, alors que les décompteurs le diminuent. Les compteurs-décompteurs peuvent faire les deux, suivant ce qu'on leur demande.

2^4	2^3	2^2	2^1	2^0
16	8	4	2	1
0	0	0	0	0

Suivant le compteur, la représentation du nombre mémorisé change : certains utilisent le binaire traditionnel, d'autres le BCD, d'autre le code Gray, etc. Tous utilisent un registre pour mémoriser le nombre, ainsi que des circuits combinatoires pour calculer la prochaine valeur du compteur. Ce circuit combinatoire est le plus souvent, mais pas toujours, un circuit capable de réaliser des additions (compteur), des soustractions (décompteurs), voire les deux (compteur-décompteur). La plupart des compteurs utilisent un pas constant, qui est fixé à la création du compteur, ce qui simplifie la conception du circuit combinatoire. D'autres permettent un pas variable, et ont donc une entrée supplémentaire sur laquelle on peut envoyer le pas du compteur. On peut initialiser les compteurs avec la valeur de notre choix : ils possèdent une entrée d'initialisation sur laquelle on peut placer le nombre initial, couplée à une entrée Reset qui indique si le compteur doit être réinitialisé ou non.

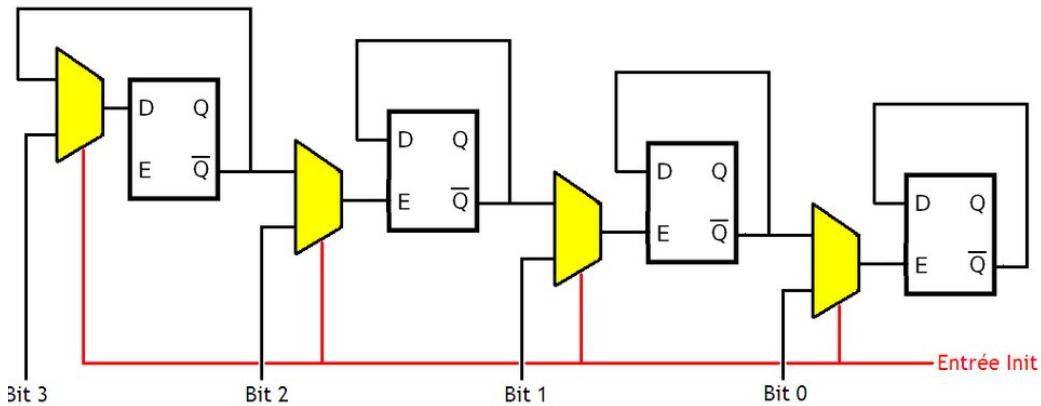


Incrémenteur/décrémenteur

Certains compteurs, aussi appelés **incrémenteurs** comptent de un en un. Les décompteurs analogues sont appelés des **décrémenteurs**. Nous allons voir comment créer ceux-ci dans ce qui va suivre. Il faut savoir qu'il existe deux méthodes pour créer des incrémentateurs/décrémentateurs. La première donne ce qu'on appelle des incrémentateurs asynchrones, et l'autre des incrémentateurs synchrones. Pour comprendre la première méthode, il suffit de regarder la séquence des premiers entiers, puis de prendre des paires de colonnes adjacentes :

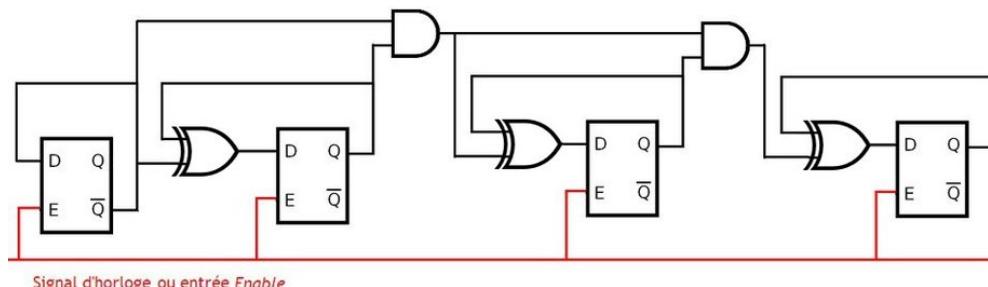
- 000 ;
- 001 ;
- 010 ;
- 011 ;
- 100 ;
- 101 ;
- 110 ;
- 111.

On remarque que le bit sur une colonne change quand le bit de la colonne précédente passe de 1 à 0. Maintenant que l'on sait cela, on peut facilement créer un compteur avec quelques bascules. Pour la colonne la plus à droite (celle des bits de poids faible), on remarque que celle-ci inverse son contenu à chaque cycle d'horloge. Pour cela, on utilise le fait que certaines bascules contiennent une sortie qui fournit l'inverse du bit stocké dans la bascule : il suffit de boucler cette sortie sur l'entrée de la bascule. Pour les autres colonnes, il faut que l'inversion du bit ne se produise que lorsque le bit de la bascule précédente passe de 1 à 0. Le mieux est d'autoriser la mise à jour une fois la transition de la colonne précédente effectuée, c'est à dire quand le bit de la colonne précédente vaut 0. Ainsi, la méthode vue au-dessus reste valable à un changement près : l'entrée Enable de la bascule n'est pas reliée au signal d'horloge, mais à l'inverse de la sortie de la bascule de la colonne précédente. Il reste alors à ajouter les multiplexeurs pour l'initialisation du compteur. On obtient le circuit décrit dans le schéma qui suit.



Un décrémenteur est strictement identique à un incrémenteur auquel on a inversé tous les bits. On peut donc réutiliser le compteur du dessus, à part que les sorties du compteurs sont reliées aux sorties Q des bascules.

Avec la seconde méthode, on repart de la séquence des premiers entiers. On peut alors remarquer que peu importe la colonne, un bit s'inversera (à la prochaine mise à jour) quand tous les bits des colonnes précédentes valent 1. Pour planter cela en circuit, on a besoin de deux circuits par bascules : un qui détermine si les bits des colonnes précédentes sont à 1, et un autre qui inverse le bit de la bascule. Le circuit qui détermine si tous les bits précédents sont à 1 est un simple ET entre les bits en question. L'autre circuit prend en entrée le contenu de la bascule et un bit qui indique s'il faut inverser ou pas. En écrivant sa table de vérité, on s'aperçoit qu'il s'agit d'un simple XOR.

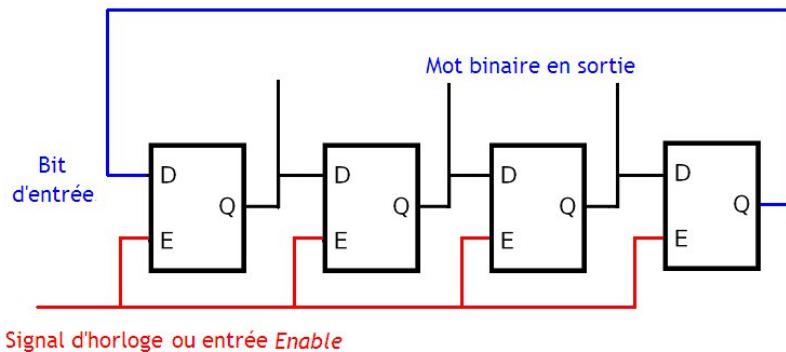


On peut appliquer la même logique pour un décrémenteur. Avec ce circuit, un bit s'inverse lorsque tous les bits précédents sont à zéro. En utilisant le même raisonnement que celui utilisé pour concevoir un incrémenteur, on obtient un circuit presque identique, si ce n'est que les sorties des bascules doivent être inversées avant d'être envoyées à la porte XOR qui suit.

Compteurs en anneau et de Johnson

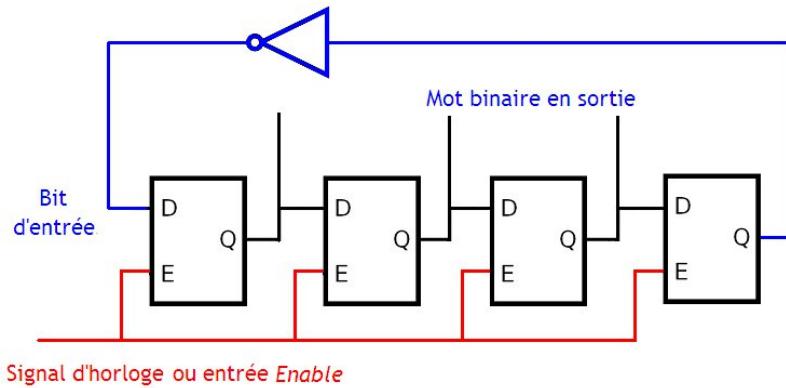
En plus des compteurs précédents, on trouve des compteurs plus simples à fabriquer, qui ont donc tendance à être plus rapides que leurs concurrents. Les **compteurs en anneau** sont des registres à décalage SIPO dont on a bouclé la sortie sur l'entrée. Avec n bits, ce compteur peut compter avec n nombres différents, qui ont tous un seul bit à 1. Petit détail : on peut créer un compteur "normal" en reliant un compteur en anneau avec un encodeur : la sortie de l'encodeur nous donne la valeur normale du compteur. La séquence de ce compteur 4 bits est celle-ci :

- 1000 ;
- 0100 ;
- 0010 ;
- 0001.



Dans d'autres cas, le bit de sortie est inversé avant d'être bouclé sur l'entrée : ce sont des **compteurs de Johnson**. Ce compteur a une limite supérieure double de celle d'un compteur en anneau. La séquence d'un tel compteur est :

- 1000 ;
- 1100 ;
- 1110 ;
- 1111 ;
- 0111 ;
- 0011 ;
- 0001 ;
- 0000.



Tic, tac, tic, tac : le signal d'horloge

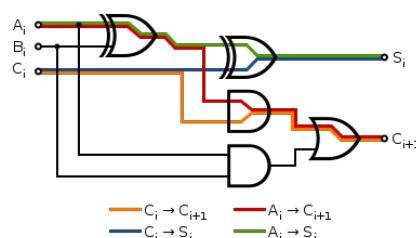
Maintenant que l'on sait créer des registres, nous sommes proches de pouvoir créer n'importe quel circuit séquentiel. Pour l'instant, on peut créer des circuits séquentiels simples, qui ne contiennent qu'un seul registre. Mais la majorité des circuits séquentiels présents dans un ordinateur possèdent plusieurs registres. Si on utilise plusieurs registres, c'est avant tout pour mémoriser des informations différentes : on pourrait les mémoriser dans un seul registre, mais utiliser un registre par nombre à mémoriser est de loin la méthode la plus intuitive.

Sauf qu'un léger détail vient mettre son grain de sel : tous les circuits combinatoires, qui mettent à jour les registres, ne vont pas à la même vitesse ! En conséquence, les registres d'un composant ne sont pas mis à jour en même temps, ce qui pose quelques problèmes relativement fâcheux si aucune mesure n'est prise.

Temps de propagation

Pour commencer, il nous faut expliquer pourquoi tous les circuits combinatoires ne vont pas à la même vitesse. Tout circuit, quel qu'il soit, va mettre un petit peu de temps avant de réagir. Ce temps mis par le circuit pour propager un changement sur les entrées vers la sortie s'appelle le temps de propagation. Pour faire simple, c'est le temps que met un circuit à faire ce qu'on lui demande : plus ce temps de propagation est élevé, plus le circuit est lent. Ce temps de propagation dépend de pas mal de paramètres, aussi je ne vais citer que les trois principaux.

- Le premier facteur que nous allons aborder est le **nombre de portes logiques reliées sur un même fil**. Plus on connecte de portes logiques sur un fil, plus il faudra du temps pour que la tension à l'entrée de ces portes passe de 1 à 0 (ou inversement).
- Ensuite, vient le **temps de propagation dans les fils**, celui mis par notre tension pour se propager dans les fils qui relient les portes logiques entre elles. Ce temps perdu dans les fils devient de plus en plus important au cours du temps, les transistors et portes logiques devenant de plus en plus rapides à force des miniaturisations. Par exemple, si vous comptez créer un circuit avec des entrées de 256 à 512 bits, il vaut mieux le modifier pour minimiser le temps perdu dans les interconnexions que de diminuer le chemin critique.
- Mais le plus important de ces paramètres est ce qu'on appelle le **chemin critique**, le nombre maximal de portes logiques entre une entrée et une sortie de notre circuit. Pour donner un exemple, nous allons prendre le schéma suivant. Pour ce circuit, le chemin critique est dessiné en rouge. En suivant ce chemin, on va traverser trois portes logiques, contre deux ou une dans les autres chemins.

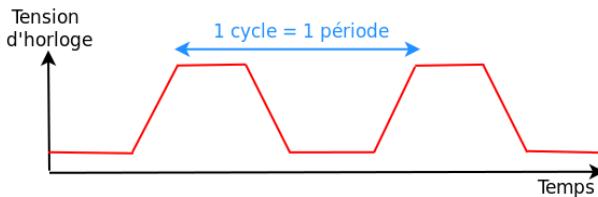


Circuits synchrones

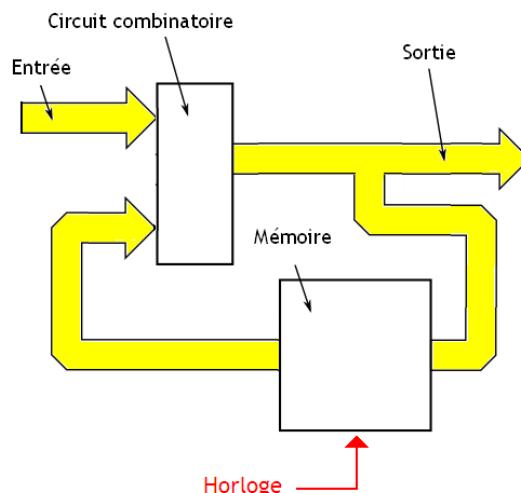
Ce temps de propagation doit être pris en compte quand on crée un circuit séquentiel : sans ça on ne sait pas quand mettre à jour la mémoire dans le circuit. Si on le fait trop tôt, le circuit combinatoire peut sauter des états : il se peut parfaitement qu'on change le bit placé sur l'entrée avant qu'il soit mémorisé. De plus, les différents circuits d'un composant électronique n'ont pas tous le même temps de propagation, et ceux-ci vont fonctionner à des vitesses différentes. Si l'on ne fait rien, on peut se retrouver avec des dysfonctionnements : par exemple, un circuit lent peut rater deux ou trois nombres envoyées par un composant un peu trop rapide.

Pour éviter les ennuis dus à l'existence de ce temps de propagation, il existe deux grandes solutions, qui permettent de faire la différence entre circuits asynchrones et synchrones. Les **circuits asynchrones** préviennent la mémoire quand ils veulent la mettre à jour. Quand le circuit combinatoire et la mémoire sont tous les deux prêts, on autorise l'écriture dans la mémoire. Mais ce n'est pas cette solution qui est utilisée dans les circuits de nos ordinateurs, qui sont des circuits synchrones. Dans les **circuits synchrones**, les registres sont tous mis à jour en même temps. On peut remarquer que c'est quelque chose d'analogique à ce qu'on trouve sur les registres : si toutes les bascules d'un registre doivent être mises à jour en même temps, tous les registres d'un circuit séquentiel doivent être mis à jour en même temps. La solution est donc similaire à celle utilisée sur les registres : on commande la mise à jour des registres par un signal d'autorisation d'écriture, qui est transmis à tous les registres en même temps.

Généralement, ces circuits mettent à jour leurs mémoires à intervalles réguliers. La durée entre deux mises à jour est constante et doit être plus grande que le pire temps de propagation possible du circuit : on se base donc sur le circuit combinatoire le plus lent. Les concepteurs d'un circuit doivent estimer le pire temps de propagation possible pour le circuit et ajouter une marge de sécurité. Pour mettre à jour nos circuits à intervalles réguliers, le signal d'autorisation d'écriture est une tension qui varie de façon cyclique : on parle alors de **signal d'horloge**. Le temps que met la tension pour effectuer un cycle est ce qu'on appelle la **période**. Le nombre de périodes par seconde est appelé la **fréquence**. Elle se mesure en hertz. On voit sur ce schéma que la tension ne peut pas varier instantanément : elle met un certain temps pour passer de 0 à 1 et de 1 à 0. On appelle cela un **front**. La passage de 0 à 1 est appelé un front montant et le passage de 1 à 0 un front descendant.



Cette horloge est reliée aux entrées d'autorisation d'écriture des bascules du circuit. Pour cela, on doit rajouter une entrée sur notre circuit, sur laquelle on enverra l'horloge. En faisant cela, notre circuit logique mettra ses sorties à jour lors d'un front montant (ou descendant) sur son entrée d'horloge. Entre deux fronts montants (ou descendants), le circuit ne réagit pas aux variations des entrées. Dans le cas où notre circuit est composé de plusieurs sous-circuits devant être synchronisés via l'horloge, celle-ci est distribuée à tous les sous-circuits à travers un réseau de connexions électriques qu'on appelle l'**arbre d'horloge**.



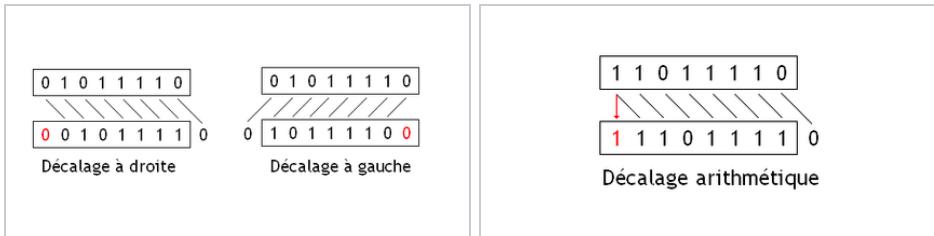
Dans un ordinateur moderne, chaque composant d'un ordinateur a sa propre horloge, qui peut être plus ou moins rapide que les autres : par exemple, notre processeur fonctionne avec une horloge différente de l'horloge de la mémoire ! Ces signaux d'horloge dérivent d'une horloge de base qui est « transformée » en plusieurs horloges, grâce à des montages électroniques spécialisés (des PLL ou des montages à portes logiques un peu particuliers). La présence de plusieurs horloges vient du fait que certains composants informatiques sont plus lents que d'autres. Plutôt que de caler tous les composants d'un ordinateur sur le plus lent en utilisant une horloge, il vaut mieux utiliser des horloges différentes pour chaque composant : les mises à jour de registres sont synchronisées à l'intérieur d'un composant (dans un processeur, ou une mémoire), alors que les composants eux-mêmes synchronisent leurs communications avec d'autres mécanismes.

Les circuits de calcul

Tout circuit de calcul peut être conçu les méthodes vues au chapitre précédent. Mais les circuits de calcul actuels manipulent des nombres de 32 bits, ce qui demanderait des tables de vérité démesurément grandes : plus de 4 milliards de lignes ! Il faut donc ruser, pour créer des circuits économies en transistors et rapides. C'est le but de ce chapitre : voir comment sont réalisés les circuits de calcul.

Décalages et rotations

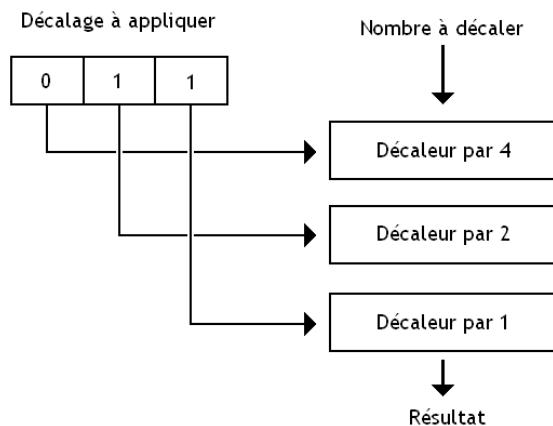
On va commencer par les circuits de **décalage**, qui décalent un nombre de un ou plusieurs rangs vers la gauche, ou la droite. Le nombre à décaler est envoyé sur une entrée du circuit, de même que le nombre de rangs l'est sur une autre. Le circuit fournit le nombre décalé sur sa sortie. Lorsqu'on décale un nombre, certains bits sont inconnus, ce qui laisse des vides dans le nombre. On peut les remplir soit avec des zéros, ce qui donne un décalage logique. Mais pour les décalages à droite, on peut aussi remplir les vides avec le bit de poids fort (on verra plus tard pourquoi dans le cours). On parle alors de décalage arithmétique.



Décalage logique.

Décalage arithmétique.

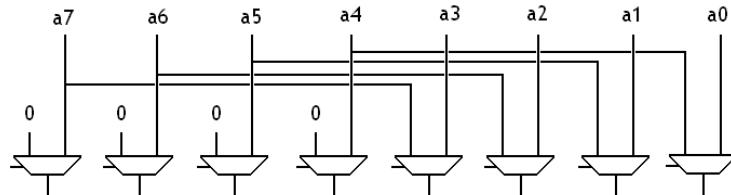
Ces circuits sont fabriqués avec des multiplexeurs à deux entrées et une sortie. Nous allons voir comment créer un circuit capable de décaler un nombre vers la droite, d'un nombre de rangs variable : on pourra décaler notre nombre de 2 rangs, de 3 rangs, de 4 rangs, etc. Il faudra préciser le nombre de rangs sur une entrée. On peut faire une remarque simple : décaler vers la droite de 6 rangs, c'est équivalent à décaler notre nombre vers la droite de 4 rangs, et redécaler le tout de 2 rangs. Même chose pour 7 rangs : cela consiste à décaler de 4 rangs, redécaler de 2 rangs et enfin redécaler d'un rang. En suivant notre idée jusqu'au bout, on se rend compte qu'on peut créer un décaleur à partir de décaleurs plus simples, reliés en cascade, qu'on d'active ou désactive suivant le nombre de rangs. Le nombre de rangs par lequel on va devoir décaler est un nombre codé en binaire, qui s'écrit donc sous la forme d'une somme de puissances de deux. Chaque bit du nombre de rang servira à actionner le décaleur qui déplace d'un nombre égal à sa valeur (la puissance de deux qui correspond en binaire).



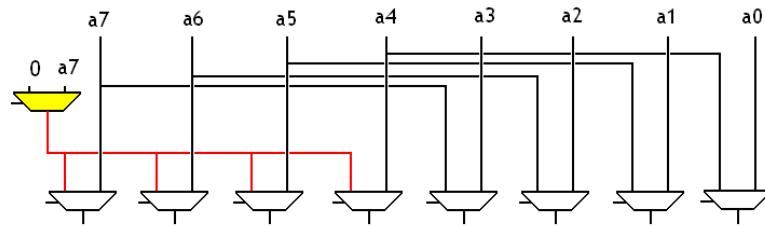
Reste à savoir comment créer ces décaleurs qu'on peut activer ou désactiver à la demande. On va prendre comme exemple un décaleur par 4, mais ce que je vais dire peut être adapté pour créer des décaleurs par 1, par 2, par 8, etc. La sortie vaudra soit le nombre tel qu'il est passé en entrée (le décaleur est inactif), soit le nombre décalé de 4 rangs. Ainsi, si je prend un nombre A, composé des bits a₇, a₆, a₅, a₄, a₃, a₂, a₁, a₀ ; (cités dans l'ordre), le résultat sera :

- soit le nombre composé des chiffres a₇, a₆, a₅, a₄, a₃, a₂, a₁, a₀ (on n'effectue pas de décalage) ;
- soit le nombre composé des chiffres 0, 0, 0, 0, a₇, a₆, a₅, a₄ (on effectue un décalage par 4).

Chaque bit de sortie peut prendre deux valeurs, qui valent soit zéro, soit un bit du nombre d'entrée. On peut donc utiliser un multiplexeur pour choisir quel bit envoyer sur la sortie. Par exemple, pour le choix du bit de poids faible du résultat, celui-ci vaut soit a₇, soit 0 : il suffit d'utiliser un multiplexeur prenant le bit a₇ sur son entrée 1, et un 0 sur son entrée 0. Il suffit de faire la même chose pour tous les autres bits, et le tour est joué.



On peut modifier le schéma vu au-dessus pour lui permettre d'effectuer des décalages arithmétiques en plus des décalages logiques. Il suffit simplement d'ajouter un ou plusieurs multiplexeurs pour chaque décaleur élémentaire par 1, 2, 4, etc. Ces multiplexeurs choisissent quoi envoyer sur l'entrée de l'ancienne couche : soit un 0 (décalage logique), soit le bit de signe (décalage arithmétique).

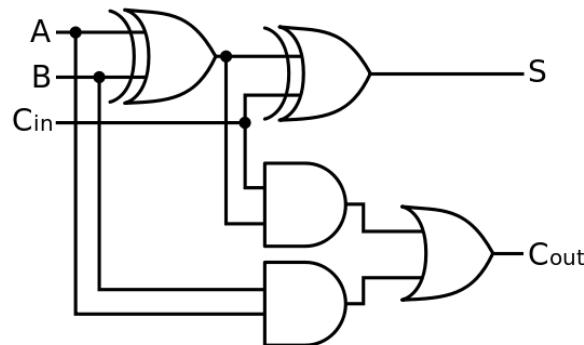


Addition non signée

Voyons maintenant un circuit capable d'additionner deux nombres : l'additionneur. Dans la version qu'on va voir, ce circuit manipulera des nombres strictement positifs ou des nombres codés en complément à deux, ou en complément à un. Tout additionneur est composé d'additionneurs plus simples, capables d'additionner deux ou trois bits suivant la situation. Un additionneur deux bits correspond à un circuit qui contient la table d'addition, qui est très simple en binaire :

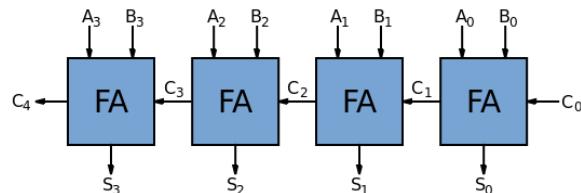
- $0 + 0 = 0$;
- $0 + 1 = 1$;
- $1 + 0 = 1$;
- $1 + 1 = 0$, plus une retenue.

Un circuit qui additionne trois bits est appelé un additionneur complet.



Additionneur à propagation de retenue

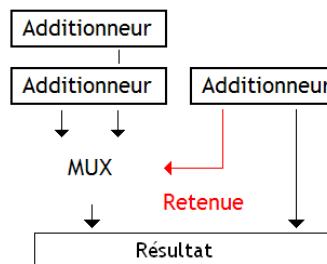
L'additionneur à propagation de retenue pose l'addition comme en décimal, en additionnant les bits colonne par colonne avec une éventuelle retenue. Évidemment, on commence par les bits les plus à droite, comme en décimal. Pour additionner les chiffres sur une colonne, on utilise notre connaissance de la table d'addition. Il suffit ainsi de câbler des additionneurs complets les uns à la suite des autres. Notez la présence de l'entrée de retenue C. Presque tous les additionneurs de notre ordinateur ont une entrée de retenue comme celle-ci, afin de faciliter l'implémentation de certaines opérations comme l'inversion de signe, l'incrémentation, etc.



Ce circuit a un gros problème : chaque additionneur doit attendre que la retenue de l'addition précédente soit disponible pour donner son résultat. Les retenues doivent se propager à travers le circuit, du premier additionneur jusqu'au dernier. Or, l'addition est une opération très fréquente dans nos programmes : il faut créer des additionneurs plus rapides.

Additionneur à sélection de retenue

Pour cela, on peut casser l'additionneur à propagation de retenue en plusieurs petits additionneurs qu'on organise différemment : on obtient un additionneur à sélection de retenue. Cet additionneur va découper nos deux nombres à additionner en blocs, qui se feront additionner en deux versions : une avec la retenue du bloc précédent valant zéro, et une autre version avec la retenue du bloc précédent valant 1. Il suffira alors de choisir le bon résultat avec un multiplexeur, une fois cette retenue connue. On gagne ainsi du temps en calculant à l'avance les valeurs de certains bits du résultat, sans connaître la valeur de la retenue. Petit détail : sur certains additionneurs à sélection de retenue, les blocs de base n'ont pas la même taille. Cela permet de tenir compte des temps de propagation des retenues entre les blocs.



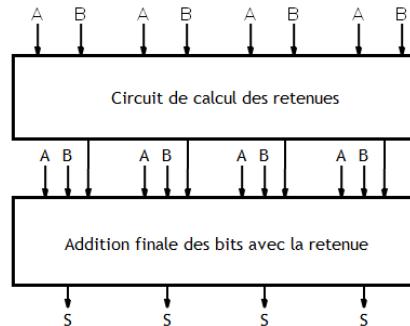
Dans les exemples du dessus, chaque sous-additionneur étaient des additionneurs à propagation de retenue. Mais ce n'est pas une obligation, et

tout autre type d'additionneur peut être utilisé. Par exemple, on peut faire en sorte que les sous-additionneurs soient eux-mêmes des additionneurs à sélection de retenue, et poursuivre ainsi de suite, récursivement. On obtient alors un **additionneur à somme conditionnelle**, plus rapide que l'additionneur à sélection de retenue, mais qui utilise beaucoup plus de portes logiques.

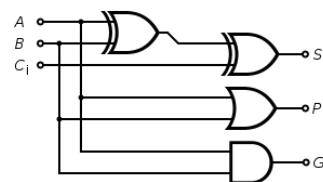
Additionneurs à anticipation de retenue

Au lieu de calculer les retenues une par une, certains additionneurs calculent toutes les retenues en parallèle à partir de la valeur de tout ou partie des bits précédents. Une fois les retenues pré-calculées, il suffit de les additionner avec les deux bits adéquats, pour obtenir le résultat. Ces additionneurs sont appelés des additionneurs à anticipation de retenue. Ces additionneurs sont composés de deux parties :

- un circuit qui pré-calcule la valeur de la retenue d'un étage ;
- et d'un circuit qui additionne les deux bits et la retenue pré-calculée : il s'agit d'une couche d'additionneurs complets simplifiés, qui ne fournissent pas de retenue.

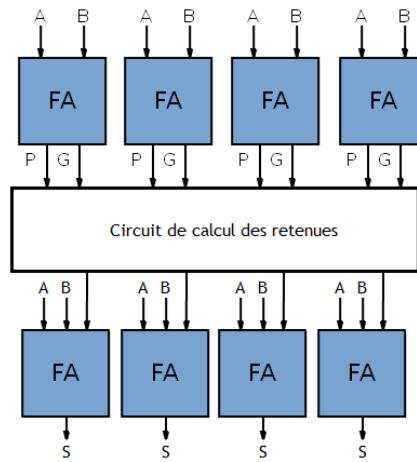


Le circuit qui détermine la valeur de la retenue est lui-même composé de deux grandes parties, qui ont chacune leur utilité. Pour commencer, nous allons faire un petit point de vocabulaire. Une addition de deux bits a et b génère une retenue si l'addition de a et de b suffit à donner une retenue en sortie, quelle que soit la retenue envoyée en entrée. De même, l'addition de deux bits propage une retenue si l'addition des deux bits donne une retenue seulement si une retenue est fournie en entrée. Nous aurons besoin d'un circuit qui indique si d'addition de deux bits génère ou propage une retenue : ce circuit a besoin de deux sorties P et G pour indiquer s'il y a propagation et génération de retenue. Le circuit final est donc composé d'un ensemble de ces briques de base, une pour chaque paire de bits.



L'additionneur commence donc à prendre forme, et est composé de trois parties :

- un circuit qui crée les signaux P et G ;
- un circuit qui déduit la retenue à partir des signaux P et G adéquats ;
- et une couche d'additionneurs qui additionnent chacun deux bits et une retenue.



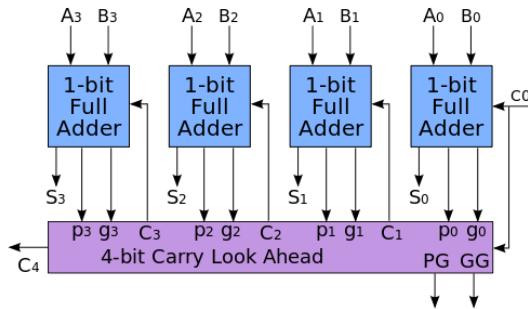
Il ne nous reste plus qu'à voir comment fabriquer le circuit qui reste. Pour cela, il faut remarquer que la retenue est égale :

- à 1 si l'addition des deux bits génère une retenue ;
- à 1 si l'addition des deux bits propage une retenue ;
- à zéro sinon.

Ainsi, l'addition des bits de rangs i va produire une retenue C_i , qui est égale à $G_i + (P_i \cdot C_{i-1})$. Si on utilisait cette formule sans trop réfléchir, on retomberait sur un additionneur à propagation de retenue inutilement compliqué. L'astuce des additionneurs à anticipation de retenue consiste à remplacer le terme C_{i-1} par sa valeur calculée avant. Par exemple, je prends un additionneur 4 bits. Je dispose de deux nombres A et B , contenant chacun 4 bits : A_3, A_2, A_1 , et A_0 pour le nombre A , et B_3, B_2, B_1 , et B_0 pour le nombre B . Si j'effectue les remplacements, j'obtiens les formules suivantes :

- $C_1 = G_0 + (P_0 \cdot C_0)$;
- $C_2 = G_1 + (P_1 \cdot G_0) + (P_1 \cdot P_0 \cdot C_0)$;
- $C_3 = G_2 + (P_2 \cdot G_1) + (P_2 \cdot P_1 \cdot G_0) + (P_2 \cdot P_1 \cdot P_0 \cdot C_0)$;
- $C_4 = G_3 + (P_3 \cdot G_2) + (P_3 \cdot P_2 \cdot G_1) + (P_3 \cdot P_2 \cdot P_1 \cdot G_0) + (P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0)$.

Ces formules nous permettent de déduire la valeur d'une retenue directement : il reste alors à créer un circuit qui implémente ces formules, et le tour est joué. On peut même simplifier le tout en fusionnant les deux couches d'additionneurs.



Ces additionneurs sont plus rapides que les additionneurs à propagation de retenue. Ceci dit, utiliser un additionneur à anticipation de retenue sur des nombres très grands (16/32bits) utiliserait trop de portes logiques. Pour éviter tout problème, nos additionneurs à anticipation de retenue sont souvent découpés en blocs, avec soit une anticipation de retenue entre les blocs et une propagation de retenue dans les blocs, soit l'inverse.

Additionneur à calcul parallèle de préfixes

D'autres additionneurs modifient un petit peu ce principe. Ceux-ci sont toujours découplés en trois couches :

- un circuit qui crée les signaux P et G ;
- un circuit qui déduit la retenue à partir des signaux P et G adéquats ;
- et une couche d'additionneurs qui additionnent chacun deux bits et une retenue.

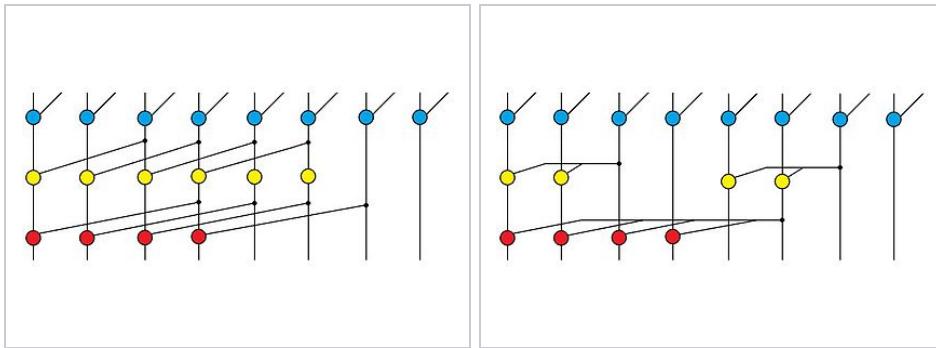
Simplement, ils vont concevoir le circuit de calcul des retenues différemment. Avec eux, le calcul $G_i + (P_i \cdot C_{i-1})$ va être modifié pour prendre en entrée non pas la retenue C_{i-1} , mais les signaux G_{i-1} et P_{i-1} . Dans ce qui va suivre, nous allons noter ce petit calcul o. On peut ainsi écrire que :

$$C_i = (((G_i, P_i) o (G_{i-1}, P_{i-1})) o (G_{i-2}, P_{i-2})) o (G_{i-3}, P_{i-3}) \dots$$

Si on utilisait cette formule sans trop réfléchir, on retomberait sur un additionneur à propagation de retenue inutilement compliqué. Le truc, c'est que o est associatif, et que cela peut permettre de créer pas mal d'optimisations : il suffit de réorganiser les parenthèses. Cette réorganisation peut se faire de diverses manières qui donnent des additionneurs différents :

- l'additionneur de Ladner-Fisher ;
- l'additionneur de Brent-Kung ;
- l'additionneur de Kogge-Stone ;
- ou tout design hybride.

L'additionneur de Brent-Kung est le plus lent de tous les additionneurs cités, mais c'est celui qui utilise le moins de portes logiques. L'additionneur de Ladner-Fisher est théoriquement le plus rapide de tous, mais aussi celui qui utilise le plus de portes logiques. Les autres sont des intermédiaires.



Additionneur de Kogge-Stone.

Additionneur de Ladner-Fisher.

Addition multiopérande

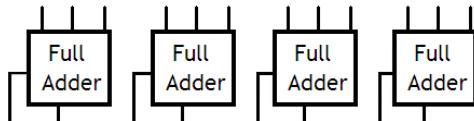
Après avoir vu comment additionner deux nombres, il est temps de voir comment faire la même chose avec plus de deux nombres. Il existe de nombreux types de circuits capables d'effectuer cette addition multiopérande, construits à partir d'additionneurs simples. La solution la plus simple consiste à additionner les nombres les uns après les autres, soit avec un circuit séquentiel, soit en enchainant des additionneurs les uns à la suite des autres. Mais diverses optimisations sont possibles.

Additionneurs carry-save

Les additionneurs carry-save se basent sur une nouvelle représentation des nombres, particulièrement adaptée aux additions successives : la représentation carry-save. Avec l'addition carry-save, on ne propage pas les retenues. L'addition de trois bits en carry-save fournit deux résultats : un résultat obtenu en effectuant l'addition sans tenir compte des retenues, et un autre composé uniquement des retenues. Par exemple, 1000 + 1010 + 1110 donne 1010 pour les retenues, et 1100 pour la somme sans retenue.

	A3	A2	A1	A0			
+	B3	B2	B1	B0			
+	K3	K2	K1	K0			
<hr/>							
	S3	S2	S1	S0	Somme		
	C3	C2	C1	C0		Retenues	

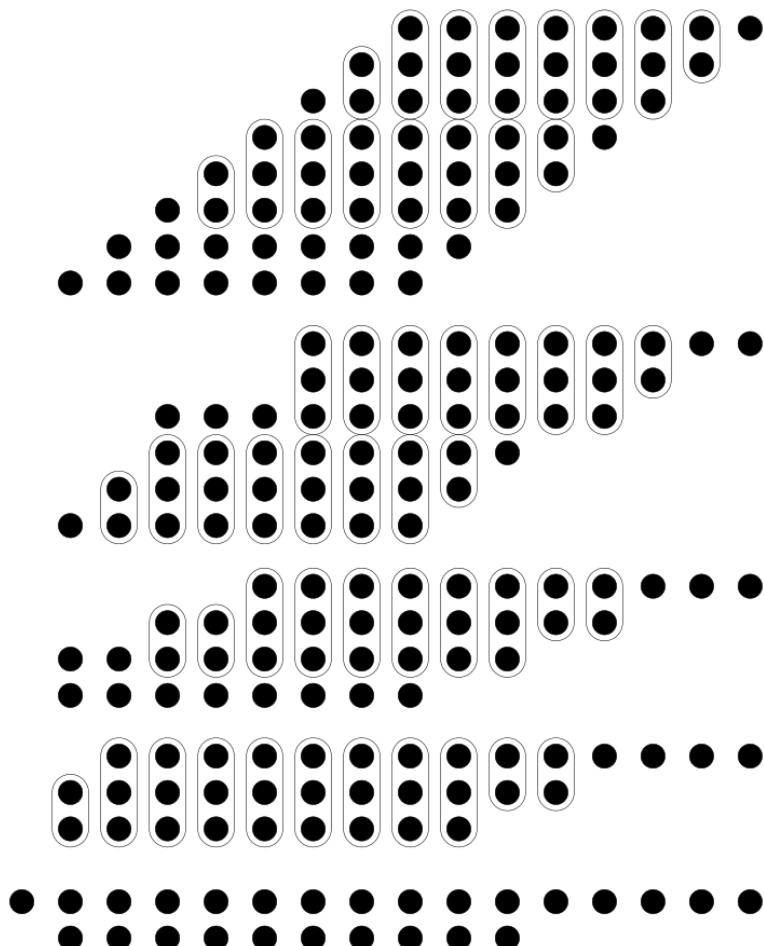
Ainsi, additionner trois nombres devient très facile : il suffit d'additionner les trois nombres avec un additionneur carry-save, et d'additionner les deux résultats avec un additionneur normal. Le même principe peut s'appliquer avec plus de trois nombres : il suffit juste d'assembler plusieurs additionneurs carry-save les uns à la suite des autres.



Chainage d'additionneurs en arbre

Enchaîner les additionneurs les uns à la suite des autres n'est pas la meilleure solution. Le mieux est de les organiser en arbre pour effectuer certaines additions en parallèle d'autres. On peut bien évidemment utiliser des additionneurs normaux dans cet arbre, mais l'idéal est d'utiliser des additionneurs carry-save. En faisant cela, on peut se retrouver avec plusieurs formes pour l'arbre, qui donnent respectivement des additionneurs en arbres de Wallace, ou des arbres Dadda.

Les arbres les plus simples à construire sont les **arbres de Wallace**. Le principe est d'ajouter des couches d'additionneurs carry-save, et de capturer un maximum de nombres lors de l'ajout de chaque couche. On commence par utiliser un maximum de nombres avec des additionneurs carry-save. Pour additionner n nombres, on commence par utiliser $n/3$ additionneurs carry-save. Si jamais n n'est pas divisible par 3, on laisse tranquille les 1 ou 2 nombres restants. On se retrouve ainsi avec une couche d'additionneurs carry-save. On répète cette étape sur les sorties des additionneurs ainsi ajoutés : on rajoute une nouvelle couche. Il suffit de répéter cette étape jusqu'à ce qu'il ne reste plus que deux résultats : on se retrouve avec une couche finale composée d'un seul additionneur carry-save. Là, on rajoute un additionneur normal, pour additionner retenues et sommes.

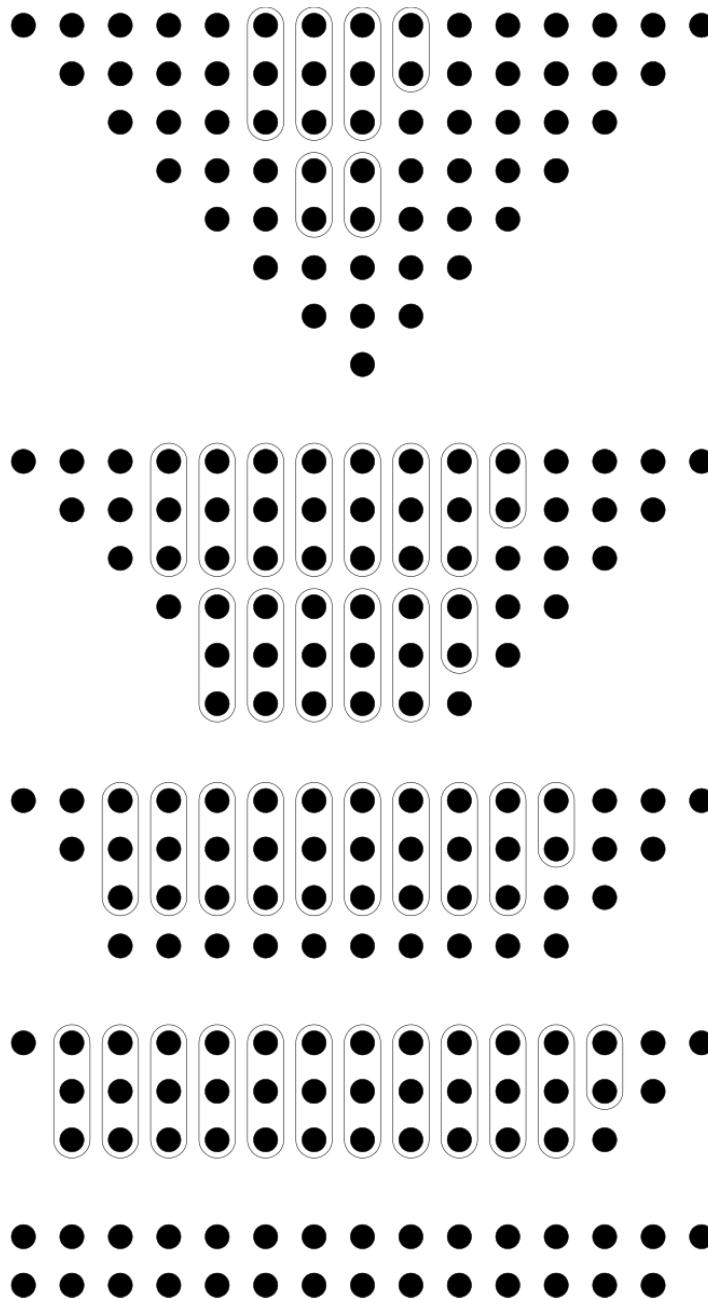


Les **arbres de Dadda** sont plus difficiles à comprendre. Contrairement à l'arbre de Wallace qui cherche à réduire la hauteur de l'arbre le plus vite possible, l'arbre de Dadda cherche à diminuer le nombre d'additionneurs carry-save utilisés. Pour cela, l'arbre de Dadda se base sur un principe mathématique simple : un additionneur carry-save peut additionner trois nombres, pas plus. Cela implique que l'utilisation d'un arbre de Wallace gaspille des additionneurs si on additionne n nombres, avec n non multiple de trois. L'arbre de Dadda résout ce problème d'une manière simple :

- si n est multiple de trois, on ajoute une couche complète d'additionneurs carry-save ;

- si n n'est pas multiple de trois, on ajoute seulement 1 ou 2 additionneur carry-save : le but est de faire en sorte que la couche suivante fournit un nombre d'opérandes multiple de trois.

Et on répète cette étape d'ajout de couche jusqu'à ce qu'il ne reste plus que deux résultats : on se retrouve avec une couche finale composée d'un seul additionneur carry-save. Là, on rajoute un additionneur normal, pour additionner retenues et sommes.

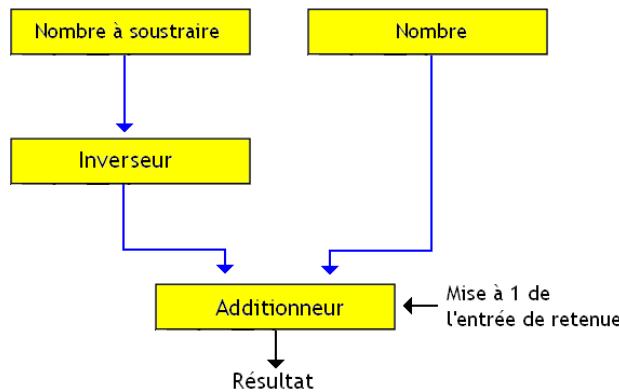


Soustraction et addition signée

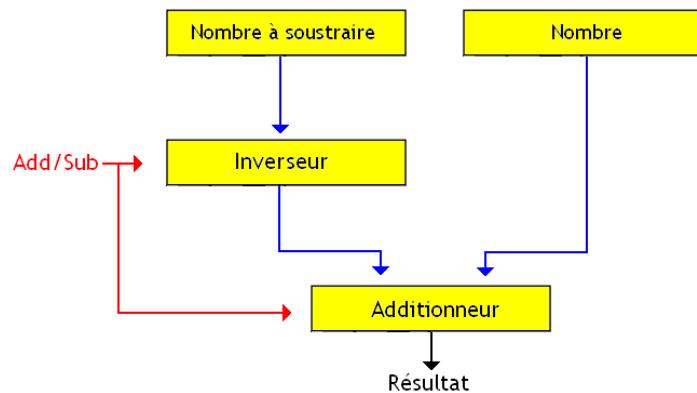
Si on sait câbler une addition entre entiers positifs, câbler une soustraction n'est pas très compliqué. On va commencer par un circuit capable de soustraire deux nombres représentés en complément à deux ou en complément à un.

Complément à deux

Vous savez sûrement que $a - b$ et $a + (-b)$ sont deux expressions équivalentes. Et en complément à deux, $-b = \text{not}(b) + 1$. Dit autrement, $a - b = a + \text{not}(b) + 1$. On pourrait se dire qu'il faut deux additionneurs pour faire le calcul, mais la majorité des additionneurs possède une entrée de retenue pour incrémenter le résultat de l'addition.

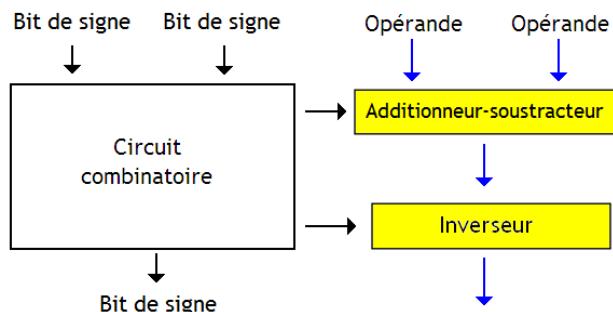


Il est possible de créer un circuit capable d'effectuer soit une addition, soit une soustraction : il suffit de remplacer l'inverseur par un inverseur commandable, qui peut être désactivé. On a vu comment créer un tel inverseur commandable dans le chapitre sur les circuits combinatoires. On peut remarquer que l'entrée de retenue et l'entrée de commande de l'inverseur sont activées en même temps : on peut fusionner les deux signaux en un seul.



Signe-valeur absolue

Passons maintenant aux nombres codés en signe-valeur absolue. On remarque que $B - A$ est égal à $-(A - B)$, et $-A - B$ vaut $-(A + B)$. Ainsi, le circuit n'a besoin que de calculer $A + B$ et $A - B$: il peut les inverser pour obtenir $-A - B$ ou $B - A$. $A + B$ et $A - B$ peuvent se calculer avec un additionneur-soustracteur, auquel on ajoute un inverseur commandable. On peut transformer ce circuit en additionneur-soustracteur en signe-valeur absolue, mais le circuit combinatoire devient plus complexe.



Multiplication

Nous allons maintenant aborder la multiplication, effectuée par un circuit : le multiplicateur. Pour commencer, petite précision de vocabulaire : une multiplication s'effectue sur deux nombres, le multiplicande et le multiplicateur. Comme pour l'addition, nous allons multiplier en binaire de la même façon qu'on a appris à le faire en primaire, si ce n'est que la table de multiplication est modifiée. Une multiplication génère des résultats temporaires, chacun provenant de la multiplication du multiplicande par un chiffre du multiplicateur, auquel on aura appliqué un décalage : ces résultats temporaires sont appelés des produits partiels. Multiplier deux nombres en binaire demande de générer les produits partiels avant de les additionner. Et si les additionneurs multiopérande permettent de faire l'addition des produits partiels, il ne manque plus qu'à les générer.

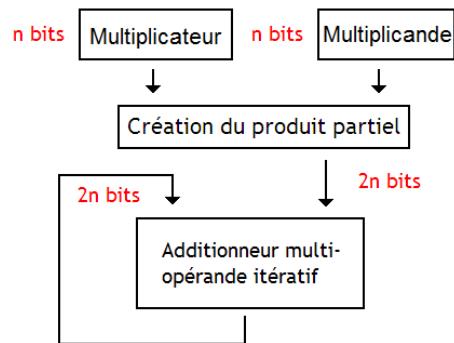
Pour multiplier un nombre par un bit du multiplicateur, rien de plus simple : les tables de multiplication sont vraiment très simples en binaire, jugez plutôt !

- $0 \times 0 = 0$.
- $0 \times 1 = 0$.
- $1 \times 0 = 0$.
- $1 \times 1 = 1$.

Cette table de vérité est celle d'une porte ET. Ainsi, notre circuit est donc très simple : il suffit d'effectuer un ET entre les bits du multiplicande, et le bit du multiplicateur adéquat. Il reste ensuite à faire un décalage (et encore, on peut s'en passer avec quelques optimisations), et le tour est joué.

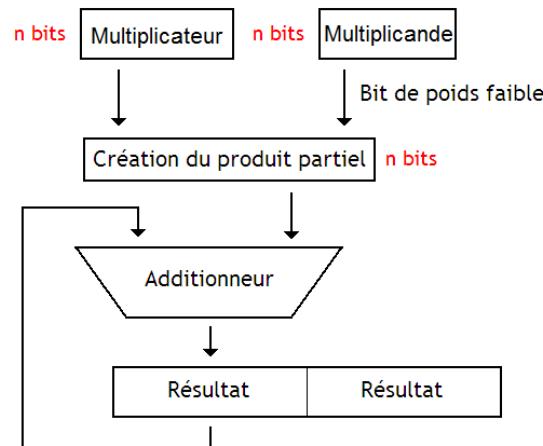
Circuits itératifs

Les multiplicateurs les plus simples génèrent les produits partiels les uns après les autres, et les additionnent au fur et à mesure. Le multiplicateur et le multiplicande sont mémorisés dans des registres. Le reste du circuit est composé d'un circuit de génération des produits partiels, suivi d'un additionneur multiopérande itératif. La multiplication est finie quand tous les bits du multiplicateur ont été traités (ce qui peut se déterminer avec un compteur). Il existe plusieurs multiplicateurs itératifs, qui diffèrent par la façon dont ils génèrent le produit partiel : certains traitent les bits du multiplicateur de droite à gauche, les autres dans le sens inverse. Dans les deux cas, on décale le multiplicateur d'un cran à chaque cycle, et on prend son bit de poids faible. Pour cela, on stocke le multiplicateur dans un registre à décalage.

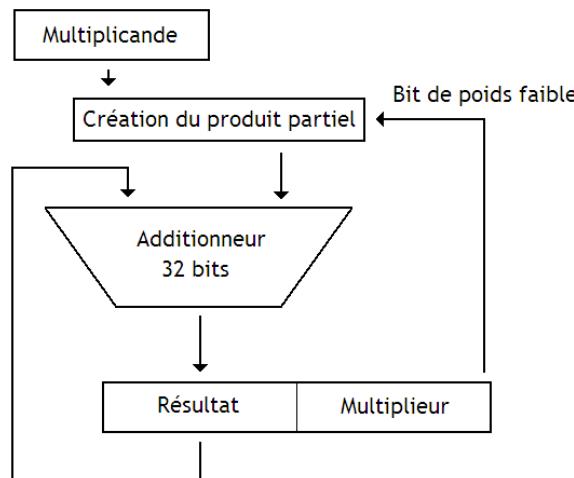


On peut remarquer une chose assez intéressante : si le produit partiel est nul, pourquoi l'ajouter ? Ainsi, si le bit du multiplicateur qui génère le produit partiel est 0, le produit partiel sera nul : on peut se passer de l'addition et ne faire que les décalages.

On peut encore optimiser le circuit en utilisant des produits partiels sur n bits. Pour cela, on fait le calcul en commençant par les bits de poids fort du multiplicateur : on parcourt le multiplicateur de droite à gauche au lieu de faire de gauche à droite. L'astuce, c'est qu'on additionne le produit partiel avec les bits de poids fort du registre pour le résultat, et non les bits de poids faible. Le contenu du registre est décalé d'un cran à droite à chaque cycle, ce qui décale automatiquement les produits partiels comme il faut.



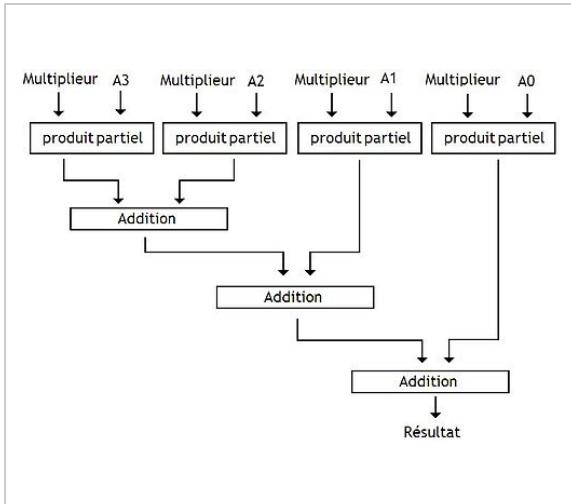
Il est même possible de ruser encore plus : on peut se passer du registre pour le multiplicateur. Il suffit d'initialiser les bits de poids faible du registre résultat avec le multiplicateur au démarrage de la multiplication. Le bit du multiplicateur choisi pour le calcul du produit partiel est simplement le bit de poids faible du résultat.



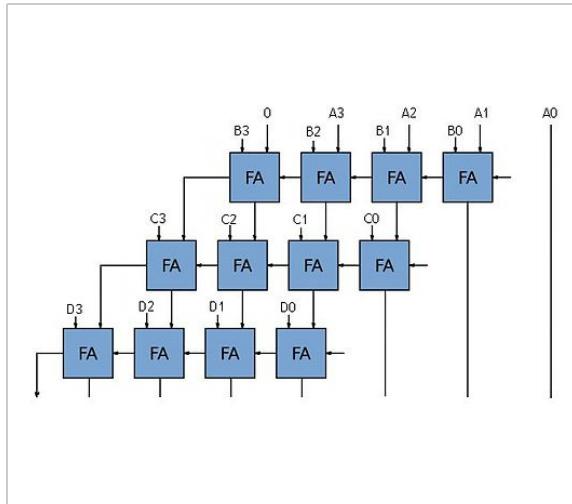
Multiplicateur tableau

Il est possible d'optimiser les multiplicateurs précédents en calculant et en additionnant plusieurs produits partiels à la fois. Il suffit d'un additionneur

multi-opérande et de plusieurs circuits de génération de produits partiels. Toutefois, cette technique demande de prendre en compte plusieurs bits du multiplicateur à chaque cycle : le nombre de rangs à décaler augmente, sans compter que la génération du produit partiel est un peu plus complexe. Il est aussi possible de pousser cette logique plus loin, et de calculer tous les produits partiels en parallèle, en même temps. Dans sa version la plus simple, un circuit basé sur ce principe va simplement enchaîner les additionneurs les uns à la suite des autres. Dans sa version la plus simple, on peut utiliser des additionneurs à propagation de retenue pour créer le multiplicateur. Utiliser d'autres additionneurs ne donnerait que des gains en performance relativement faibles. On peut aussi enchaîner des additionneurs carry-save à trois opérandes pour additionner les produits partiels.



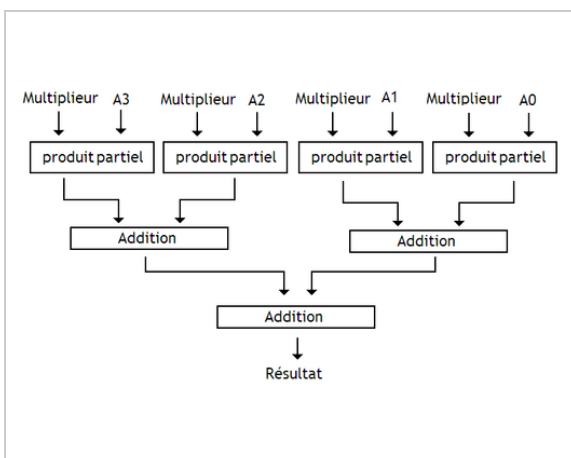
Multiplicateur tableau.



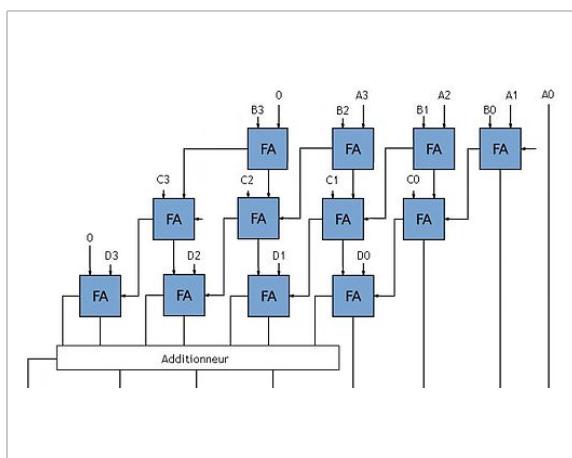
Multiplicateur à propagation de retenues.

Multiplicateurs en arbre

Mais enchaîner les additionneurs les uns à la suite des autres n'est pas la meilleure solution. Le mieux est d'utiliser un additionneur multiopérande en arbre.



Multiplicateur en arbre.



Implémentation d'un multiplicateur tableau carry-save.

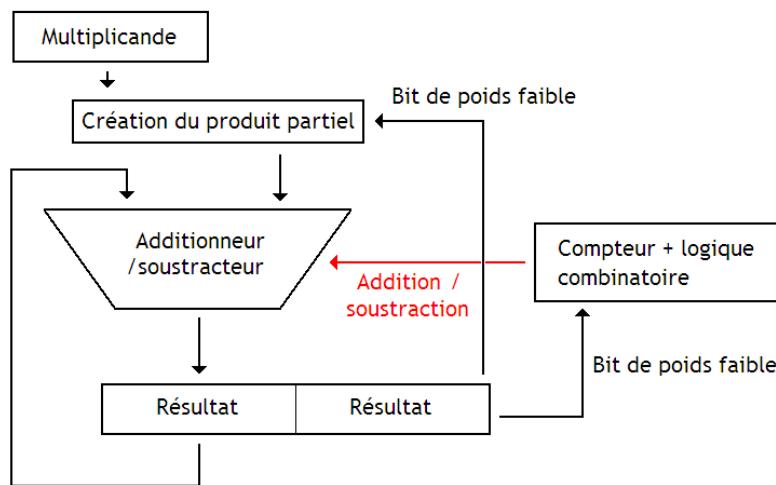
Multiplicateurs diviser pour régner

Il existe enfin un tout dernier type de multiplicateurs : les multiplicateurs diviser pour régner. Pour comprendre le principe, nous allons prendre un multiplicateur qui multiplie deux nombres de 32 bits. Les deux opérandes A et B peuvent être décomposées en deux morceaux de 16 bits, qu'il suffit de multiplier entre eux pour obtenir les produits partiels voulus : une seule multiplication 32 bits se transforme en quatre multiplications d'opérandes de 16 bits. En clair, ces multiplicateurs sont composés de multiplicateurs qui travaillent sur des opérandes plus petites, associés à des additionneurs.

Multiplication signée

Tous les circuits qu'on a vus plus haut sont capables de multiplier des nombres entiers positifs, mais on peut les adapter pour qu'ils fassent des calculs sur des entiers signés. Pour les entiers en signe-valeur absolue, il suffit de multiplier les valeurs absolues et de déduire le signe du résultat avec un vulgaire XOR entre les bits de signe des nombres à multiplier. Dans ce qui va suivre, nous allons nous intéresser à la multiplication signée en complément à deux.

Les multiplicateurs vus plus haut fonctionnent parfaitement quand les deux opérandes ont le même signe, mais pas quand un des deux opérandes est négatif. Avec un multiplicande négatif, le produit partiel est censé être négatif. Mais dans les multiplicateurs vus plus haut, les bits inconnus du produit partiel sont remplis avec des zéros, et donc positifs. Pour résoudre ce problème, il suffit d'utiliser une extension de signe sur les produits partiels. Pour cela, il faut faire en sorte que le décalage du résultat soit un décalage arithmétique. Pour traiter les multiplicateurs négatifs, on ne doit pas ajouter le produit partiel, mais le soustraire (l'explication du pourquoi est assez dure à comprendre, aussi je vous épargne les détails). L'additionneur doit donc être remplacé par un additionneur-soustracteur.



Multiplieurs de Booth

Il existe une autre façon, nettement plus élégante, inventée par un chercheur en cristallographie du nom de Booth : l'**algorithme de Booth**. Le principe de cet algorithme est que des suites de bits à 1 consécutifs dans l'écriture binaire d'un nombre entier peuvent donner lieu à des simplifications. Si vous vous rappelez, les nombres de la forme 01111...111 sont des nombres qui valent $2n - 1$. Donc, $X \times (2^n - 1) = (X \times 2^n) - X$. Cela se calcule avec un décalage (multiplication par 2^n) et une soustraction. Ce principe peut s'appliquer aux suites de 1 consécutifs dans un nombre entier, avec quelques modifications. Prenons un nombre composé d'une suite de 1 qui commence au n-ième bit, et qui termine au X-ième bit : celle-ci forme un nombre qui vaut $2^n - 2^{n-x}$. Par exemple, 0011 1100 = 0011 1111 - 0000 0011, ce qui donne $(2^7 - 1) - (2^2 - 1)$. Au lieu de faire des séries d'additions de produits partiels et de décalages, on peut remplacer le tout par des décalages et des soustractions.

C'est le principe qui se cache derrière l'algorithme de Booth : il regarde le bit du multiplicateur à traiter et celui qui précède, pour déterminer s'il faut soustraire, additionner, ou ne rien faire. Si les deux bits valent zéro, alors pas besoin de soustraire : le produit partiel vaut zéro. Si les deux bits valent 1, alors c'est que l'on est au beau milieu d'une suite de 1 consécutifs, et qu'il n'y a pas besoin de soustraire. Par contre, si ces deux bits valent 01 ou 10, alors on est au bord d'une suite de 1 consécutifs, et l'on doit soustraire ou additionner. Si les deux bits valent 10 alors c'est qu'on est au début d'une suite de 1 consécutifs : on doit soustraire le multiplicande multiplié par 2^{n-x} . Si les deux bits valent 01, alors on est à la fin d'une suite de bits, et on doit additionner le multiplicande multiplié par 2^n . On peut remarquer que si le registre utilisé pour le résultat décale vers la droite, il n'y a pas besoin de faire la multiplication par la puissance de deux : se contenter d'ajouter ou de soustraire le multiplicande suffit.

Reste qu'il y a un problème pour le bit de poids faible : quel est le bit précédent ? Pour cela, le multiplicateur est stocké dans un registre qui contient un bit de plus qu'il n'en faut. On remarque que pour obtenir un bon résultat, ce bit précédent doit être mis à 0. Le multiplicateur est placé dans les bits de poids fort, tandis que le bit de poids faible est mis à zéro. Cet algorithme gère les signes convenablement. Le cas où le multiplicande est négatif est géré par le fait que le registre du résultat subit un décalage arithmétique vers la droite à chaque cycle. La gestion du multiplicateur négatif est plus complexe à comprendre mathématiquement, mais je peux vous certifier que cet algorithme gère convenablement ce cas.

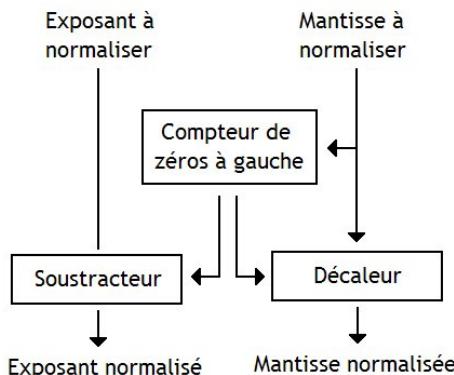
Les opérations sur les nombres flottants

Maintenant, nous allons voir dans les grandes lignes comment fonctionnent les circuits capables de calculer avec des nombres flottants. Ce chapitre ne traite que des nombres flottants de la norme IEEE754 : nous ne traiterons pas des flottants logarithmiques ou d'autres formats de flottants exotiques.

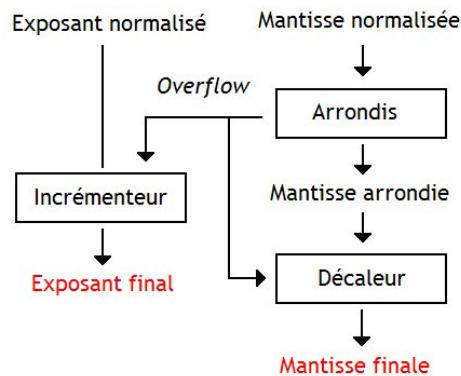
Normalisation et arrondis

Calculer sur des nombres flottants peut sembler trivial. Mais le résultat du calcul devra subir quelques transformations pour être un nombre flottant : on peut citer les arrondis, mais aussi d'autres étapes dites de normalisation.

La **prénormalisation** gère le bit implicite. Lorsqu'un circuit de calcul fournit son résultat, celui-ci n'a pas forcément son bit implicite à 1. On est obligé de décaler la mantisse du résultat de façon à ce que le bit implicite soit un 1. Pour savoir de combien de rangs il faut décaler, il faut compter le nombre de zéros situés avant le 1 de poids fort, avec un circuit spécialisé. Ce circuit permet aussi de détecter si la mantisse vaut zéro. Mais si on décale notre résultat de n rangs, cela signifie qu'on le multiplie par 2 à la puissance n. Pour régler ce problème, il faut corriger l'exposant du résultat pour annuler la multiplication par 2 à la puissance n. Il suffit pour cela de lui soustraire n, le nombre de rangs dont on a décalé la mantisse.



Une fois ce résultat calculé, il faut faire un arrondi du résultat avec un circuit de **normalisation**. Malheureusement, il arrive que ces arrondis décalent la position du bit implicite d'un rang, ce qui se résout avec un décalage si cela arrive. Le circuit de normalisation contient donc de quoi détecter ces débordements et un décaleur. Bien évidemment, l'exposant doit alors lui aussi être corrigé en cas de décalage de la mantisse.



Multiplication et division

Prenons deux nombres flottants de mantes m₁ et m₂ et les exposants e₁ et e₂. Leur multiplication donne :

$$(m_1 \times 2^{e_1}) \times (m_2 \times 2^{e_2})$$

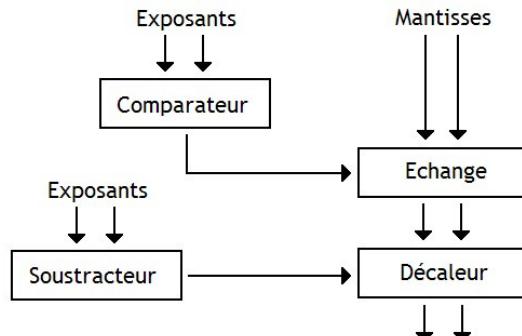
$$(m_1 \times m_2) \times (2^{e_1} \times 2^{e_2})$$

$$(m_1 \times m_2) \times 2^{e_1+e_2}$$

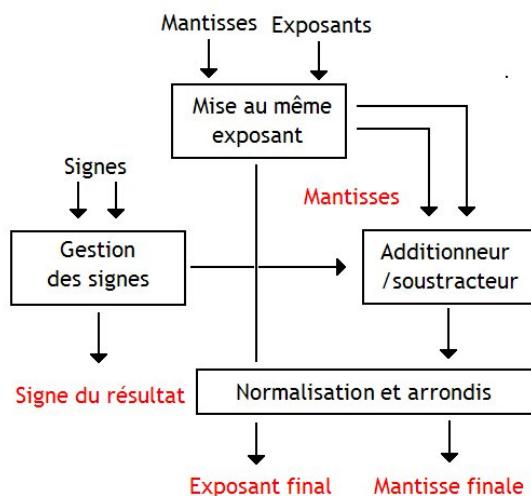
En clair, multiplier deux flottants revient à multiplier les mantes et additionner les exposants. C'est la même chose pour la division, à part que les exposants sont soustraits et que les mantes sont divisées. Il faut cependant penser à ajouter les bits implicites aux mantes avant de les multiplier. De plus, il faut aussi gérer le fait que les exposants sont codés en représentation par excès en retirant le biais de l'exposant calculé.

Addition et soustraction

La somme de deux flottants n'est simplifiable que quand les exposants sont égaux : dans ce cas, il suffit d'ajouter les mantes. Il faut donc mettre les deux flottants au même exposant, l'exposant choisi étant souvent le plus grand exposant des deux flottants. Convertir le nombre dont l'exposant est le plus petit demande de décaler la mantisse vers la droite, d'un nombre de rangs égal à la différence entre les deux exposants. Pour comprendre pourquoi, il faut se souvenir que décaler vers la droite diminuer l'exposant du nombre de rangs. Pour faire ce décalage, on utilise un décaleur et un circuit qui échange les deux opérandes (histoire d'envoyer le plus petit exposant dans le décaleur). Ce circuit d'échange est piloté par un comparateur qui détermine quel est le nombre avec le plus petit exposant.



Suivant les signes, il faudra additionner ou soustraire les opérandes.

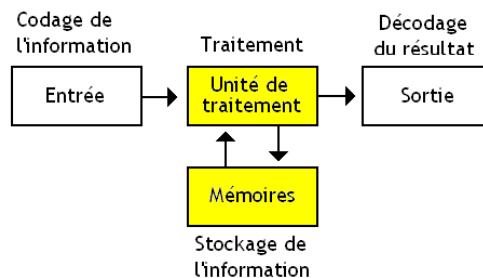


Architecture d'un ordinateur

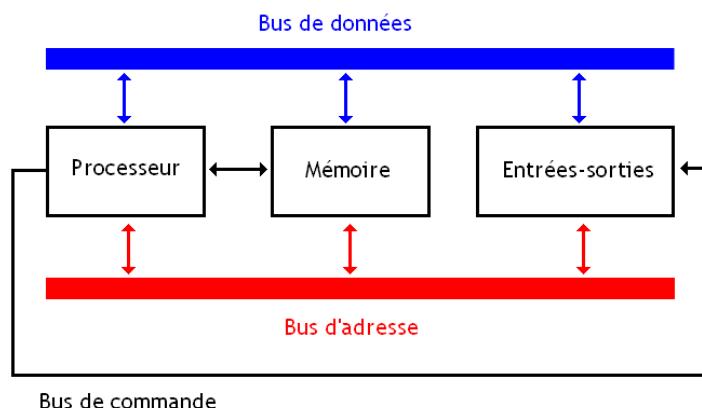
Dans les chapitres précédents, nous avons vu comment représenter de l'information, la traiter et la mémoriser avec des circuits. Mais un ordinateur n'est pas qu'un amoncellement de circuits et est organisé d'une manière bien précise. Il est structuré autour de trois grandes catégories de circuits :

- les **entrées/sorties**, qui permettent à l'ordinateur de communiquer avec l'extérieur ;
- une **mémoire** qui mémorise les données à manipuler ;
- un **processeur**, qui manipule l'information et donne un résultat.

Pour faire simple, le processeur est un circuit qui s'occupe de faire des calculs et de traiter des informations. La mémoire s'occupe purement de la mémorisation des informations. Les entrées-sorties permettent au processeur et à la mémoire de communiquer avec l'extérieur et d'échanger des informations avec des périphériques. Tout ce qui n'appartient pas à la liste du dessus est obligatoirement connecté sur les ports d'entrée-sortie et est appelé **périphérique**.



Ces composants communiquent via un **bus**, un ensemble de fils électriques qui relie les différents éléments d'un ordinateur. Ce bus est souvent divisé en deux à trois bus spécialisés : le bus de commande, le bus d'adresse, et le bus de données. Le **bus de données** est un ensemble de fils par lequel s'échangent les données entre les composants. Le **bus de commande** permet au processeur de configurer la mémoire et les entrées-sorties. Le **bus d'adresse**, facultatif, permet au processeur de sélectionner l'entrée, la sortie ou la portion de mémoire avec qui il veut échanger des données.



Parfois, on décide de regrouper la mémoire, les bus, le CPU et les ports d'entrée-sortie dans un seul composant électronique nommé **microcontrôleur**. On trouve des microcontrôleurs dans les disques durs, les baladeurs mp3, dans les automobiles, etc.

Les entrées-sorties

Tous les circuits vus précédemment sont des circuits qui se chargent de traiter des données codées en binaire. Ceci dit, les données ne sortent pas de n'importe où : l'ordinateur contient des composants électroniques qui se chargent de traduire des informations venant de l'extérieur en nombres. Ces composants sont ce qu'on appelle des **entrées**. Par exemple, le clavier est une entrée : l'électronique du clavier attribue un nombre entier (scancode) à une touche, nombre qui sera communiqué à l'ordinateur lors de l'appui d'une touche. Pareil pour la souris : quand vous bougez la souris, celle-ci envoie des informations sur la position ou le mouvement du curseur, informations qui sont codées sous la forme de nombres. La carte son évoquée il y a quelques chapitres est bien sûr une entrée : elle est capable d'enregistrer un son, et de le restituer sous la forme de nombres.

Si il y a des entrées, on trouve aussi des **sorties**, des composants électroniques qui transforment des nombres présents dans l'ordinateur en quelque chose d'utile. Ces sorties effectuent la traduction inverse de celle faite par les entrées : si les entrées convertissent une information en nombre, les sorties font l'inverse : là où les entrées encodent, les sorties décoden. Par exemple, un écran LCD est un circuit de sortie : il reçoit des informations, et les transforme en image affichée à l'écran. Même chose pour une imprimante : elle reçoit des documents texte encodés sous forme de nombres, et se permet de les imprimer sur du papier. Et la carte son est aussi une sortie, vu qu'elle transforme les sons d'un fichier audio en tensions destinées à un haut-parleur : c'est à la fois une entrée, et une sortie.

Le processeur

L'**unité de traitement** est un circuit qui s'occupe de faire des calculs et de manipuler l'information provenant des entrées-sorties ou récupérée dans la mémoire. Dans les circuits vus auparavant, le traitement effectué par l'unité de traitement est toujours le même et dépend uniquement du câblage de celle-ci : impossible de faire faire autre chose à l'unité de traitement que ce pour quoi elle a été conçue. Par exemple, un circuit conçu pour additionner des nombres ne pourra pas faire autre chose. On peut parfois reconfigurer le circuit, pour faire varier certains paramètres via des interrupteurs ou des boutons, mais cela s'arrête là : l'unité de traitement n'est pas programmable. Et cela pose un problème : à chaque problème qu'on veut résoudre en utilisant un automate, on doit recréer un nouveau circuit.

Mais sur certains circuits, on peut remplacer le traitement effectué sans modifier leur câblage. On peut donc faire ce que l'on veut de ces circuits : ceux-ci sont réutilisables à volonté et il est possible de modifier leur fonction du jour au lendemain. On dit qu'ils sont programmables. Pour rendre un circuit programmable, il faut impérativement modifier son unité de traitement, qui devient ce qu'on appelle un **processeur**, ou **Central Processing Unit**, abrégé en CPU.

Tout processeur est conçu pour effectuer un nombre limité d'opérations bien précise. Ces opérations sont des traitements élémentaires auquel on donne le nom d'**instructions**. Ce sont des calculs, des échanges de données avec la mémoire, etc. Un grand nombre d'instructions servent à faire des calculs : un ordinateur peut ainsi additionner deux nombres, les soustraire, les multiplier, les diviser, etc. Les instructions de test peuvent comparer deux nombres entre eux pour vérifier si deux nombres sont égaux ou non, et agir en fonction. D'autres instructions permettent d'échanger des informations entre la mémoire et le processeur.

Pour modifier le comportement de ce processeur, il suffit de créer une suite d'instructions qui va faire ce que l'on souhaite : cette suite d'instructions s'appelle un **programme**. La totalité des logiciels présents sur un ordinateur sont des programmes comme les autres. Tout processeur est conçu pour exécuter cette suite d'instructions dans l'ordre demandé. Ce programme est stocké dans la mémoire de l'ordinateur, ce qui fait qu'on peut le modifier, rendant l'ordinateur programmable.

La mémoire

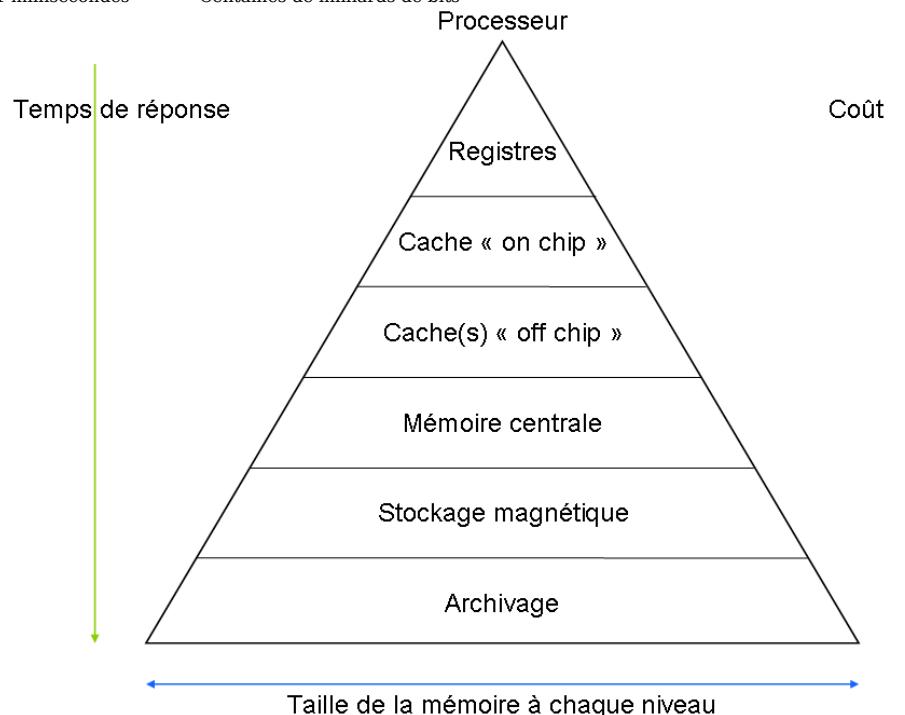
La mémoire est le composant qui mémorise des informations, des données. Bien évidemment, une mémoire ne peut stocker qu'une quantité finie de données. Et à ce petit jeu, certaines mémoires s'en sortent mieux que d'autres et peuvent stocker beaucoup plus de données que les autres. La **capacité** d'une mémoire correspond à la quantité d'informations que celle-ci peut mémoriser. Plus précisément, il s'agit du nombre de bits que celle-ci peut contenir. Dans la majorité des cas, la mémoire est découpée en plusieurs **bytes**, des blocs de mémoire qui contiennent chacun un nombre fini et constant de bits. Le plus souvent, ces bytes sont composés de plusieurs groupes de 8 bits, appelés des **octets**.

Outre leurs capacités respectives, tous les mémoires ne sont pas aussi rapides. La rapidité d'une mémoire se mesure grâce à deux paramètres :

- Le **temps de latence** correspond au temps qu'il faut pour effectuer une lecture ou une écriture : plus celui-ci est bas, plus la mémoire sera rapide.
- Le **débit mémoire** correspond à la quantité d'informations qui peut être récupéré ou enregistré en une seconde dans la mémoire : plus celui-ci est élevé, plus la mémoire sera rapide.

La lenteur d'une mémoire dépend de sa capacité : plus la capacité est importante, plus la mémoire est lente. Le fait est que si l'on souhaitait utiliser une seule grosse mémoire dans notre ordinateur, celle-ci serait trop lente et l'ordinateur serait inutilisable. Pour résoudre ce problème, il suffit d'utiliser plusieurs mémoires de taille et de vitesse différentes, qu'on utilise suivant les besoins. Des mémoires très rapides de faible capacité seconderont des mémoires lentes de capacité importante. On peut regrouper ces mémoires en niveaux : toutes les mémoires appartenant à un même niveau ont grosso modo la même vitesse. Pour simplifier, il existe quatre grandes niveaux de hiérarchie mémoire, indiqués dans le tableau ci-dessous.

Type de mémoire	Temps d'accès	Capacité
Registres	1 nanoseconde	Entre 1 à 512 bits
Caches	10 - 100 nanosecondes	Milliers ou millions de bits
Mémoire RAM	1 microseconde	Milliards de bits
Mémoires de masse	1 milliseconde	Centaines de milliards de bits



Le but de cette organisation est de placer les données accédées souvent, ou qui ont de bonnes chances d'être accédées dans le futur, dans une mémoire qui soit la plus rapide possible. Le tout est faire en sorte de placer les données intelligemment, et les répartir correctement dans cette hiérarchie des mémoires. Ce placement se base sur deux principes qu'on appelle les principes de localité spatiale et temporelle :

- un programme a tendance à réutiliser les instructions et données accédées dans le passé : c'est la **localité temporelle** ;
- et un programme qui s'exécute sur un processeur a tendance à utiliser des instructions et des données consécutives, qui sont proches, c'est la **localité spatiale**.

On peut exploiter ces deux principes pour placer les données dans la bonne mémoire. Par exemple, si on a accédé à une donnée récemment, il vaut mieux la copier dans une mémoire plus rapide, histoire d'y accéder rapidement les prochaines fois : on profite de la localité temporelle. On peut aussi profiter de la localité spatiale : si on accède à une donnée, autant précharger aussi les données juste à côté, au cas où elles seraient accédées. Ce placement des données dans la bonne mémoire peut être géré par le matériel de notre ordinateur, mais aussi par le programmeur.

Mémoires de masse

Les **mémoires de masse** servent à stocker de façon permanente des données ou des programmes qui ne doivent pas être effacés (on dit qu'il s'agit de mémoires non-volatiles). Les mémoires de masse servent toujours à stocker de grosses quantités de données : elles ont une capacité énorme comparée aux autres types de mémoires, et sont donc très lentes. Parmi ces mémoires de masse, on trouve notamment :

- les disques durs ;
- les mémoires Flash, utilisées dans les clés USB, voire dans les disques durs SSD ;
- les disques optiques, comme les CD-ROM, DVD-ROM, et autres CD du genre ;
- les fameuses disquettes, totalement obsolètes de nos jours ;
- mais aussi quelques mémoires très anciennes et rarement utilisées de nos jours, comme les rubans perforés et quelques autres.

Mémoire principale

La mémoire principale est appelée, par abus de langage, **la mémoire RAM**. Il s'agit d'une mémoire qui stocke temporairement des données que le processeur doit manipuler (on dit qu'elle est volatile). Elle sert donc essentiellement pour maintenir des résultats de calculs, contenir temporairement des programmes à exécuter, etc. Avec cette mémoire, chaque donnée ou information stockée se voit attribuer un nombre binaire unique, l'**adresse mémoire**, qui va permettre de la sélectionner et de l'identifier parmi toutes les autres. On peut comparer une adresse à un numéro de téléphone ou à une adresse d'appartement : chaque correspondant à un numéro de téléphone et vous savez que pour appeler telle personne, vous devez composer tel numéro. Les adresses mémoires, c'est pareil, mais avec des données.

Caches et local stores

Le troisième niveau est intermédiaire entre les registres et la mémoire principale. Dans la majorité des cas, la mémoire intercalée entre les registres et la mémoire RAM/ROM est ce qu'on appelle une **mémoire cache**. Celle-ci a quelques particularités qui la rendent vraiment différente d'une mémoire RAM ou ROM. Premièrement, et aussi bizarre que cela puisse paraître, elle n'est jamais adressable ! Ensuite, le contenu du cache est géré par un circuit qui s'occupe des échanges avec les registres et la mémoire principale : le programmeur ne peut pas gérer directement ce cache. Si cela peut paraître contre-intuitif, tout s'éclairera dans le chapitre dédié à ces mémoires. De nos jours, ce cache est intégré dans le processeur.

Sur certains processeurs, les mémoires caches sont remplacées par des mémoires RAM appelées des **local stores**. Ce sont des mémoires RAM, identiques à la mémoire RAM principale, mais qui sont plus petites et plus rapides. Contrairement aux mémoires caches, il s'agit de mémoires adressables, ce qui fait qu'elles ne sont plus gérées automatiquement par le processeur : c'est le programme en cours d'exécution qui prend en charge les transferts de données entre local store et mémoire RAM. Ces local stores consomment moins d'énergie que les caches à taille équivalente : en effet, ceux-ci n'ont pas besoin de circuits compliqués pour les gérer automatiquement, contrairement aux caches. Côté inconvénients, ces local stores peuvent entraîner des problèmes de compatibilité : un programme conçu pour fonctionner avec des local stores ne fonctionnera pas sur un ordinateur qui en est dépourvu.

Registres

Enfin, le dernier niveau de hiérarchie mémoire est celui des **registres**, de petites mémoires très rapides et de faible capacité. Celles-ci sont intégrées à l'intérieur du processeur. La capacité des registres dépend fortement du processeur. Au tout début de l'informatique, il n'était pas rare de voir des registres de 3, 4, voire 8 bits. Par la suite, la taille de ces registres a augmenté, passant rapidement de 16 à 32 bits, voire 48 bits sur certaines processeurs spécialisés. De nos jours, les processeurs de nos PC utilisent des registres de 64 bits. Il existe toujours des processeurs de faible performance qui utilisent des registres relativement petits, de 8 à 16 bits.

Certains processeurs disposent de **registres spécialisés**, dont la fonction est prédéterminée une bonne fois pour toutes : un registre est conçu pour stocker, uniquement des nombres entiers, ou seulement des flottants, quand d'autres sont spécialement dédiés aux adresses mémoires. Par exemple, les processeurs présents dans nos PC séparent les registres entiers des registres flottants. Pour plus de flexibilité, certains processeurs remplacent les registres spécialisés par des **registres généraux**, utilisables pour tout et n'importe quoi. Pour reprendre notre exemple du dessus, un processeur peut fournir 12 registres généraux, qui peuvent stocker 12 entiers, ou 10 entiers et 2 flottants, ou 7 adresses et 5 entiers, etc. Dans la réalité, les processeurs utilisent à la fois des registres généraux et quelques registres spécialisés.

Performance et consommation d'énergie

Dans ce qui précède, nous avons abordé divers paramètres, qui sont souvent indiqués lorsque vous achetez un processeur, de la mémoire, ou tout autre composant électronique :

- la finesse de gravure ;
- le nombre de transistors ;
- la fréquence ;
- et la tension d'alimentation.

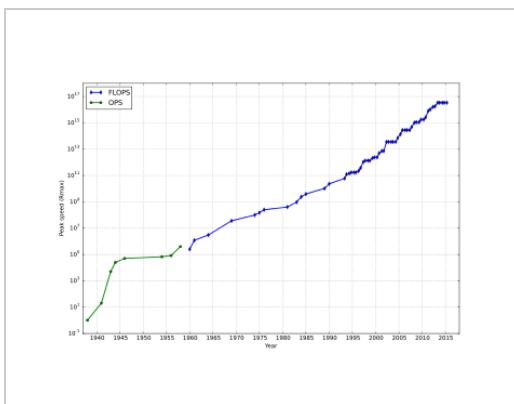
Ces paramètres ont une influence indirecte sur les performances d'un ordinateur, ou sur sa consommation énergétique. Dans ce qui va suivre, nous allons voir comment déduire la performance d'un processeur ou d'une mémoire suivant ces paramètres. Nous allons aussi étudier quel est l'impact de la finesse de gravure sur la consommation d'énergie d'un composant électronique. Cela nous amènera à étudier les tendances de l'industrie.

Performances

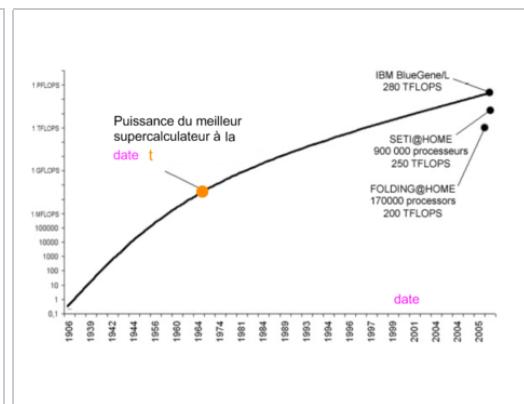
Pour commencer, il faut d'abord définir ce qu'est la performance d'un ordinateur. C'est loin d'être une chose triviale : de nombreux paramètres font qu'un ordinateur sera plus rapide qu'un autre. De plus, la performance ne signifie pas la même chose selon le composant dont on parle. La performance d'un processeur n'est ainsi pas comparable à la performance d'une mémoire ou d'un périphérique. La finesse de gravure n'a pas d'impact en elle-même sur la performance ou la consommation d'énergie : elle permet juste d'augmenter le nombre de transistors et la fréquence, tout en diminuant la tension d'alimentation. Et ce sont ces trois paramètres qui vont nous intéresser.

Processeur

Pour un processeur, cette performance correspondra à la **puissance de calcul**, à savoir le nombre de calculs que l'ordinateur peut faire par secondes. Cette puissance de calcul se mesure en MIPS : Million Instructions Per Second, (million de calcul par seconde en français). Plus un processeur a un MIPS élevé, plus il sera rapide : un processeur avec un faible MIPS mettra plus de temps à faire une même quantité de calcul qu'un processeur avec un fort MIPS. Le MIPS est surtout utilisé comme estimation de la puissance de calcul sur des nombres entiers. Mais il existe cependant une mesure annexe, utilisée pour la puissance de calcul sur les nombres flottants. Il s'agit du FLOPS, à savoir le nombre d'opérations flottantes par seconde. Qu'il s'agisse des FLOPS ou des MIPS, on observe que la puissance de calcul augmente du fil du temps.



Évolution de la puissance de calcul en FLOPS des processeurs commerciaux.



Évolution de la puissance de calcul en FLOPS des supercalculateurs.

MIPS et FLOPS dépendent de la fréquence : plus la fréquence est élevée, plus le processeur sera rapide. Mais la relation entre fréquence et puissance de calcul dépend fortement du processeur. Par exemple, deux processeurs différents peuvent avoir des puissances de calcul très différentes avec des fréquences identiques. Cela vient du fait que chaque opération prend un certain temps, un certain nombre de cycles d'horloge. Par exemple, sur les processeurs modernes, une addition va prendre un cycle d'horloge, une multiplication entre 1 et 2 cycles, etc. Cela dépend du processeur, de l'opération, et d'autres paramètres assez compliqués. Mais on peut calculer un nombre moyen de cycle d'horloge par opération : le **CPI (Cycle Per Instruction)**.

On peut reformuler le tout en utilisant non pas le CPI, mais son inverse : le nombre de calculs qui sont effectués par cycle d'horloge. Celui-ci porte le doux nom d'**IPC (Instruction Per Cycle)**. Celui-ci a plus de sens sur les processeurs actuels, qui peuvent effectuer plusieurs calculs en même temps, dans des circuits différents (des unités de calcul différentes, pour être précis). Sur ces ordinateurs, l'IPC est supérieur à 1. La puissance de calcul est égale à la fréquence multipliée par l'IPC. En clair, plus la fréquence ou l'IPC sont élevés, plus le processeur sera rapide. La fréquence est généralement une information qui est mentionnée lors de l'achat d'un processeur. Malheureusement, l'IPC ne l'est pas. La raison vient du fait que la mesure de l'IPC n'est pas normalisée et qu'il existe de très nombreuses façons de le mesurer. Il faut dire que l'IPC varie énormément suivant les opérations, le programme, diverses optimisations matérielles, etc.

$$\text{MIPS} = \text{IPC} \times f \times 10^6$$

La vitesse des transistors est proportionnelle à la finesse de gravure. En conséquence, elle augmente de 40 % tous les deux ans (elle est multipliée par la racine carrée de deux). Pour cette raison, la fréquence devrait augmenter au même rythme. Cependant, la fréquence dépend aussi de la rapidité du courant dans les interconnexions (les fils) qui relient les transistors, celles-ci devenant de plus en plus un facteur limitant pour la fréquence. Outre l'augmentation des fréquences, diverses optimisations permettent d'augmenter l'IPC. La loi de Pollack dit que l'augmentation de l'IPC d'un processeur est approximativement proportionnelle à la racine carrée du nombre de transistors ajoutés : si on double le nombre de transistors, la performance est multipliée par la racine carrée de 2. En utilisant la loi de Moore, on en déduit qu'on gagne approximativement 40% d'IPC tous les deux ans, à ajouter aux 40 % d'augmentation de fréquence.

On peut expliquer cette loi de Pollack assez simplement. Il faut se rappeler que les processeurs modernes peuvent changer l'ordre des instructions pour gagner en performances (on parle d'exécution dans le désordre). Pour cela, les instructions sont pré-chargées dans une mémoire tampon, de taille fixe. Cependant, le processeur doit gérer les situations où une instruction a besoin du résultat d'une autre pour s'exécuter : chaque instruction doit être comparée à toutes les autres, pour savoir quelle instruction a besoin des résultats d'une autre. Avec N instructions, vu que chacune d'entre elles doit être comparée à toutes les autres, ce qui demande N^2 comparaisons. En doublant le nombre de transistors, on peut donc doubler le nombre de comparateurs, ce qui signifie que l'on peut multiplier le nombre d'instructions par la racine carrée de deux.

On peut cependant contourner la loi de Pollack, qui ne vaut que pour un seul processeur. Mais en utilisant plusieurs processeurs, la performance est la somme des performances individuelles de chacun d'entre eux. C'est pour cela que les processeurs actuels sont double, voire quadruple coeurs : ce sont simplement des circuits imprimés qui contiennent deux, quatre, voire 8 processeurs différentes, placés sur la même puce. Chaque

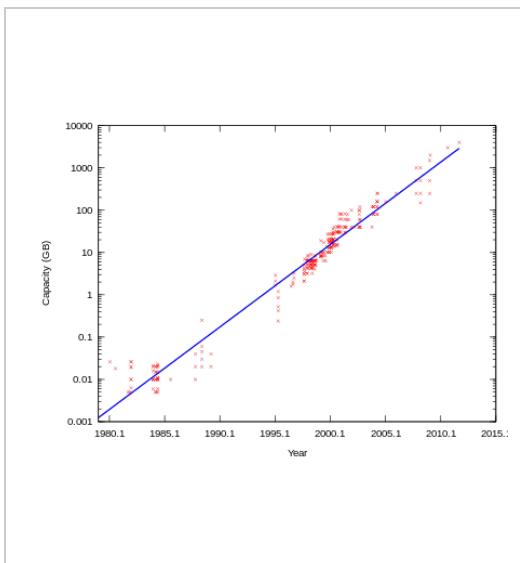
coeur correspond à un processeur. En faisant ainsi, doubler le nombre de transistors permet de doubler le nombre de coeurs et donc de doubler la performance, ce qui est mieux qu'une amélioration de 40%.

Mémoire

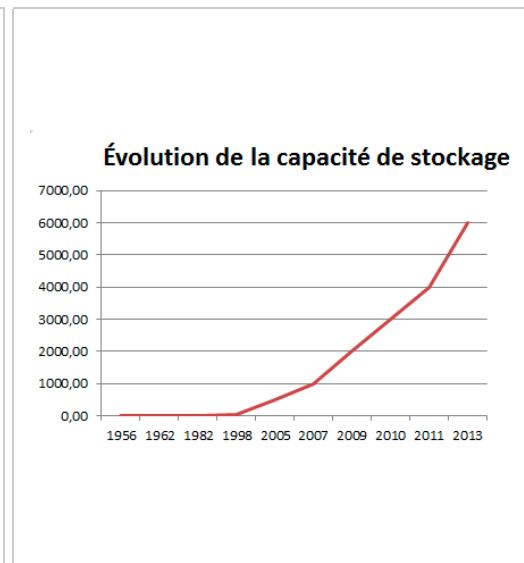
Pour la mémoire, la performance se mesure différemment. Pour une mémoire, la performance correspond à trois paramètres : sa vitesse, son débit et la quantité d'informations qu'elle peut mémoriser.

- La **capacité** d'une mémoire correspond à la quantité d'informations que celle-ci peut mémoriser. Plus précisément, il s'agit du nombre de bits que celle-ci peut contenir. Elle se mesure donc en bits, ou en octets (des groupes de 8 bits). Nous reviendrons sur le sujet dans le chapitre sur la mémoire.
- La vitesse d'une mémoire correspond au temps qu'il faut pour récupérer une information dans la mémoire, ou pour y effectuer un enregistrement. On parle aussi de **temps de latence**. Plus celui-ci est bas, plus la mémoire sera rapide. Celui-ci se mesure en secondes, millisecondes, microsecondes pour les mémoires les plus rapides. Dans la réalité, tous les accès à la mémoire ne sont pas égaux, et ceux-ci ont des latences différentes. Par exemple, les lectures sont plus rapides que les enregistrements/écritures. Ces latences peuvent se déduire de temps de latences plus élémentaires, qui sont appelés les **timings mémoire**.
- Le **débit** correspond à la quantité d'informations qui peut être récupéré ou enregistré en une seconde dans la mémoire. Plus celui-ci est élevé, plus la mémoire sera rapide. Elle a pour unité une capacité mémoire divisée par une seconde.

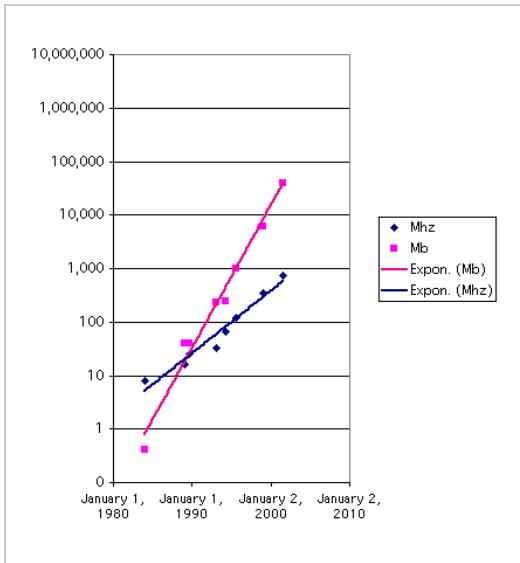
Ces paramètres varient suivant le type de mémoires. De manière générale, les mémoires électroniques sont plus rapides que les mémoires magnétiques ou optiques, mais ont une capacité inférieure. La loi de Moore a une influence certaine sur la capacité des mémoires électroniques. En effet, une mémoire électronique est composée de bascules de 1 bits, elles-mêmes composées de transistors : plus la finesse de gravure est petite, plus la taille d'une bascule sera faible. Quand le nombre de transistors d'une mémoire double, on peut considérer que le nombre de bascules, et donc la capacité double. D'après la loi de Moore, cela arrive tous les deux ans, ce qui est bien ce qu'on observe pour les mémoires RAM. Par contre, les mémoires magnétiques, comme les disques durs, augmentent à un rythme différent, celles-ci n'étant pas composées de transistors.



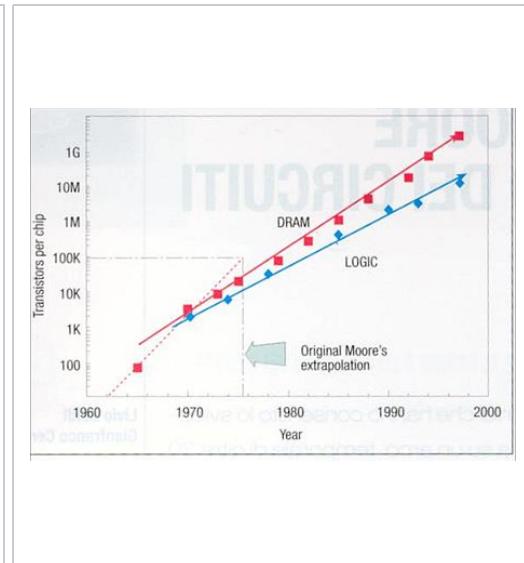
Évolution de la capacité des disques durs (mémoires magnétiques) dans le temps en échelle logarithmique.



Évolution de la capacité des disques durs (mémoires magnétiques) dans le temps, en échelle linéaire (normale).



Comparaison entre la croissance de la capacité des disques durs dans le temps comparé à la loi de Moore.



Evolution du nombre de transistors d'une mémoire électronique au cours du temps. On voit que celle-ci suit de près la loi de Moore.



Evolution du prix et de la capacité moyenne des mémoires RAM (mémoires électroniques).

L'évolution de la fréquence des mémoires suit plus ou moins celle des processeurs, elle double au même rythme. Mais malheureusement, cette fréquence reste inférieure à celle des processeurs. Cette augmentation de fréquence permet au débit des mémoire d'augmenter avec le temps. En effet, à chaque cycle d'horloge, la mémoire peut envoyer ou recevoir une quantité fixe de donnée. En multipliant cette largeur du bus par la fréquence, on obtient le débit. Par contre, la fréquence n'a aucun impact sur le temps de latence.

Entrées-sorties

Pour une entrée-sortie, la mesure de la performance dépend énormément du composant en question. On ne mesure pas la vitesse d'un disque dur de la même manière que celle d'un écran. Par exemple, le nombre de plateaux d'un disque dur n'a pas d'équivalent pour un écran. Divers cours seront consacrés à chaque périphérique, aussi nous aborderons plus en détail les mesures de performances des différents périphériques dans ceux-ci. Cependant, on peut donner le principe de base pour les mémoires de masse.

Le débit d'une mémoire de masse ou d'un périphérique est le produit de deux facteurs : le nombre d'**opérations d'entrée-sortie par secondes**, et la taille des données échangée par opération. Ces opérations d'entrée-sortie correspondent chacune à un échange de données entre le périphérique et le reste de l'ordinateur, via le bus. Cela peut se résumer par la formule suivante, avec D le débit du HDD, IOPS le nombre d'opérations disque par secondes, et T la quantité de données échangée lors d'une opération d'entrée-sortie : $D = \text{IOPS} \times T$. Augmenter la taille des données transmises augmente donc le débit, mais cette technique est rarement utilisée. Les bus ne sont extensibles à l'infini. A la place, la majorité des périphériques incorpore diverses optimisations pour augmenter l'IOPS, ce qui permet un meilleur débit pour une taille T identique.

Consommation énergétique

Pour terminer, il nous faut évoquer la consommation énergétique. Nos ordinateurs consomment une certaine quantité de courant pour fonctionner, et donc une certaine quantité d'énergie. Il se trouve que la plupart de cette énergie finit par être dissipée sous forme de chaleur : plus un composant consomme d'énergie, plus il chauffe. La chaleur dissipée est mesurée par ce qu'on appelle l'**enveloppe thermique**, ou TDP (Thermal Design Power), mesurée en Watts.

Dans les grandes lignes, l'efficacité énergétique des processeurs n'a pas cessé d'augmenter au cours du temps : pour le même travail, les processeurs chauffent moins. Avant les années 2000, 2010, la quantité de calcul que peut effectuer le processeur en dépensant un watt double tous les 1,57 ans. Cette observation porte le nom de **loi de Kommeij**. Mais de nos jours, cette loi n'est plus tellement respectée, pour diverses raisons.

Pour comprendre pourquoi, on doit faire quelques précisions. Une partie de la consommation d'énergie d'un processeur vient du fait que nos transistors consomment de l'énergie en changeant d'état : on appelle cette perte la **consommation dynamique**. Une autre partie vient du fait que les transistors ne sont pas des dispositifs parfaits et qu'ils laissent passer un peu de courant entre la grille et le drain (y compris dans l'état fermé). Cette consommation est appelée la **consommation statique**. Et l'évolution de la finesse de gravure a des effets différents sur chaque type de consommation : si elle diminue la consommation dynamique, elle augmente la consommation statique. D'où la fin de la loi de Kommeij, qui ne valait que tant que la consommation dynamique était la plus importante des deux. Mais voyons cela dans le détail.

Consommation dynamique

Commençons par évaluer la consommation dynamique en fonction de la finesse de gravure. Pour l'évaluer, il faut se rappeler d'une chose : un transistor est composé de deux morceaux de conducteurs (l'armature de la grille et la liaison drain-source) séparés par un isolant. Dit autrement, c'est une sorte de condensateur, à savoir un composant capable d'accumuler des charges électriques (ici, des électrons). La quantité d'énergie que peut stocker un tel condensateur est égale au produit ci-dessous, avec C un coefficient de proportionnalité appelé la capacité du condensateur et U la tension d'alimentation. Cette équation nous dit que la consommation d'énergie d'un transistor dépend du carré de la tension d'alimentation et de sa capacité. Ainsi, plus la tension d'alimentation est élevée, plus le composant consomme d'énergie.

$$E = \frac{1}{2} CU^2$$

Cette énergie est dissipée quand le transistor change d'état, il dissipe une quantité de chaleur proportionnelle au carré de la tension d'alimentation. Or, la fréquence définit le nombre maximal de changements d'état qu'un transistor peut subir en une seconde : pour une fréquence de 50 hertz, un transistor peut changer d'état 50 fois par seconde maximum. Pour un seul transistor, l'énergie dissipée en une seconde est donc calculable de la sorte :

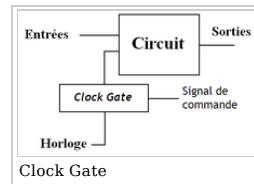
$$E = \frac{1}{2} CU^2 \times f$$

Et si on prend en compte le fait qu'un processeur contient un nombre n de transistors, on obtient alors que la consommation énergétique d'un processeur est égale à :

$$E = \frac{1}{2} C U^2 \times f \times n$$

On en déduit donc facilement que diminuer la tension ou la fréquence permettent de diminuer la consommation énergétique, la diminution de tension ayant un effet plus marqué. Les fabricants de CPU ont donc eu l'idée de faire varier la tension et la fréquence en fonction de ce que l'on demande au processeur. Rien ne sert d'avoir un processeur qui tourne à 200 gigahertz pendant que l'on regarde ses mails. Par contre, avoir un processeur à cette fréquence peut être utile lorsque l'on joue à un jeu vidéo dernier cri. Dans ce cas, pourquoi ne pas adapter la fréquence suivant l'utilisation qui est faite du processeur ? C'est l'idée qui est derrière le **Dynamic Frequency Scaling**, aussi appelé DFS. De plus, diminuer la fréquence permet de diminuer la tension d'alimentation, pour diverses raisons techniques. La technologie consistant à diminuer la tension d'alimentation suivant les besoins s'appelle le **Dynamic Voltage Scaling**, de son petit nom : DVS. Ces techniques sont gérées par un circuit intégré dans le processeur, qui estime en permanence l'utilisation du processeur et la fréquence/tension adaptée.

Une autre solution consiste à jouer sur la manière dont l'horloge est distribuée dans le processeur. On estime que 20 à 35 % des pertes ont lieu dans l'arbre d'horloge (l'ensemble de fils qui distribuent l'horloge aux bascules). Une grande partie des pertes vient des composants reliés à l'horloge qui doivent changer d'état tant que l'horloge est présente, même quand ils sont inutilisés. La solution est de ne pas envoyer le signal d'horloge en entrée des différents circuits inutilisés : on appelle cela le **clock gating**. Pour implémenter cette technique, on est obligé de découper notre processeur en sous-circuits, tous reliés à l'horloge. L'ensemble de ces circuits formera un tout du point de vue de l'horloge : on pourra alors tous les déconnecter de l'horloge d'un coup. Pour implémenter le Clock Gating, on dispose entre l'arbre d'horloge et le circuit un circuit qui inhibera l'horloge au besoin. Ce circuit d'inhibition de l'horloge, qui s'appelle une Clock Gate, est relié à la fameuse unité de gestion de l'énergie intégrée dans le processeur qui se charge de le commander. Comme on le voit sur le schéma ci-dessus, ces Clock Gates sont commandées par un bit, qui ouvre ou ferme la Clock Gate. Ce signal est généré par une unité spéciale dans notre processeur.



L'influence de la loi de Moore

La loi de Moore a des conséquences assez intéressantes. En effet, l'évolution de la finesse de gravure permet d'augmenter le nombre de transistors et la fréquence, mais aussi de diminuer la tension d'alimentation et la capacité des transistors. Pour comprendre pourquoi, il faut savoir que le condensateur formé par la grille, l'isolant et le morceau de semi-conducteur est ce que l'on appelle un condensateur plan. La capacité de ce type de condensateur dépend de la surface de la plaque de métal (la grille), du matériau utilisé comme isolant (en fait, de sa permittivité), et de la distance entre la grille et le semi-conducteur. On peut calculer cette capacité comme suit, avec S la surface de la grille, e un paramètre qui dépend de l'isolant utilisé et d la distance entre le semi-conducteur et la grille (l'épaisseur de l'isolant, quoi).

$$\frac{S \times e}{d}$$

Généralement, le coefficient e (la permittivité électrique) reste le même d'une génération de processeur à l'autre. Les fabricants ont bien tenté de diminuer celui-ci, et trouver des matériaux ayant un coefficient faible n'a pas été une mince affaire. Le dioxyde de silicium pur a longtemps été utilisé, celui-ci ayant une permittivité de 4,2, mais il ne suffit plus de nos jours. De nombreux matériaux sont maintenant utilisés, notamment des terres rares. Leur raréfaction laisse planer quelques jours sombres pour l'industrie des processeurs et de la micro-électronique en général. Dans les faits, seuls les coefficients S et d vont nous intéresser.

Si la finesse de gravure diminue de $\sqrt{2}$, la distance d va diminuer du même ordre. Quant à la surface S , elle va diminuer du carré de $\sqrt{2}$, c'est à dire qu'elle sera divisée par 2. La capacité totale sera donc divisée par $\sqrt{2}$ tous les deux ans. La fréquence suit le même motif. Cela vient du fait que la période de l'horloge correspond grossièrement au temps qu'il faut pour remplir ou vider l'armature de la grille. Dans ces conditions, diminuer la capacité diminue le temps de remplissage/vidange, ce qui diminue la période et fait augmenter la fréquence.

Les équations de Dennard

Les conséquences d'une diminution par $\sqrt{2}$ de la finesse de gravure sont résumées dans le tableau ci-dessous. L'ensemble porte le nom de lois de Dennard, du nom de l'ingénieur qui a réussi à démontrer ces équations à partir des lois de la physique des semi-conducteurs.

Paramètre	Coefficient multiplicateur (tous les deux ans)
Finesse de gravure	$\frac{1}{\sqrt{2}}$
Nombre de transistors par unité de surface	2
Tension d'alimentation	$\frac{1}{\sqrt{2}}$
Capacité d'un transistor	$\frac{1}{\sqrt{2}}$
Fréquence	$\sqrt{2}$

Plus la finesse de gravure est faible, plus le composant électronique peut fonctionner avec une tension d'alimentation plus faible. La tension d'alimentation est proportionnelle à la finesse de gravure : diviser par deux la finesse de gravure divisera la tension d'alimentation par deux. La finesse de gravure étant divisée par la racine carrée de deux tous les deux ans, la tension d'alimentation fait donc de même. Cela a pour conséquence de diviser la consommation énergétique par deux, toute chose égale par ailleurs. Mais dans la réalité, les choses ne sont pas égales par ailleurs : la hausse du nombre de transistors va compenser exactement l'effet de la baisse de tension. A tension d'alimentation égale, le doublement du nombre de transistors tous les deux ans va faire doubler la consommation énergétique, ce qui compensera exactement la baisse de tension.

La finesse de gravure permet aussi de faire diminuer la capacité d'un transistor. Comme pour la tension d'alimentation, une division par deux de la finesse de gravure divise par deux la capacité d'un transistor. Dit autrement, la capacité d'un transistor est divisée par la racine carrée de deux tous les deux ans. On peut remarquer que cela compense exactement l'effet de la fréquence sur la consommation énergétique. La fréquence étant multipliée par la racine carrée de deux tous les deux ans, la consommation énergétique ferait de même toute chose égale par ailleurs.

Si on fait le bilan, la consommation énergétique des processeurs ne change pas avec le temps, du moins si ceux-ci gardent la même surface. Cette conservation de la consommation énergétique se fait cependant avec une augmentation de performance. Ainsi, la performance par watt d'un processeur augmente avec le temps. L'augmentation de performance étant de 80 % par an pour un processeur, on déduit rapidement que la performance par watt augmente de 80 % tous les deux ans. Du moins, c'est la théorie. Théorie qui fonctionnait bien il y a encore quelques années, mais qui ne se traduit plus dans les faits depuis la commercialisation des premiers processeurs Intel Core. Depuis, on observe que le nombre de transistors et la finesse de gravure suivent la tendance indiquée par la loi de Moore, mais cela ne permet plus de faire baisser la tension ou la fréquence. L'ère des équations de Dennard est aujourd'hui révolue, reste à vous expliquer pourquoi.

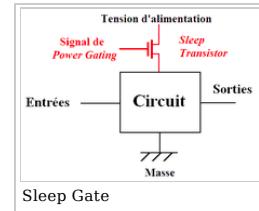
La fin des équations de Dennard

Les calculs précédents ne prenaient pas en compte la consommation statique, qui existe même quand nos transistors restent dans le même état. Il

nous reste à comprendre d'où vient cette consommation statique pour comprendre comment la finesse de gravure influe dessus. Pour commencer, la grille et la liaison source-drain d'un transistor sont séparés par un isolant : aucun courant ne devrait la traverser. Mais avec la miniaturisation, la couche d'isolant ne fait guère plus de quelques dizaines atomes d'épaisseur et laisse passer un peu de courant : on parle de **courants de fuite**. Plus cette couche d'isolant est petite, plus le courant de fuite sera fort. En clair, une diminution de la finesse de gravure a tendance à augmenter les courants de fuite. Les lois de la physique nous disent alors que la consommation d'énergie qui résulte de ce courant de fuite est égale au produit entre la tension d'alimentation et le courant de fuite.

Pour limiter la casse, on peut éteindre les circuits inutilisés d'un processeur, qui ne consommeront donc plus de courant : on parle de **power gating**. Elle s'implémente en utilisant des Power Gates, qui déconnecter les circuits de la tension d'alimentation quand ceux-ci sont inutilisés. Cette technique est très efficace, surtout pour couper l'alimentation du cache du processeur.

Une autre solution prend en compte le fait que certaines portions du processeur sont naturellement plus rapides que d'autres. Dans un processeur complet, on doit se limiter au plus petit dénominateur commun et s'aligner sur le circuit le plus lent. Vu que ces circuits fonctionnent à une fréquence inférieure à ce qu'ils peuvent, on peut baisser leur tension juste ce qu'il faut pour les faire aller à la bonne vitesse. Pour ce faire, on doit utiliser plusieurs tensions d'alimentation pour un seul processeur. Ainsi, certaines portions du processeur seront alimentées par une tension plus faible, tandis que d'autres seront alimentées par des tensions plus élevées.



Cependant, les courants de fuite traversent l'isolant du transistor, qui a une certaine résistance électrique. Or, quand on fait passer un courant dans une résistance, il se forme une tension à ses bornes, appelée **tension de seuil**. Cette tension de seuil est proportionnelle au courant et à la résistance (c'est la fameuse loi d'Ohm vue au collège). or, on ne peut pas faire fonctionner un transistor si la tension d'alimentation (entre source et drain) est inférieure à la tension de seuil. C'est pour cela que ces dernières années, la tension d'alimentation des processeurs est restée plus ou moins stable, à une valeur proche de la tension de seuil (1 volt, environ). Il faut alors trouver autre chose pour limiter la consommation à des niveaux soutenables. Les concepteurs de processeurs ne pouvaient pas diminuer la fréquence pour garder une consommation soutenable, et ont donc préféré augmenter le nombre de coeurs. L'augmentation de consommation énergétique ne découle que de l'augmentation du nombre de transistors et des diminutions de capacité. Et la diminution de 30 % tous les deux ans de la capacité ne compense plus le doublement du nombre de transistors : la consommation énergétique augmente ainsi de 40 % tous les deux ans. Bilan : la performance par watt stagne. Et ce n'est pas près de s'arranger tant que les tensions de seuil restent ce qu'elles sont.

Mémoires

Mémoire. Ce mot signifie dans le langage courant le fait de se rappeler quelque chose, de pouvoir s'en souvenir. La mémoire d'un ordinateur fait exactement la même chose (vous croyez qu'on lui a donné le nom de mémoire par hasard ? :-o) mais dans notre ordinateur. Son rôle est de retenir des données stockées sous la forme de suites de bits, afin qu'on puisse les récupérer si nécessaire et les traiter.

Performance d'une mémoire

Toutes les mémoires ne sont pas faites de la même façon et les différences entre mémoires sont nombreuses. Dans cette partie, on va passer en revue les différences les plus importantes.

Capacité mémoire

Pour commencer, une mémoire ne peut pas stocker une quantité infinie de données : qui n'a jamais eu un disque dur ou une clé USB plein ? Et à ce petit jeu là, toutes les mémoires ne sont pas égales : elles n'ont pas la même capacité. Cette **capacité mémoire** n'est autre que le nombre maximal de bits qu'une mémoire peut contenir. Dans la majorité des mémoires, les bits sont regroupés en paquets de taille fixe : des cases mémoires, aussi appelées **bytes**. De nos jours, le nombre de bits par byte est généralement un multiple de 8 bits : ces groupes de 8 bits s'appellent des octets. Mais toutes les mémoires n'ont pas des bytes d'un octet ou plusieurs octets : certaines mémoires assez anciennes utilisaient des cases mémoires contenant 1, 2, 3, 4, 7, 18, 36 bits.

Le fait que nos mémoires aient presque toutes des bytes faisant un octet nous arrange pour compter la capacité d'une mémoire. Au lieu de compter cette capacité en bits, on préfère mesurer la capacité d'une mémoire en donnant le nombre d'octets que celle-ci peut contenir. Mais les mémoires des PC font plusieurs millions ou milliards d'octets. Pour se faciliter la tâche, on utilise des préfixes pour désigner les différentes capacités mémoires. Vous connaissez sûrement ces préfixes : kibioctets, mebioctets et gibioctets, notés respectivement Kio, Mio et Gio.

Préfixe	Capacité mémoire en octets	Puissance de deux
Kio	1024	2^{10} octets
Mio	1 048 576	2^{20} octets
Gio	1 073 741 824	2^{30} octets

On peut se demander pourquoi utiliser des puissances de 1024, et ne pas utiliser des puissances un peu plus communes ? Dans la majorité des situations, les électroniciens préfèrent manipuler des puissances de deux pour se faciliter la vie. Par convention, on utilise souvent des puissances de 1024, qui est la puissance de deux la plus proche de 1000. Or, dans le langage courant, kilo, méga et giga sont des multiples de 1000. Quand vous vous pesez sur votre balance et que celle-ci vous indique 58 kilogrammes (désolé mesdames), cela veut dire que vous pesez 58000 grammes. De même, un kilomètre est égal à mille mètres, et non 1024 mètres.

Autrefois, on utilisait les termes kilo, méga et giga à la place de nos kibi, mebi et gibi, par abus de langage. Mais peu de personnes sont au courant de l'existence de ces nouvelles unités, et celles-ci sont rarement utilisées. Et cette confusion permet aux fabricants de disques durs de nous « arnaquer » : Ceux-ci donnent la capacité des disques durs qu'ils vendent en kilo, mega ou giga octets : l'acheteur croit implicitement avoir une capacité exprimée en kibi, mebi ou gibi octets, et se retrouve avec un disque dur qui contient moins de mémoire que prévu.

Temps d'accès

Si l'on souhaite accéder à une donnée dans une mémoire, il va falloir attendre un certain temps que la mémoire ait finie de lire ou d'écrire notre donnée : ce délai est appelé le **temps d'accès**. Généralement, lire une donnée ne prend pas le même temps que l'écrire : le temps d'accès en lecture est souvent inférieur au temps d'accès en écriture. Il faut dire qu'il est beaucoup plus fréquent de lire dans une mémoire qu'y écrire, et les fabricants préfèrent donc diminuer au maximum le temps d'accès en lecture comparé au temps d'écriture.

Voici les temps d'accès moyens en lecture de chaque type de mémoire :

- Registres : 1 nanoseconde (10-9)
- Caches : 10 - 100 nanosecondes (10-9)
- Mémoire RAM : 1 microseconde (10-6)
- Mémoires de masse : 1 milliseconde (10-3)

Débit

Enfin, toutes les mémoires n'ont pas le même **débit**. Par débit on veut dire que certaines sont capables d'échanger un grand nombre de données par secondes, alors que d'autres ne peuvent échanger qu'un nombre limité de données sur le bus. Cette quantité maximale de données que la mémoire peut envoyer ou recevoir par seconde porte le nom de débit binaire de la mémoire. Il est parfois improprement appelé bande passante. Evidemment, plus ce débit est élevé, plus la mémoire sera rapide. Celui-ci se mesure en octets par secondes ou en bits par secondes.

Mémoires volatiles et non-volatiles

Lorsque vous éteignez votre ordinateur, le système d'exploitation et les programmes que vous avez installés ne s'effacent pas, contrairement au document Word que vous avez oublié de sauvegarder. Les programmes et le système d'exploitation sont placés sur une mémoire qui ne s'efface pas quand on coupe le courant, contrairement à votre document Word non-sauvegardé. Cette observation nous permet de classer les mémoires en deux types : les mémoires **non-volatiles** conservent leurs informations quand on coupe le courant, alors que les mémoires **volatiles** les perdent.

Les mémoires volatiles peuvent être divisées en deux catégories : les mémoires statiques et mémoires dynamiques. Les données d'une **mémoire statique** ne s'effacent pas tant qu'elles sont alimentées en courant. Avec les **mémoires dynamiques**, les données s'effacent en quelques millièmes ou centièmes de secondes si l'on n'y touche pas. Il faut donc réécrire chaque bit de la mémoire régulièrement, ou après chaque lecture, pour éviter qu'il ne s'efface : on doit effectuer régulièrement un rafraîchissement mémoire. Le rafraîchissement prend du temps, et a tendance à légèrement diminuer la rapidité des mémoires dynamiques.

Mémoires ROM et RWM

Une autre différence concerne la façon dont on peut accéder aux informations stockées dans la mémoire. Avec les **mémoires ROM**, on peut récupérer les informations dans la mémoire, mais pas les modifier : la mémoire est dite accessible en lecture. La totalité de ces mémoires sont des mémoires volatiles. Si on ne peut pas écrire dans une ROM, on peut cependant réécrire intégralement son contenu : on dit qu'on reprogramme la ROM. Ce terme de programmation vient du fait que les mémoires ROM sont souvent utilisées pour stocker des programmes sur certains ordinateurs assez simples. On peut les classes en plusieurs types :

- les **mémoires FROM** sont fournies intégralement vierges, et on peut les programmer une seule fois ;
- les **mémoires PROM** sont reprogrammables ;
- les **mémoires EPROM** s'effacent avec des rayons UV et peuvent être reprogrammées plusieurs fois de suite ;
- certaines EPROM peuvent être effacées par des moyens électriques : ce sont les **mémoires EEPROM**.

Sur les **mémoires RWM**, on peut récupérer les informations dans la mémoire et les modifier : la mémoire est dite accessible en lecture et en écriture. Quand un composant souhaite échanger des données avec une mémoire RWM, il doit préciser le sens de transfert : lecture ou écriture. Pour préciser ce sens de transfert à la mémoire, il va devoir utiliser un bit du bus de commande nommé READ/WRITE, ou encore R/W (read veut dire lecture en anglais, alors que write veut dire écriture). Il est souvent admis par convention que R/W à 1 correspond à une lecture, tandis que R/W vaut 0 pour les écritures. Ce bit de commande est évidemment inutile sur les mémoires ROM, vu qu'elles ne peuvent effectuer que des lectures. Les mémoires RWM peuvent être aussi bien volatiles que non-volatiles.

Attention aux abus de langage : le terme mémoire RWM est souvent confondu dans le langage commun avec les mémoires RAM.

Tout ordinateur contient au moins une mémoire ROM et une mémoire RWM (souvent une RAM). La mémoire ROM stocke un programme et est souvent non-volatile. Dans les PC, cette mémoire contient un programme qui permet à l'ordinateur de démarrer : le **BIOS**. Une fois l'ordinateur démarré, celui-ci charge le système d'exploitation dans la mémoire RWM (la RAM) et lui rend la main. La mémoire RWM est par contre une mémoire volatile, qui stocke temporairement des données que le processeur doit manipuler. Elle sert donc essentiellement pour maintenir des résultats de calculs. Elle peut éventuellement mémoriser des programmes à exécuter, mais seulement si la ROM est trop lente ou trop petite. C'est notamment le cas sur le PC que vous êtes en train d'utiliser : les programmes sont mémorisés sur le disque dur de votre ordinateur (une mémoire de masse), et sont copiés en mémoire RAM à chaque fois que vous les lancez. On peut préciser que le système d'exploitation ne fait pas exception à la règle, vu qu'il est lancé par le BIOS.

Architecture Von Neumann

Si ces deux mémoires sont reliées au processeur par un bus unique, on a une **architecture Von Neumann**. Avec l'architecture Von Neumann, tout se passe comme si les deux mémoires étaient fusionnées en une seule mémoire. Une adresse bien précise va ainsi correspondre soit à la mémoire RAM, soit à la mémoire ROM, mais pas aux deux. Quand une adresse est envoyée sur le bus, les deux mémoires vont la recevoir mais une seule va répondre.

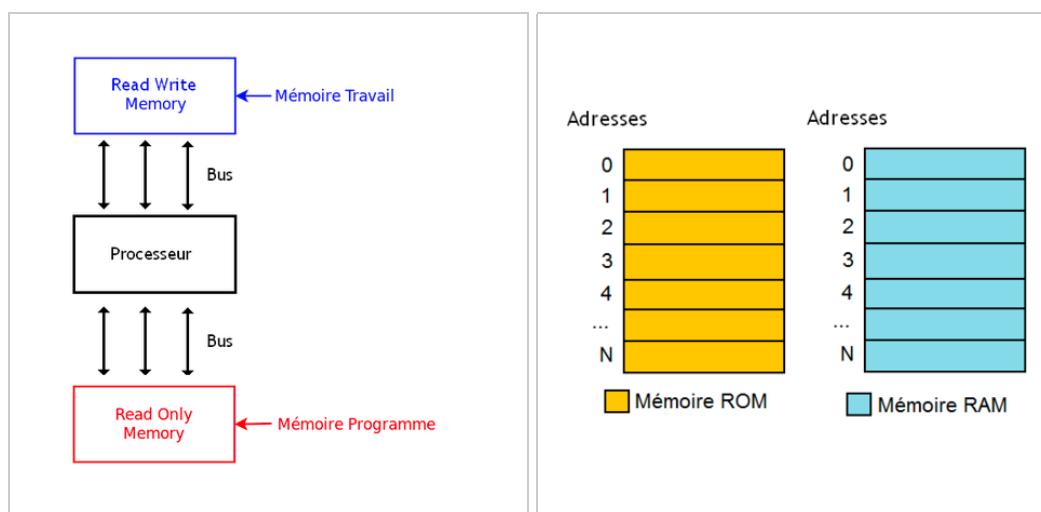


Architecture Von Neumann, avec deux bus séparés.

Vision de la mémoire par un processeur sur une architecture Von Neumann.

Architecture Harvard

Si ces deux mémoires sont reliées au processeur par deux bus séparés, on a une **architecture Harvard**. L'avantage de cette architecture est qu'elle permet de charger une instruction et une donnée simultanément : une instruction chargée sur le bus relié à la mémoire programme, et une donnée chargée sur le bus relié à la mémoire de données. Sur ces architectures, une adresse peut correspondre soit à la ROM, soit à la RAM : le processeur voit bien deux mémoires séparées.



Architecture Harvard, avec une ROM et une RAM séparées.

Vision de la mémoire par un processeur sur une architecture Harvard.

Adressage et accès

Les mémoires se différencient aussi par la méthode d'accès aux données mémorisées.

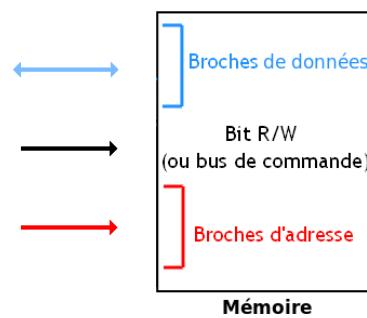
Mémoires adressables

Les mémoires actuelles utilisent l'**adressage** : chaque case mémoire se voit attribuer un numéro, l'**adresse**, qui va permettre de la sélectionner et de l'identifier parmi toutes les autres. On peut comparer une adresse à un numéro de téléphone (ou à une adresse d'appartement) : chacun de vos correspondants a un numéro de téléphone et vous savez que pour appeler telle personne, vous devez composer tel numéro. Les adresses mémoires en sont l'équivalent pour les cases mémoires. Ces mémoires adressables peuvent se classer en deux types : les **mémoires à accès aléatoire**, et les **mémoires adressables par contenu**.

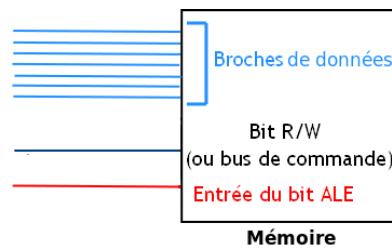
Les **mémoires à accès aléatoire** sont des mémoires adressables, sur lesquelles on doit préciser l'adresse de la donnée à lire ou modifier. Certaines d'entre elles sont des mémoires électroniques non-volatiles de type ROM : on les nomme, par abus de langage, des mémoires ROM. D'autres sont des mémoires volatiles RWM : elles portent alors, par abus de langage, le nom de mémoires RAM (Random Access Memory). Parmi les mémoires RAM volatiles, on peut distinguer les SRAM des DRAM : les premières sont des mémoires statiques alors que les secondes sont des mémoires dynamiques.

Les **mémoires associatives** fonctionnent comme une mémoire à accès aléatoire, mais dans le sens inverse. Au lieu d'envoyer l'adresse pour accéder à la donnée, on va envoyer la donnée pour récupérer son adresse : à la réception de la donnée, la mémoire va déterminer quelle case mémoire contient cette donnée et renverra l'adresse de cette case mémoire. Cela peut paraître bizarre, mais ces mémoires sont assez utiles dans certains cas de haute volée. Dès que l'on a besoin de rechercher rapidement des informations dans un ensemble de données, ou de savoir si une donnée est présente dans un ensemble, ces mémoires sont reines. Certains circuits internes au processeur ont besoin de mémoires qui fonctionnent sur ce principe. Mais laissons cela à plus tard.

Toutes les mémoires adressables sont naturellement connectées au bus. Mais celui-ci ne se limite plus à un bus de données couplé à un bus de commande : il faut ajouter un troisième bus pour envoyer les adresses à la mémoire (ou les récupérer, sur une mémoire associative).

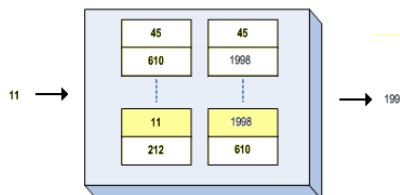


Il existe une petite astuce pour économiser des fils : utiliser un bus qui servira alternativement de bus de donnée ou d'adresse. Ces bus rajoutent un bit sur le bus de commande, qui précise si le contenu du bus est une adresse ou une donnée. Ce bit Adresse Line Enable, aussi appelé bit ALE, vaut 1 quand une adresse transite sur le bus, et 0 si le bus contient une donnée (ou l'inverse !). Ce genre de bus est plus lent pour les écritures : l'adresse et la donnée à écrire ne peuvent pas être envoyées en même temps. Par contre, les lectures ne posent pas de problèmes, vu que l'envoi de l'adresse et la lecture proprement dite ne sont pas simultanées. Heureusement, les lectures en mémoire sont bien plus courantes que les écritures, ce qui fait que la perte de performance due à l'utilisation d'un bus multiplexé est souvent supportable.



Mémoires caches

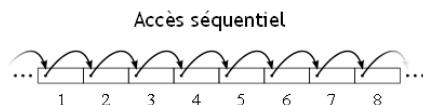
Sur les **mémoires caches**, chaque donnée se voit attribuer un identifiant, qu'on appelle le **tag**. Une mémoire à correspondance stocke non seulement la donnée, mais aussi l'identifiant qui lui est attribué : cela permet ainsi de mettre à jour l'identifiant, de le modifier, etc. En somme, le Tag remplace l'adresse, tout en étant plus souple. La mémoire cache stocke donc des couples tag-donnée. A chaque accès mémoire, on envoie le tag de la donnée voulue pour sélectionner la donnée.



Il faut noter qu'il est possible de créer une mémoire cache en utilisant une mémoire RAM et une mémoire associative, ainsi que quelques circuits pour faire le lien entre les deux.

Mémoires séquentielles

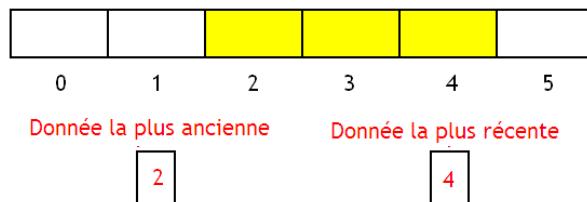
Sur d'anciennes mémoires, comme les bandes magnétiques, on était obligé d'accéder aux données dans un ordre prédéfini. On parcourait la mémoire dans l'ordre, en commençant par la première donnée : c'est l'accès séquentiel. Pour lire ou écrire une donnée, il fallait visiter toutes les cases mémoires précédentes avant de tomber sur la donnée recherchée. Et impossible de revenir en arrière !



Mémoires FIFO et LIFO

Les mémoires à accès séquentiel ne sont pas les seules à imposer un ordre d'accès aux données. Il existe deux autres types de mémoires qui forcent l'ordre d'accès : les mémoires FIFO et LIFO. Dans les deux cas, les données sont triées dans la mémoire dans l'ordre d'écriture, ce dernier étant la même chose que l'ordre d'arrivée. La différence est qu'une lecture dans une **mémoire FIFO** renvoie la donnée la plus ancienne, alors que pour une **mémoire LIFO**, elle renverra la donnée la plus récente, celle ajoutée en dernier dans la mémoire. Dans les deux cas, la lecture sera destructrice : la donnée lue est effacée. On peut voir les mémoires FIFO comme des files d'attente, des mémoires qui permettent de mettre en attente des données tant qu'un composant n'est pas prêt. De même, on peut voir les mémoires LIFO comme des piles de données : toute écriture empilera une donnée au sommet de cette mémoire LIFO, alors qu'une lecture enlèvera la donnée au sommet de la pile..

Il est facile de créer ce genre de mémoire à partir d'une mémoire RAM en y ajoutant des circuits pour gérer les ajouts et retraits de données. L'adresse de la donnée la plus ancienne, ainsi que l'adresse de la plus récente sont mémorisées dans des registres. Quand une donnée est retirée, l'adresse la plus récente est décrémentée, pour pointer sur la prochaine donnée. Quand une donnée est ajoutée, l'adresse la plus ancienne est incrémentée pour pointer sur la bonne donnée.



Petit détail : quand on ajoute des instructions dans la mémoire, il se peut que l'on arrive au bout, à l'adresse maximale, même s'il reste de la place à cause des retraits de données. La prochaine entrée à être remplie sera celle numérotée 0, et on poursuivra ainsi de suite.



Cellules mémoires

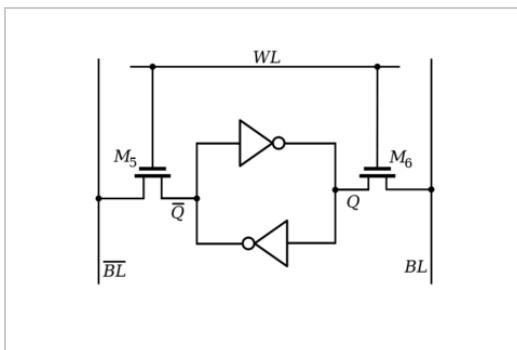
De nos jours, ces cellules mémoires sont fabriquées avec des composants électroniques et il nous faudra impérativement passer par une petite étude de ces composants pour comprendre comment fonctionnent nos mémoires. Dans les grandes lignes, les mémoires RAM et ROM actuelles sont toutes composées de **cellules mémoires**, des circuits capables de retenir un bit. En prenant plein de ces cellules et en ajoutant quelques circuits électroniques pour gérer le tout, on obtient une mémoire. Dans ce chapitre, nous allons apprendre à créer nos propres bits de mémoire à partir de composants élémentaires : des transistors et des condensateurs.

SRAM

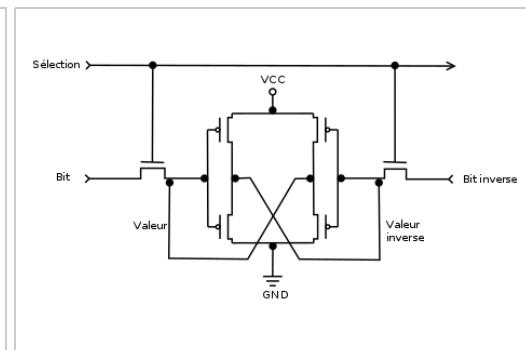
Les cellules des mémoires SRAM sont les bascules D vues dans les premiers chapitres de ce cours. Ces bascules peuvent être créées à partir de portes logiques, ce qui donne un circuit composé de 10 à 20 transistors. Mais certaines mémoires SRAM arrivent à se débrouiller avec seulement 4 ou 2 transistors par bit, les processeurs actuels utilisant une variante à 6 transistors. Pour simplifier, une bascule de SRAM est construite autour d'un circuit composé :

- de deux inverseurs reliés tête-bêche (la sortie de l'un sur l'entrée de l'autre) ;
- de deux transistors qui servent à autoriser les lectures et écritures.

Les portes NON reliées tête-bêche mémorisent un bit. Si on place un bit en entrée d'une porte, ce bit sera inversé deux fois avant de retomber sur l'entrée. L'ensemble sera stable : on peut déconnecter l'entrée d'un inverseur, elle sera rafraîchie en permanence par l'autre, avec sa valeur précédente. Les deux transistors permettent de connecter les entrées des portes NON sur le bus, afin de modifier ou lire le contenu de la bascule.



Cellule de SRAM.



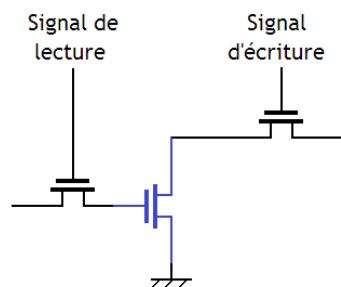
Cellule de SRAM.

DRAM

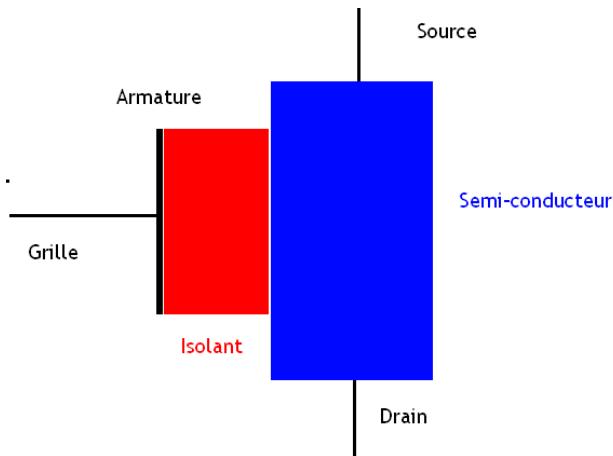
Comme pour les SRAM, les DRAM sont composées d'un circuit de mémorisation, avec des transistors autour pour autoriser les lectures et écritures. La différence avec les SRAM tient dans le circuit utilisé pour mémoriser un bit.

3T-DRAM

Les premières mémoires DRAM fabriquées commercialement utilisaient 3 transistors. Le bit est mémorisé dans celui du milieu, indiqué en bleu sur le schéma suivant, les deux autres transistors servant pour les lectures et écritures.



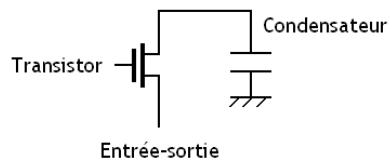
Pour comprendre ce qui se passe dans le transistor de mémorisation, il faut savoir comment fonctionne un transistor CMOS. À l'intérieur, on trouve une plaque en métal appelée l'armature, un bout de semi-conducteur entre la source et le drain, et un morceau d'isolant entre les deux. Suivant la tension qu'on envoie sur la grille, l'armature va se remplir d'électrons ou se vider, ce qui permet de stocker un bit : une grille pleine compte pour un 1, une grille vide compte pour un 0. Cette armature n'est pas parfaite et elle se vide régulièrement, d'où le fait que la mémoire obtenue soit une DRAM.



Il faut remarquer qu'avec cette organisation, lire un bit ne le détruit pas : on peut relire plusieurs fois un bit sans que celui-ci ne soit effacé. C'est une qualité que les DRAM modernes n'ont pas.

1T-DRAM

Les DRAM actuelles fonctionnent différemment : elle n'utilisent qu'un seul et unique transistor, et un autre composant électronique nommé un condensateur. Ce condensateur n'est qu'un réservoir à électrons : on peut le remplir d'électrons ou le vider en mettant une tension sur ses entrées. C'est ce condensateur qui va stocker notre bit : le condensateur stocke un 1 s'il est rempli, un 0 s'il est vide. A côté, on ajoute un transistor qui va autoriser l'écriture ou la lecture dans notre condensateur. Tant que notre transistor se comporte comme un interrupteur ouvert, le condensateur est isolé du reste du circuit : pas d'écriture ou de lecture possible. Si on l'ouvre, on pourra alors lire ou écrire dedans. Une DRAM peut stocker plus de bits pour la même surface qu'une SRAM : un transistor couplé à un condensateur prend moins de place que 6 transistors.



Seul problème : quand on veut lire ou écrire dans notre cellule mémoire, le condensateur va être connecté sur le bus de données et se vider entièrement : on prend son contenu et il faut le récrire après chaque lecture. Pire : le condensateur se vide sur le bus, mais cela ne suffit pas à créer une tension de plus de quelques millivolts dans celui-ci. Pas de quoi envoyer un 1 sur le bus ! Mais il y a une solution : amplifier la tension de quelques millivolts induite par la vidange du condensateur sur le bus, avec un circuit adapté.

Il faut préciser qu'un condensateur est une vraie passoire : il possède toujours quelques défauts et des imperfections qui font que celui-ci se vide tout seul au bout d'un moment. Pour expliquer pourquoi, il faut savoir qu'un condensateur est formé de deux morceaux de conducteur (les armatures) séparés par un isolant. Logiquement, l'isolant empêche les électrons de passer d'une armature à l'autre, mais l'isolant n'est jamais totalement étanche : des électrons passent de temps en temps d'une armature à l'autre et quittent le condensateur. En clair, le bit contenu dans la cellule de mémoire DRAM s'efface.

ROM

Pour rappel, il existe différents types de ROM, qui sont construites différemment :

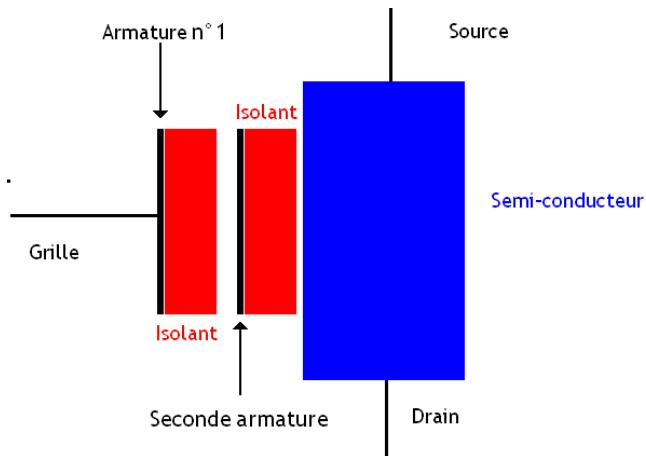
- celles qu'on ne peut pas effacer ni reprogrammer (réécrire tout leur contenu d'un seul coup) : les PROM ;
- celles qu'on peut effacer et reprogrammer électriquement : les EEPROM.

PROM

Dans les mémoires PROM, chaque bit est stocké en utilisant un fusible : un 1 est codé par un fusible intact, et le zéro par un fusible grillé. Suivant la mémoire, ce fusible peut être un transistor, ou une diode. Programmer une PROM consiste à faire claquer certains fusibles en les soumettant à une tension très élevée, pour les fixer à 0. Une fois le fusible claqué, on ne peut pas revenir en arrière : la mémoire est programmée définitivement.

EPROM et EEPROM

Les mémoires EPROM et EEPROM sont fabriquées avec des **transistors à grille flottante** (un par cellule mémoire), des transistors qui possèdent deux armatures et deux couches d'isolant. La seconde armature est celle qui stocke un bit : il suffit de la remplir d'électrons pour stocker un 1, et la vider pour stocker un 0. Pour effacer une EPROM, on doit soumettre la mémoire à des ultra-violets : ceux-ci vont donner suffisamment d'énergie aux électrons coincés dans l'armature pour qu'ils puissent s'échapper. Pour les EEPROM, ce remplissage ou vidage se fait en faisant passer des électrons entre la grille et le drain, et en plaçant une tension sur la grille : les électrons passeront alors dans l'armature à travers l'isolant.



Sur la plupart des EEPROM, un transistor à grille flottante sert à mémoriser un bit. La tension contenue dans la seconde armature est alors divisée en deux intervalles : un pour le zéro, et un autre pour le un. De telles mémoires sont appelées des mémoires SLC (**Single Level Cell**). Mais d'autres EEPROM utilisent plus de deux intervalles, ce qui permet de stocker plusieurs bits par transistor. De telles mémoires sont appelées des mémoires MLC (**Multi Level Cell**). Évidemment, utiliser un transistor pour stocker deux à trois fois plus de bits aide beaucoup les mémoires MLC à obtenir une grande capacité.

Contrôleur mémoire interne

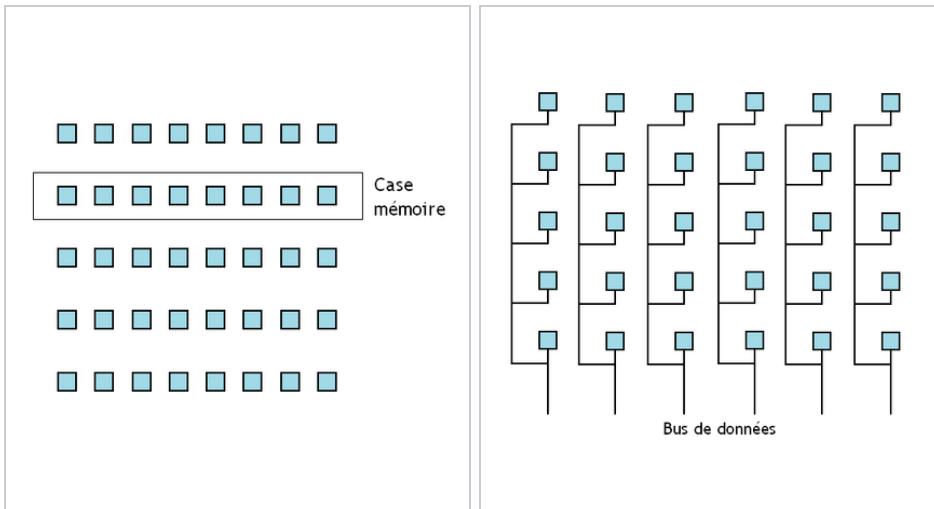
La gestion de l'adressage et de la communication avec le bus sont assurées par un circuit spécialisé : le contrôleur mémoire. Une mémoire adressable est ainsi composée :

- d'un plan mémoire ;
- du contrôleur mémoire ;
- et des connexions avec le bus.

Nous avons vu le fonctionnement du plan mémoire dans les chapitres précédents. Les circuits qui font l'interface entre le bus et la mémoire ne sont pas différents des circuits qui relient n'importe quel composant électronique à un bus, aussi ceux-ci seront vus dans le chapitre sur les bus. Bref, il est maintenant temps de voir comment fonctionne un contrôleur mémoire. Je parlerai du fonctionnement des mémoires multiports dans le chapitre suivant.

Mémoires à adressage linéaire

Pour commencer, nous allons voir les mémoires à **adressage linéaire**. Sur ces mémoires, le plan mémoire est un tableau rectangulaire de cellules mémoires, et toutes les cellules mémoires d'une ligne appartiennent à une même case mémoire. Chaque cellule mémoire est connectée sur un fil qui lui permettra de communiquer avec le bus de données : la bitline. Avec cette organisation, la cellule mémoire stockant le i ème bit du contenu d'une case mémoire (le bit de poids i) est reliée au i ème fil du bus.

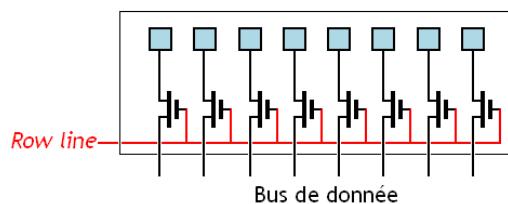


Principe d'un plan mémoire linéaire.

Plan mémoire, avec les bitlines.

Connexion au bus

Vu qu'une case mémoire est stockée sur une ligne, gérer l'adressage est très simple : il suffit de connecter la bonne ligne sur les bitlines, et de déconnecter les autres. Cette connexion/déconnexion est réalisée par des transistors intégrés dans la cellule mémoire, qui servent d'interrupteur. On a vu dans le chapitre précédent qu'il y a un à deux transistors de ce type dans une cellule de DRAM ou de SRAM (un pour les 1T DRAM, deux pour les autres types de SRAM/DRAM). Les transistors des cellules mémoires d'un byte doivent s'ouvrir ou se fermer en même temps : on relie donc leur grille à un même signal de commande, qu'on nomme row line.

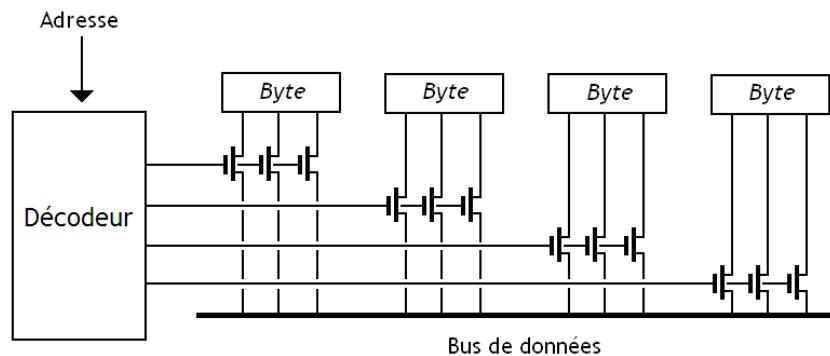


Décodeurs

Le rôle du contrôleur mémoire est de déduire quelle entrée Row Line mettre à un à partir de l'adresse envoyée sur le bus d'adresse. Pour cela, le contrôleur mémoire doit répondre à plusieurs contraintes :

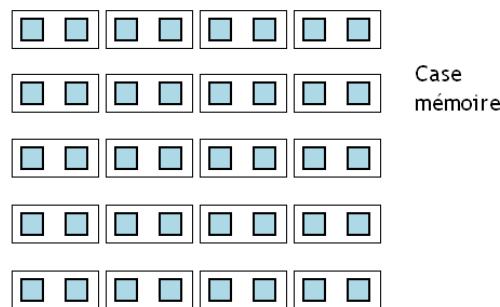
- il reçoit des adresses codées sur n bits : ce contrôleur a donc n entrées ;
- l'adresse de n bits peut adresser 2^n bytes : notre contrôleur mémoire doit donc posséder 2^n sorties ;
- la sortie numéro N est reliée au N -ième signal Row Line (et donc à la N -ième case mémoire) ;
- on ne doit sélectionner qu'une seule case mémoire à la fois : une seule sortie devra être placée à 1 ;
- et enfin, deux adresses différentes devront sélectionner des cases mémoires différentes : la sortie de notre contrôleur qui sera mise à 1 sera différente pour deux adresses différentes placées sur son entrée.

Le seul composant électronique qui répond à ce cahier des charges est le décodeur. Le contrôleur mémoire se résume à un simple décodeur, avec quelques circuits pour gérer le sens de transfert (lecture ou écriture), et la détection/correction d'erreur. Ce genre d'organisation s'appelle l'adressage linéaire.



Mémoires à adressage par coïncidence

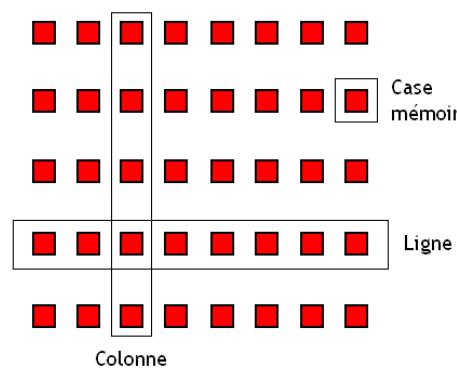
Sur des mémoires ayant une grande capacité, le décodeur utilise trop de portes logiques et le temps de propagation augmente à cause de la longueur des bitlines. Pour éviter cela, on est obligé de diminuer le nombre de lignes. Pour cela, certaines mémoires regroupent plusieurs cases mémoire sur une seule ligne, ce qui, pour un nombre de cases mémoires fixé, diminue le nombre de lignes.



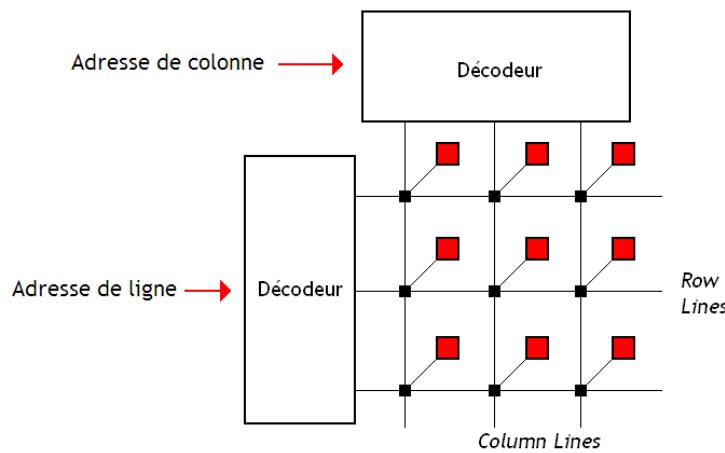
Ainsi, adresser la mémoire demande de sélectionner la ligne voulue, et de sélectionner la case mémoire à l'intérieur de la ligne. Sélectionner une ligne est facile : on utilise un décodeur. Mais la méthode utilisée pour sélectionner la colonne dépend de la mémoire utilisée. Commençons par aborder la première méthode, celle des mémoires à adressage par coïncidence.

Adressage par coïncidence

Sur ces mémoires, les cases mémoire sont organisées en lignes et en colonnes, avec une case mémoire à l'intersection entre une colonne et une ligne.

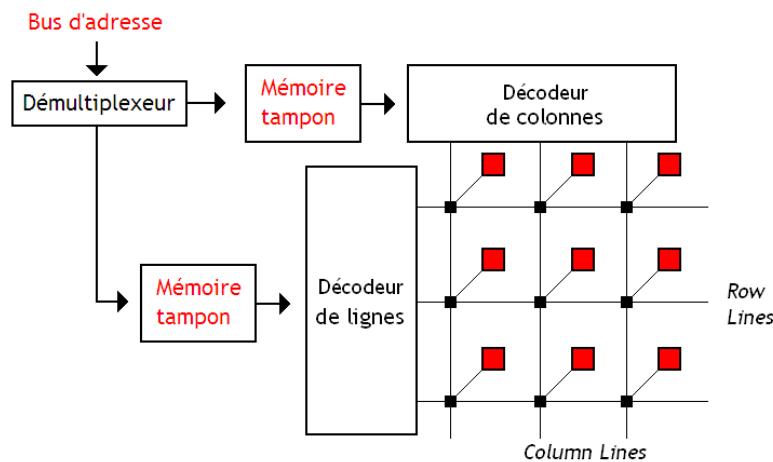


Toutes les cases mémoire d'une colonne sont reliées à un autre fil, le column line. Une case mémoire est sélectionnée quand ces row lines et la column line sont tous les deux mis à 1 : il suffit d'envoyer ces deux signaux aux entrées d'une porte ET pour obtenir le signal d'autorisation de lecture/écriture pour une cellule.



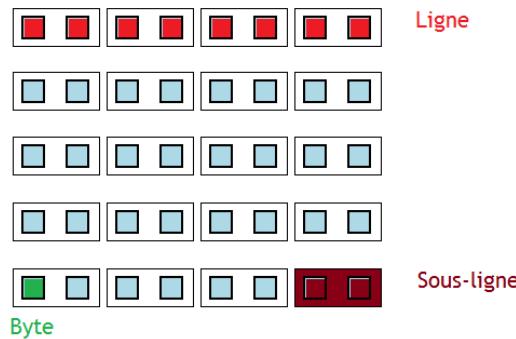
On utilise donc deux décodeurs : un pour sélectionner la ligne (pour le row line) et un autre pour sélectionner la colonne. Sur certaines mémoires, les deux décodeurs fonctionnent en même temps : on peut décoder une ligne en même temps qu'une colonne. Sur d'autres, on envoie les adresses en deux fois : d'abord l'adresse de ligne, puis l'adresse de colonne. Cela permet d'économiser des fils sur le bus d'adresse, mais nécessite de modifier l'intérieur de la mémoire.

Si l'adresse est envoyée en deux fois, la mémoire doit mémoriser les deux morceaux de l'adresse : si la mémoire ne se souvient pas de l'adresse de la ligne, elle ne pourra pas sélectionner le byte voulu. Pour cela, on place deux registres, entre les décodeurs et le bus d'adresse. Il faut aussi ajouter de quoi aiguiller le contenu du bus d'adresse vers le bon registre, en utilisant un démultiplexeur.

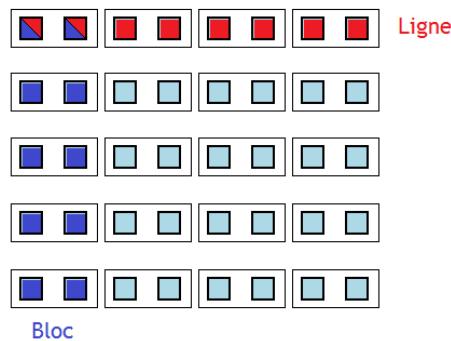


Divided word line

Cette organisation par coïncidence peut être améliorée en rajoutant un troisième niveau de subdivision : chaque ligne est découpée en sous-lignes, qui contiennent plusieurs colonnes. On obtient alors des **mémoires par blocs**, ou divided word line structures. Chaque ligne est donc découpée en N lignes, numérotées de 0 à N-1. Les sous-lignes qui ont le même numéro sont en quelque sorte alignées verticalement, et sont reliées aux mêmes bitlines : celles-ci forment ce qu'on appelle un bloc. Chacun de ces blocs contient un plan mémoire, un multiplexeur, et éventuellement des amplificateurs de lecture et des circuits d'écriture.



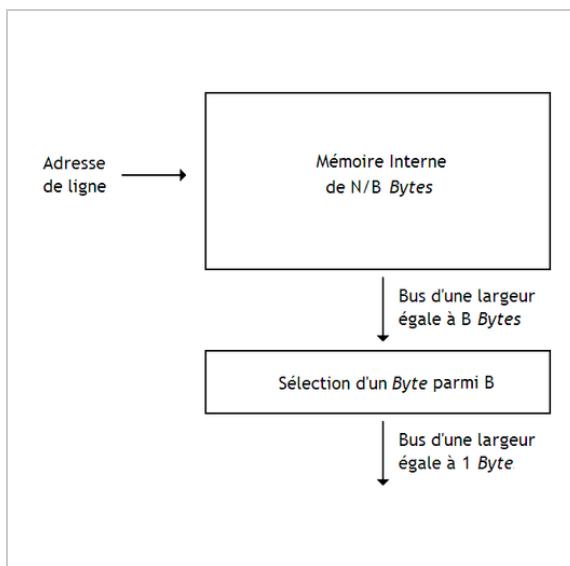
Tous les blocs de la mémoire sont reliés au décodeur d'adresse de ligne. Mais malgré tout, on ne peut pas accéder à plusieurs blocs à la fois : seul un bloc est actif lors d'une lecture ou d'une écriture. Pour cela, un circuit de sélection du bloc se charge d'activer ou de désactiver les blocs inutilisés lors d'une lecture ou écriture. L'adresse d'une sous-ligne bien précise se fait par coïncidence entre une ligne, et un bloc.



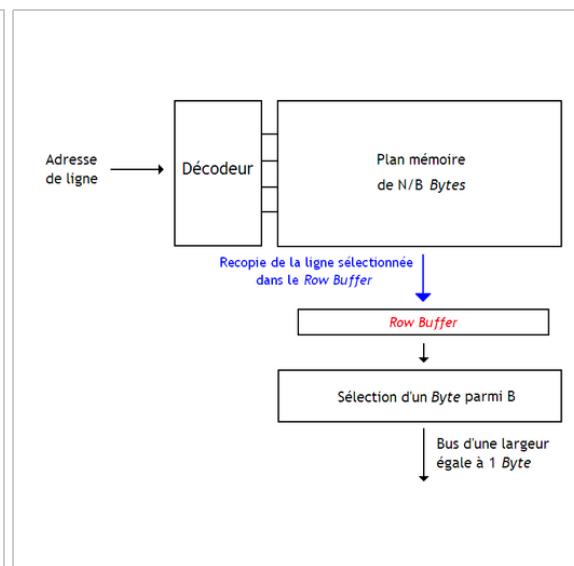
La ligne complète est activée par un signal wordline, généré par un décodeur de ligne. Les blocs sont activés individuellement par un signal nommé blocline, produit par un décodeur de bloc : ce décodeur prend en entrée l'adresse de bloc, et génère le signal blocline pour chaque bloc. Ensuite, une fois la sous-ligne activée, il faut encore sélectionner la colonne à l'intérieur de la sous-ligne sélectionnée, ce qui demande un troisième décodeur. L'adresse mémoire est évidemment coupée en trois : une adresse de ligne, une adresse de sous-ligne, et une adresse de colonne.

Mémoires à row buffer

Enfin, nous allons voir une espèce de mélange entre les deux types de mémoires vus précédemment : les **mémoires à row buffer**. Une mémoire à row buffer émule une mémoire de N bytes à partir d'une mémoire contenant B fois moins de bytes, mais dont chacun des bytes seront B fois plus gros. Chaque accès va lire un « super-byte » de la mémoire interne, et sélectionner le bon byte dans celui-ci. Sur le schéma du dessous, on voit bien que notre mémoire est composée de deux grands morceaux : une mémoire à adressage linéaire, et un circuit de sélection d'un mot mémoire parmi B (souvent un multiplexeur). Sur la grosse majorité de ces mémoires, chaque ligne sélectionnée est recopiée intégralement dans une sorte de gros registre temporaire, le row buffer, dans lequel on viendra sélectionner la case mémoire correspondant à notre colonne.



Mémoire à row buffer - 1.



Mémoire à row buffer - 2.

Les plans mémoire à row buffer récupèrent les avantages des plans mémoires par coïncidence : possibilité de décoder une ligne en même temps qu'une colonne, possibilité d'envoyer l'adresse en deux fois, consommation moindre de portes logiques, etc. De plus, cela permet d'effectuer l'opération de rafraîchissement très simplement, en rafraîchissant une ligne à la fois, au lieu d'une case mémoire à la fois. Autre avantage : en concevant correctement la mémoire, il est possible d'améliorer les performances lors de l'accès à des données proches en mémoire : si on doit lire ou écrire deux mots mémoires localisés dans la même ligne de notre mémoire interne, il suffit de charger celle-ci une fois dans le row buffer, et de faire deux sélections de colonnes différentes. C'est plus rapide que de devoir faire deux sélections de lignes et deux de colonnes. On en reparlera lorsqu'on verra les mémoires SDRAM et EDO.

Par contre, cette organisation consomme beaucoup d'énergie. Il faut dire que pour chaque lecture d'un byte dans notre mémoire, on doit charger une ligne de plusieurs cases mémoires dans le row buffer.

Mémoires évoluées

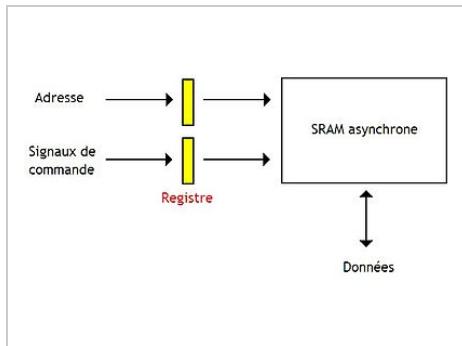
Les mémoires vues au chapitre précédent sont les mémoires les plus simples qui soient. Mais ces mémoires peuvent se voir ajouter quelques améliorations pas franchement négligeables, afin d'augmenter leur rapidité, ou de diminuer leur consommation énergétique. Dans ce chapitre, nous allons voir quelles sont ces améliorations les plus courantes.

Mémoires synchrones

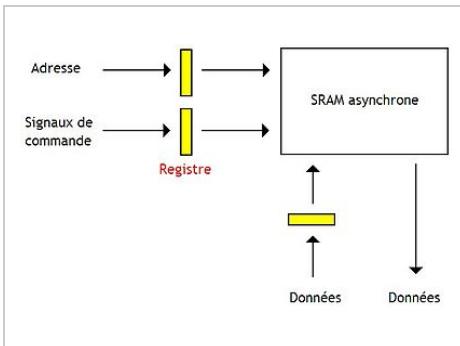
Les toutes premières mémoires n'étaient pas synchronisées avec le processeur via une horloge : c'était des **mémoires asynchrones**. Un accès mémoire devait être plus court qu'un cycle d'horloge processeur. Avec le temps, le processeur est devenu plus rapide que la mémoire : il ne pouvait pas prévoir quand la donnée serait disponible et ne faisait rien tant que la mémoire n'avait pas répondu. Pour résoudre ces problèmes, les concepteurs de mémoire ont synchronisé les échanges entre processeur et mémoire avec un signal d'horloge : les **mémoires synchrones** sont nées. L'utilisation d'une horloge a l'avantage d'imposer des temps d'accès fixes : le processeur sait qu'un accès mémoire prendra un nombre déterminé (2, 3, 5, etc) de cycles d'horloge et peut faire ce qu'il veut dans son coin durant ce temps.

Mémoires synchrones non-pipelinées

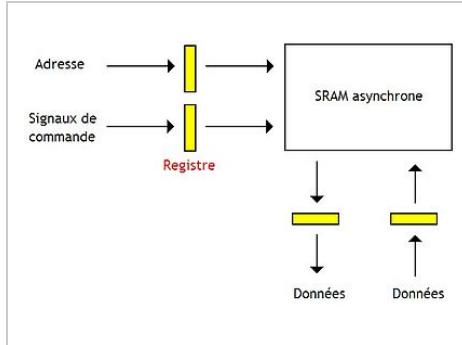
Fabriquer une mémoire synchrone demande de rajouter des registres sur les entrées/sorties d'une mémoire asynchrone. Ainsi, le processeur n'a pas à maintenir l'adresse en entrée de la mémoire durant toute la durée d'un accès mémoire : le registre s'en charge. La première méthode ne mémorise que l'adresse d'entrée et les signaux de commande dans un registre synchronisé sur l'horloge. Une seconde méthode mémorise l'adresse, les signaux de commande, ainsi que les données à écrire. Et la dernière méthode mémorise toutes les entrées et sorties de la mémoire dans des registres synchronisés sur l'horloge.



Mémoire synchrone première génération.



Mémoire synchrone seconde génération.



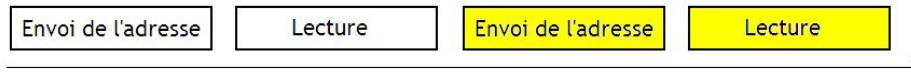
Mémoire synchrone troisième génération.

Mémoires synchrones pipelinées

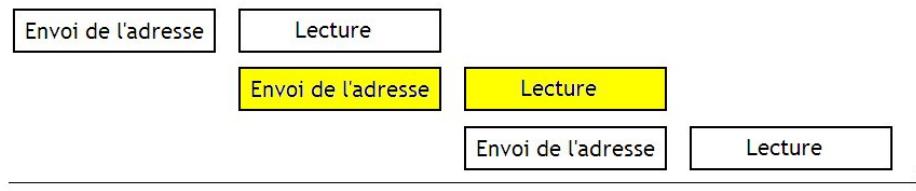
L'usage d'une horloge permet aussi de faciliter l'implantation de ce qu'on appelle un pipeline. Pour rappel, la sélection d'une case mémoire se fait en plusieurs étapes, chaque étape prenant un cycle d'horloge. Suivant la mémoire, le nombre d'étapes varie :

- on peut envoyer l'adresse lors d'un cycle, et récupérer la donnée lue au cycle suivant ;
- on peut aussi envoyer l'adresse lors d'un premier cycle, effectuer la lecture durant le cycle suivant, et récupérer la donnée sur le bus un cycle plus tard ;
- on peut aussi tenir compte du fait que la mémoire est organisée en lignes et en colonnes : l'étape de lecture peut ainsi être scindée en une étape pour sélectionner la ligne, et une autre pour sélectionner la colonne ;
- etc.

Avec une mémoire synchrone sans pipeline, on doit attendre qu'un accès mémoire soit fini avant d'en démarrer un autre.



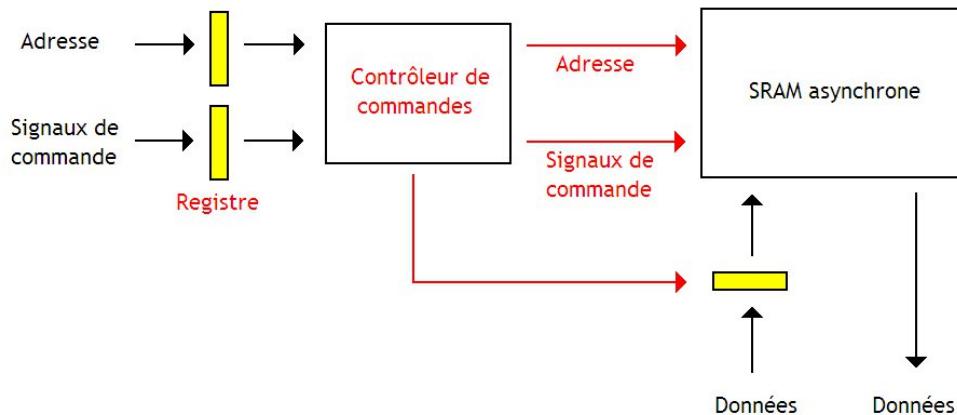
Avec les mémoires pipelinées, on peut réaliser un accès mémoire sans attendre que les précédents soient finis. En quelque sorte, ces mémoires ont la capacité de traiter plusieurs requêtes de lecture/écriture en même temps, tant que celles-ci sont chacune à des étapes différentes. Par exemple, on peut envoyer l'adresse de la prochaine requête pendant que l'on accède à la donnée de la requête courante. Les mémoires qui utilisent ce pipelining ont donc un débit supérieur aux mémoires qui ne l'utilisent pas : le nombre d'accès mémoire traités par seconde augmente.



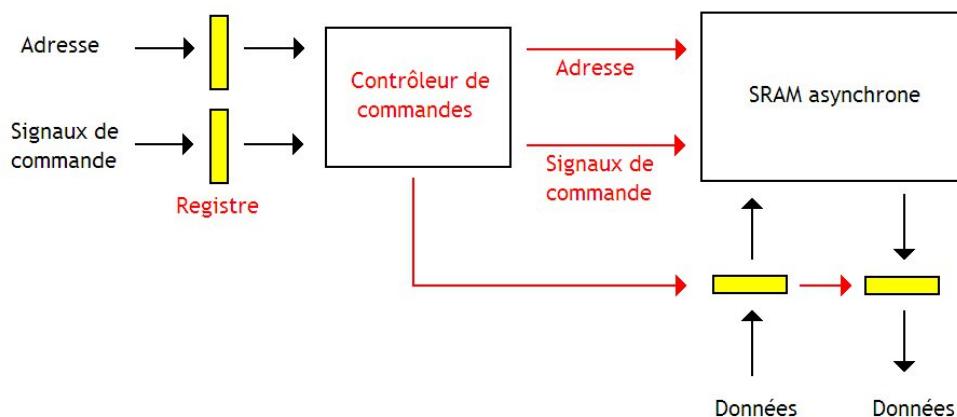
On remarque que le nombre de cycles nécessaires pour traiter une requête de lecture/écriture augmente : d'un seul cycle, on passe à autant de cycles qu'il y a d'étapes. Mais un cycle sur une mémoire non-pipelinée correspond à un accès mémoire complet, tandis qu'un cycle est égal à la durée d'une seule étape sur une mémoire pipelinée : à temps d'accès identique, la durée d'un cycle d'horloge est plus petite sur la mémoire pipelinée. Dit autrement, la fréquence est supérieure pour les mémoires pipelinées, ce qui compense exactement l'augmentation du nombre d'étapes.

Cette technique demande d'enchaîner les différentes étapes à chaque cycle : les adresses et ordres de commande doivent arriver au bon endroit au bon moment. Pour cela, on est obligé de modifier le contrôleur de mémoire interne et y ajouter un circuit séquentiel qui envoie des ordres aux différents composants de la mémoire dans un ordre déterminé. La complexité de ce circuit séquentiel dépend fortement du nombre d'étapes utilisé pour gérer l'accès mémoire.

Il existe divers types de mémoires pipelinées. Les plus simples sont les **mémoires à flot direct**. Avec celles-ci, la donnée lue ne subit pas de mémorisation dans un registre de sortie.



Avec les **mémoires synchrones registre à registre**, la donnée sortante est mémorisée dans un registre, afin d'augmenter la fréquence de la mémoire et son débit. Le temps de latence des lectures est plus long avec cette technique : il faut ajouter un cycle supplémentaire pour enregistrer la donnée dans le registre, avant de pouvoir la lire. Sur certaines mémoires, le registre en question est séquencé non sur un front, mais par un bit : le temps mis pour écrire dans ce registre est plus faible, ce qui supprime le cycle d'attente supplémentaire.



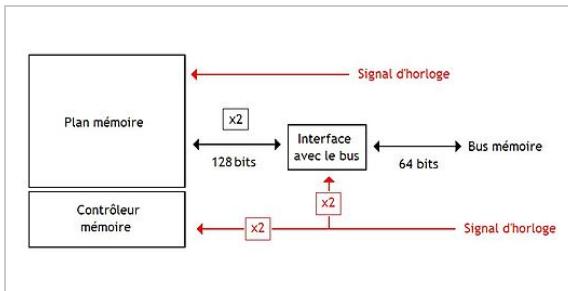
Vu que lectures et écritures n'ont pas forcément le même nombre d'étapes, celles-ci peuvent vouloir accéder au plan mémoire en même temps : ces conflits, qui ont lieu lors d'alternances entre lectures et écritures, sont appelés des retournements de bus. Pour les résoudre, la mémoire doit mettre en attente un des deux accès : le registre d'adresse (et éventuellement le registre d'écriture) est maintenu pour conserver la commande en attente. Ces conflits sont détectés par le contrôleur mémoire.

Une solution pour éviter ces cycles morts est de retarder l'envoi de la donnée à écrire d'un cycle, ce qui donne une **écriture tardive**. Sur les mémoires pour lesquelles l'écriture tardive ne donne pas d'amélioration, l'idéal est de retarder l'écriture de deux cycles d'horloge, et non d'un seul. Cette technique s'appelle l'écriture doublement tardive. Mais si des lectures et écritures consécutives accèdent à la même adresse, la lecture lira une donnée qui n'a pas encore été mise à jour par l'écriture. Pour éviter cela, on doit ajouter un comparateur qui vérifie les deux adresses consécutives : si elles sont identiques, le contenu du registre d'écriture sera envoyé sur le bus au cycle adéquat. Dans le cas des écritures doublement tardives, il faut ajouter deux registres.

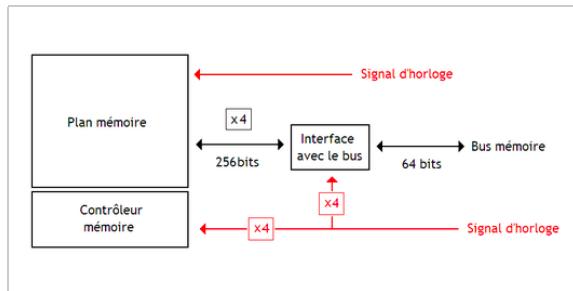
Dual et quad data rate

Nos processeurs sont de plus en plus exigeants, et la vitesse de la mémoire est un sujet primordial. La solution la plus évidente est d'accroître sa fréquence ou la largeur du bus, mais cela aurait pas mal de désavantages : le prix de la mémoire s'envolerait, elle serait bien plus difficile à concevoir, et tout cela sans compter les difficultés pour faire fonctionner l'ensemble à haute fréquence. Le compromis entre ces deux techniques a donné naissance aux **mémoires Dual Data Rate**. Le plan mémoire a une fréquence inférieure à celle du bus, mais a une largeur plus importante :

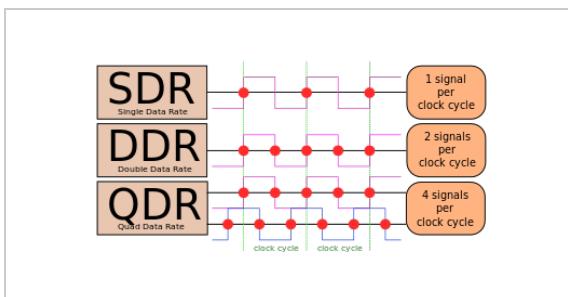
on peut y lire ou y écrire 2, 4, 8 fois plus de données d'un seul coup. Le contrôleur et le bus mémoire fonctionnent à une fréquence multiple de celle du plan mémoire, pour compenser. Il existe aussi des mémoires quad data rate, pour lesquelles la fréquence du bus est quatre fois celle du plan mémoire. Évidemment, la mémoire peut alors lire ou écrire 4 fois de données par cycle que ce que le bus peut supporter.



Mémoire DDR.



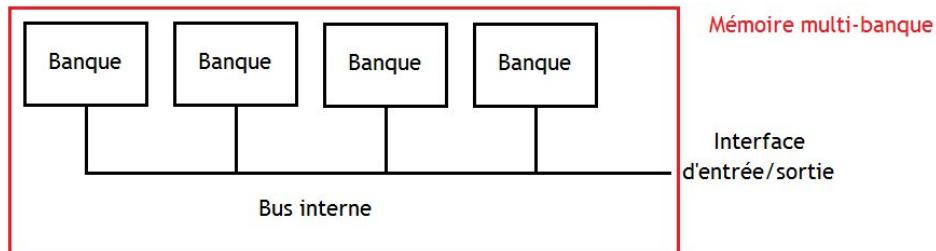
Mémoire QDR.



SDR DDR QDR.

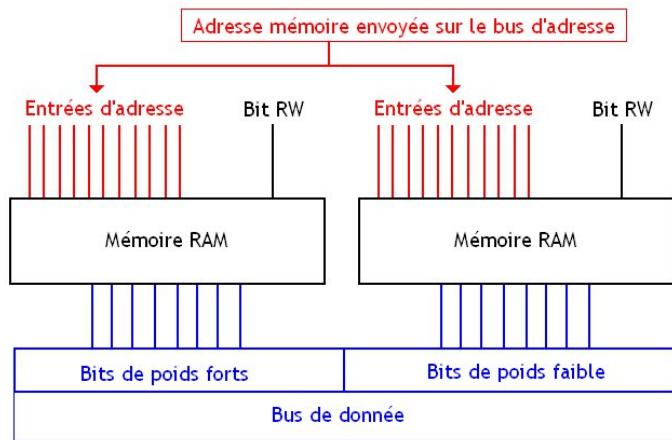
Banques et rangées

Sur certaines puces mémoires, un seul boîtier peut contenir plusieurs mémoires indépendantes regroupées pour former une mémoire unique plus grosse. Découper ainsi une mémoire en mémoires plus petites assemblées dans une seule puce améliore les performances, la consommation d'énergie, et j'en passe. Par exemple, cela permet de faciliter le rafraîchissement d'une mémoire DRAM : on peut rafraîchir chaque sous-mémoire en parallèle, indépendamment des autres. Dans ce qui va suivre, nous allons voir comment assembler plusieurs mémoires simples pour former des mémoires plus grosses, ce qui peut être fait de plusieurs manières : on peut décider de doubler le nombre d'adresses, doubler la taille d'un mot mémoire, ou faire les deux. Dans tous les cas, chaque sous-mémoire indépendante est appelée une **banque**, ou encore un banc mémoire. La mémoire obtenue par combinaison de plusieurs banques est appelée une mémoire multi-banques.



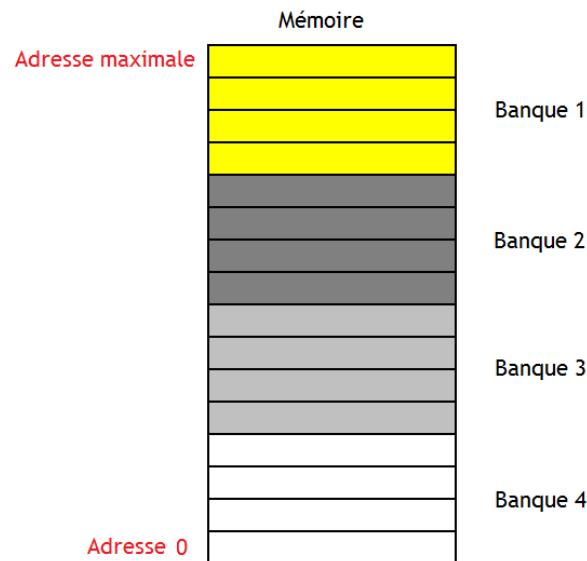
Arrangement horizontal

La première méthode utilise plusieurs banques pour augmenter la taille d'un mot mémoire sans changer le nombre d'adresses : c'est l'**arrangement horizontal**. Chaque banc mémoire contient une partie du mot mémoire final. Avec cette organisation, on accède à tous les bancs en parallèle à chaque accès, avec la même adresse. Pour l'exemple, les barrettes de mémoires SDRAM ou DDR-RAM des PC actuels possèdent un mot mémoire de 64 bits, mais sont en réalité composées de 8 sous-mémoires ayant un mot mémoire de 8 bits. Cela permet de répartir la production de chaleur sur la barrette : la production de chaleur est répartie entre plusieurs puces, au lieu d'être concentrée dans la puce en cours d'accès. La technologie dual-channel est basée sur le même principe. Sauf qu'au lieu de rassembler plusieurs puces mémoires sur une même barrette, on fait la même chose avec plusieurs barrettes de mémoires. Ainsi, on peut connecter deux barrettes avec un mot mémoire de 64 bits et on les relie à un bus de 128 bits.



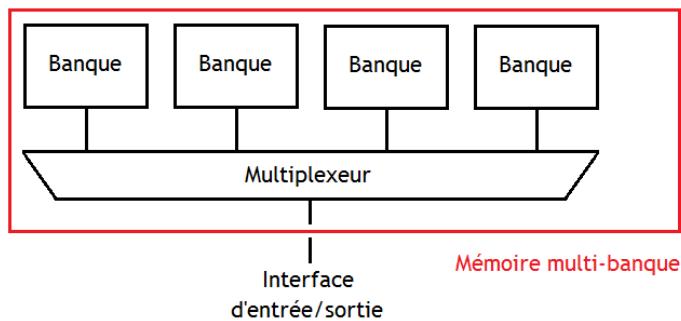
Arrangement vertical

L'**arrangement vertical** rassemble plusieurs boîtiers de mémoires pour augmenter la capacité sans changer la taille d'un mot mémoire. On utilisera un boîtier pour une partie de la mémoire, un autre boîtier pour une autre, et ainsi de suite. Sans aucune optimisation, on utilise les bits de poids forts pour sélectionner la banque, ce qui fait que les adresses sont réparties comme illustré dans le schéma ci-dessous. Un défaut de cette organisation est que, si on souhaite lire/écrire deux mots mémoire consécutifs, on devra attendre que l'accès au premier mot soit fini avant de pouvoir accéder au suivant (vu que ceux-ci sont dans la même banque).



Lors d'une lecture ou écriture, il faut connecter la bonne banque et déconnecter toutes les autres. Pour cela, chaque banque possède une broche supplémentaire nommée CS, qui indique s'il faut connecter ou déconnecter la mémoire du bus. Cette broche est commandée par un décodeur, qui prend les bits de poids forts de l'adresse en entrée. Ainsi, l'adresse est coupée en deux morceaux : un morceau qui détermine quel est le numéro de la banque à activer, et une portion qui adresse le mot mémoire voulu dans la banque.

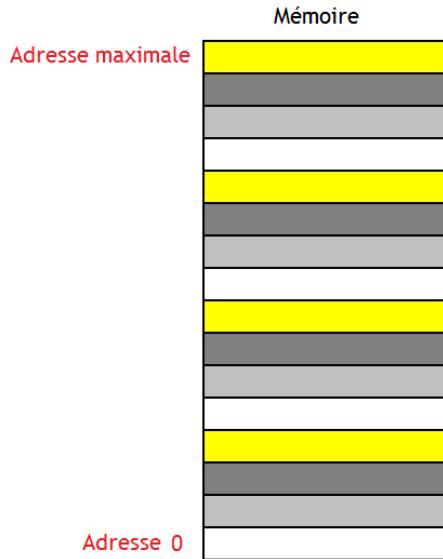
Une autre solution est d'ajouter un multiplexeur en sortie des banques, et de commander celui-ci convenablement avec les bits de poids forts.



Entrelacement interleaved

On peut prendre en compte le fait que chaque banque est indépendante des autres pour optimiser : si une banque est occupée par un accès mémoire, on peut accéder aux autres banques en parallèle. Cela permet de pipeliner les accès mémoire : on peut démarrer une lecture ou écriture dans une banque à chaque cycle, tant qu'une banque est inutilisée. Vu que les accès mémoire se font souvent à des adresses consécutives, c'est une bonne idée de mettre des mots mémoire consécutifs dans des banques différentes. A chaque cycle d'horloge, on peut ainsi accéder à une nouvelle banque, et donc à un nouveau mot mémoire : on peut ainsi pipeliner l'accès à des mots mémoire consécutifs. Pour cela, il suffit de prendre une mémoire à arrangement vertical, avec un petit changement : il faut utiliser les bits de poids faible pour sélectionner la banque : ce sont ce

qu'on appelle des **mémoires interleaved**.



Mémoires à entrelacement par décalage

Mais les mémoires interleaved ont un petit problème : sur une mémoire à N banques, des accès dont les adresses sont séparées par N mots mémoires vont tous tomber dans la même banque et seront donc impossibles à pipeliner. Pour résoudre ce problème, il faut répartir les mots mémoires dans la mémoire autrement. Dans les explications qui vont suivre, la variable N représente le nombre de banques, qui sont numérotées de 0 à N-1. Pour obtenir cette organisation, on va découper notre mémoire en blocs de N adresses. On commence par organiser les N premières adresses comme une mémoire interleaved : l'adresse 0 correspond à la banque 0, l'adresse 1 à la banque 1, etc. Sur une mémoire interleaved, on continuera à assigner des adresses à partir de la fin du bloc. Cette fois-ci, nous allons décaler d'une adresse, et continuer à remplir le bloc suivant. Une fois la fin du bloc atteinte, on finit de remplir le bloc en repartant du début du bloc. Et on poursuit l'assignation des adresses en décalant d'un cran en plus à chaque bloc. Ainsi, chaque bloc verra ses adresses décalées d'un cran en plus comparé au bloc précédent. Si jamais le décalage dépasse la fin d'un bloc, alors on reprend au début.



En faisant cela, on remarque que les banques situées à N adresses d'intervalle sont différentes. Dans l'exemple du dessus, nous avons ajouté un décalage de 1 à chaque nouveau bloc à remplir. Mais on aurait tout aussi bien pu prendre un décalage de 2, 3, etc. Dans tous les cas, on obtient un entrelacement par décalage. Ce décalage est appelé le pas d'entrelacement, noté P. Le calcul de l'adresse à envoyer à la banque, ainsi que la banque à sélectionner se fait en utilisant les formules suivantes :

- adresse à envoyer à la banque = adresse totale / N ;
- numéro de la banque = (adresse + décalage) modulo N, avec décalage = (adresse totale * P) mod N.

Avec cet entrelacement par décalage, on peut prouver que la bande passante maximale est atteinte si le nombre de banques est un nombre premier. Seulement, utiliser un nombre de banques premier peut créer des trous dans la mémoire, des mots mémoires inadressables. Pour éviter cela, il faut faire en sorte que N et la taille d'une banque soient premiers entre eux : ils ne doivent pas avoir de diviseur commun. Dans ce cas, les formules se simplifient :

- adresse à envoyer à la banque = adresse totale / taille de la banque ;
- numéro de la banque = adresse modulo N.

Entrelacement pseudo-aléatoire

Une dernière méthode de répartition consiste à répartir les adresses dans les banques de manière pseudo-aléatoire. La première solution consiste

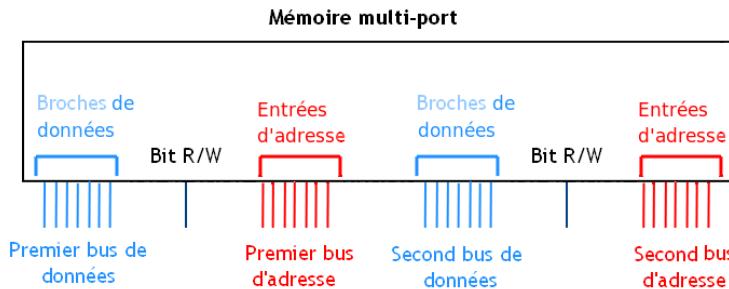
à permuter des bits entre ces champs : des bits qui étaient dans le champ de sélection de ligne vont être placés dans le champ pour la colonne, et vice-versa. Pour ce faire, on peut utiliser des permutations : il suffit d'échanger des bits de place avant de couper l'adresse en deux morceaux : un pour la sélection de la banque, et un autre pour la sélection de l'adresse dans la banque. Cette permutation est fixe, et ne change pas suivant l'adresse. D'autres inversent les bits dans les champs : le dernier bit devient le premier, l'avant-dernier devient le second, etc. Autre solution : couper l'adresse en morceaux, faire un XOR bit à bit entre certains morceaux, et les remplacer par le résultat du XOR bit à bit. Il existe aussi d'autres techniques qui donnent le numéro de banque à partir d'un polynôme modulo N, appliquée sur l'adresse.

Rangées

Si on mélange l'arrangement vertical et l'arrangement horizontal, on obtient ce que l'on appelle une rangée. Sur ces mémoires, les adresses sont découpées en trois morceaux, un pour sélectionner la rangée, un autre la banque, puis la ligne et la colonne.

Mémoires multiports

Les **mémoires multiports** sont reliées non pas à un, mais à plusieurs bus. Chaque bus est connecté sur la mémoire sur ce qu'on appelle un port. Ces mémoires permettent de transférer plusieurs données à la fois, une par port. Le débit est donc supérieur à celui des mémoires mono-port. De plus, chaque port peut être relié à des composants différents, ce qui permet de partager une mémoire entre plusieurs composants. Comme autre exemple, certaines mémoires multiports ont un bus sur lequel on ne peut que lire une donnée, et un autre sur lequel on ne peut qu'écrire.

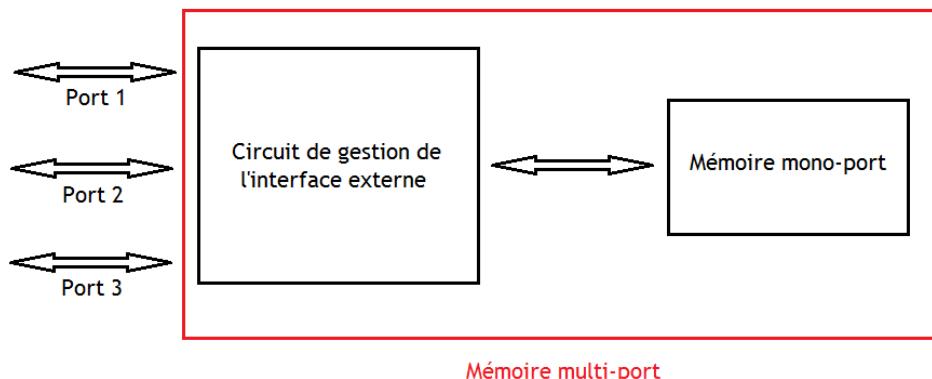


Multiports idéal

Une première solution consiste à créer une mémoire qui soit vraiment multiports. Pour rappel, dans une mémoire normale, chaque cellule mémoire est relié à bitline via un transistor, lui-même commandé par le décodeur. Avec une mémoire multiports, tout est dupliqué. Chaque port aura sa propre bitline, qui lui sera dédié. Pour une mémoire à N ports, chaque cellule sera reliée à N bitlines. Évidemment, cela demande d'ajouter des transistors de sélection, pour la connexion et la déconnexion. De plus, ces transistors sont dorénavant commandés par des décodeurs différents : un par port. En clair, tout est dupliqué, sauf les cellules mémoires : les bitlines sont dupliquées, pareil pour les transistors de connexion/déconnexion, et pareil pour les décodeurs. Et on a autant de duplications que l'on a de ports : N ports signifie tout multiplier par N. Autant dire que ça bouffe du circuit, sans compter la consommation énergétique ! Cette solution pose toutefois un problème : que se passe-t-il lorsque des ports différents écrivent simultanément dans la même cellule mémoire ? Eh bien tout dépend de la mémoire : certaines donnent des résultats plus ou moins aléatoires et ne sont pas conçues pour gérer de tels accès, d'autres mettent en attente un des ports lors de l'accès en écriture. Sur ces dernières, il faut évidemment rajouter des circuits pour détecter les accès concurrents et éviter que deux ports se marchent sur les pieds.

Multiports externe

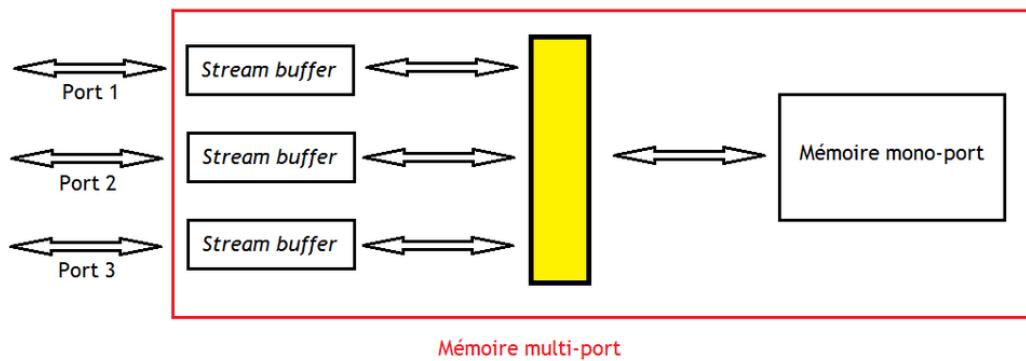
Certaines mémoires multiports sont fabriquées à partir d'une mémoire à un seul port, couplée à des circuits pour faire l'interface avec chaque port.



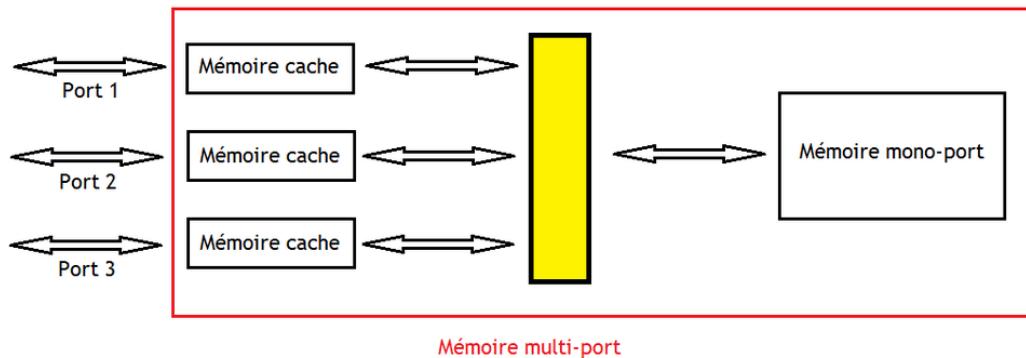
Une première méthode pour concevoir ainsi une mémoire multiports est d'augmenter la fréquence de la mémoire mono-port sans toucher à celle du bus. A chaque cycle d'horloge interne, un port a accès au plan mémoire.

La seconde méthode est basée sur des **stream buffers**. Elle fonctionne bien avec des accès à des adresses consécutives. Dans ces conditions, on peut tricher en lisant ou en écrivant plusieurs blocs à la fois dans la mémoire interne monoport : la mémoire interne a un port très large, capable de lire ou d'écrire une grande quantité de données d'un seul coup. Mais ces données ne pourront pas être envoyées sur les ports de lecture ou reçues via les ports d'écritures, nettement moins larges. Pour la lecture, il faut obligatoirement utiliser un circuit qui découpe les mots mémoire lus depuis la mémoire interne en données de la taille des ports de lecture, et qui envoie ces données une par une. Et c'est la même chose pour les ports d'écriture, si ce n'est que les données doivent être fusionnées pour obtenir un mot mémoire complet de la RAM interne.

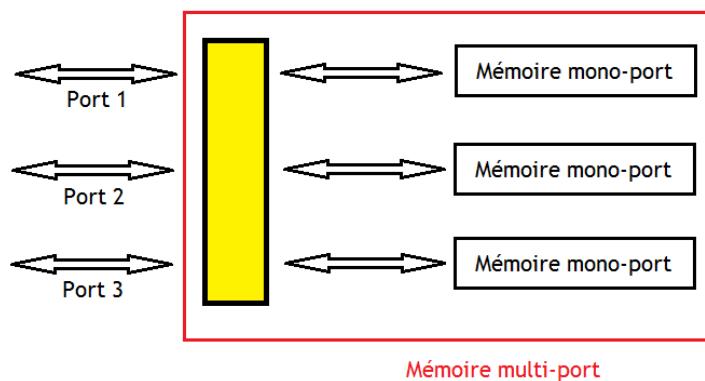
Pour cela, chaque port se voit attribuer une mémoire qui met en attente les données lues ou à écrire dans la mémoire interne : le stream buffer. Si le transfert de données entre RAM interne et stream buffer ne prend qu'un seul cycle, ce n'est pas le cas pour les échanges entre ports de lecture et écriture et stream buffer : si le mot mémoire de la RAM interne est n fois plus gros que la largeur d'un port de lecture/écriture, il faudra envoyer le mot mémoire en n fois, ce qui donne n^2 cycles. Ainsi, pendant qu'un port accèdera à la mémoire interne, les autres ports seront occupés à lire le contenu de leurs stream buffers. Ces stream buffers sont gérés par des circuits annexes, pour éviter que deux stream buffers accèdent en même temps dans la mémoire interne.



La troisième méthode remplace les stream buffers par des caches, et utilise une mémoire interne qui ne permet pas de lire ou d'écrire plusieurs mots mémoires d'un coup. Ainsi, un port pourra lire le contenu de la mémoire interne pendant que les autres ports seront occupés à lire ou écrire dans leurs caches.



La méthode précédente peut être améliorée, en utilisant non pas une seule mémoire monoport en interne, mais plusieurs banques monoports. Dans ce cas, il n'y a pas besoin d'utiliser de mémoires caches ou de stream buffers : chaque port peut accéder à une banque tant que les autres ports n'y touchent pas. Évidemment, si deux ports veulent lire ou écrire dans la même banque, un choix devra être fait et un des deux ports devra être mis en attente.



Multiports à état partagé

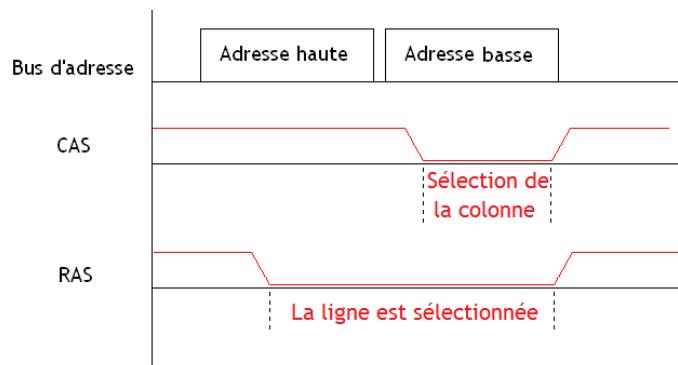
Certaines mémoires ont besoin d'avoir un très grand nombre de ports de lecture. Pour cela, on peut utiliser une mémoire multiparts à état dupliqué. Au lieu d'utiliser une seule mémoire de 20 ports de lecture, le mieux est d'utiliser 4 mémoires qui ont chacune 5 ports de lecture. Toutefois, ces quatre mémoires possèdent exactement le même contenu, chacune d'entre elles étant une copie des autres : toute donnée écrite dans une des mémoires l'est aussi dans les autres. Comme cela, on est certain qu'une donnée écrite lors d'un cycle pourra être lue au cycle suivant, quel que soit le port, et quelles que soient les conditions.

Mémoires DDR, SDRAM et leurs cousins

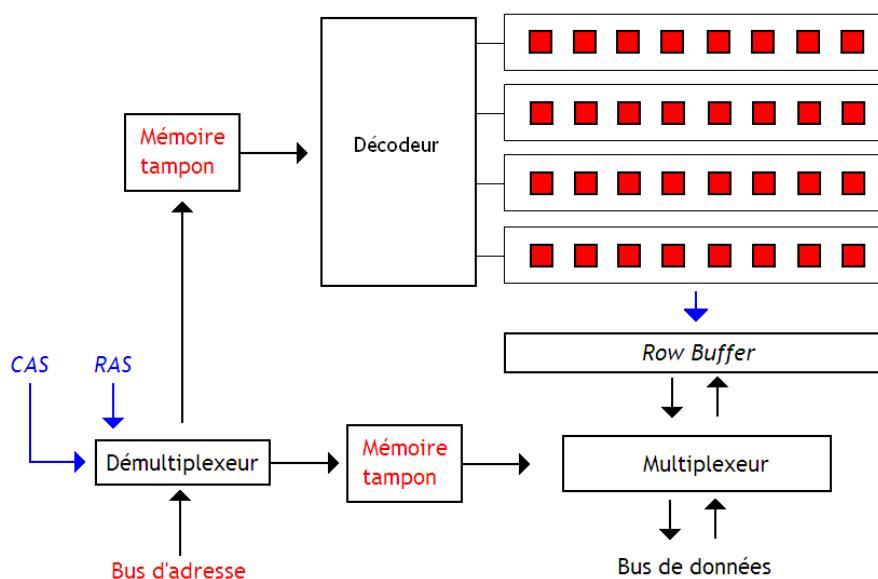
Les mémoires actuelles sont un peu plus complexes que les mémoires vues dans les chapitres précédents. Ce chapitre va vous expliquer dans les grandes lignes en quoi nos mémoires actuelles se démarquent des autres. On verra que les mémoires modernes ne sont que des améliorations des mémoires vues précédemment.

RAM asynchrones

Avant l'invention des mémoires SDRAM et DDR, il existait un grand nombre de mémoires différentes, les plus connues étant les mémoires fast page mode et EDO-RAM. Ces mémoires n'étaient pas synchronisées avec le processeur via une horloge. Quand ces mémoires ont été créées, cela ne posait aucun problème : les accès mémoire étaient très rapides et le processeur était certain que la mémoire aurait déjà fini sa lecture ou écriture au cycle suivant. Ces mémoires asynchrones étaient des mémoires à adressage par coïncidence ou à row buffer, avec une adresse découpée en deux : une adresse haute pour sélectionner la ligne, et une adresse basse qui sélectionne la colonne. Cette adresse est envoyée en deux fois : la ligne, puis la colonne. Pour savoir si une donnée envoyée sur le bus d'adresse est une adresse de ligne ou de colonne, le bus de commande de ces mémoires contenait deux fils bien particuliers : les RAS et le CAS. Pour simplifier, le signal RAS permettait de sélectionner une ligne, et le signal CAS permettait de sélectionner une colonne.

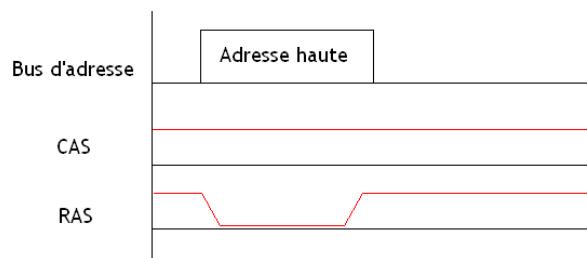


On remarque que la mémoire va prendre en compte les signaux RAS et CAS quand ils passent de 1 à 0 : c'est à ce moment là que la ligne ou colonne dont l'adresse est sur le bus sera sélectionnée. Tant que des signaux sont à zéro, la ligne ou colonne reste sélectionnée : on peut changer l'adresse sur le bus, cela ne désélectionnera pas la ligne ou la colonne et la valeur présente lors du front descendant est conservée.



Rafraîchissement mémoire

Les mémoires FPM et EDO doivent être rafraîchies régulièrement. Au début, le rafraîchissement se faisait ligne par ligne.

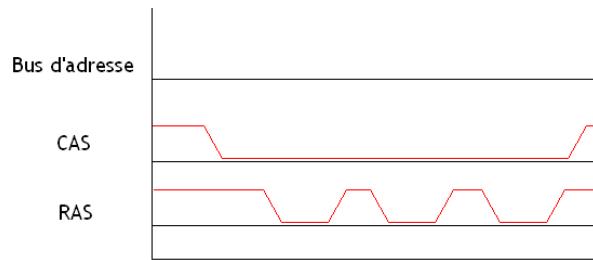


Par la suite, certaines mémoires ont implémenté un compteur interne d'adresse, pour déterminer la prochaine adresse à rafraîchir sans la préciser sur le bus d'adresse. Mais le déclenchement du rafraîchissement se faisait par une commande externe, provenant du contrôleur mémoire ou du

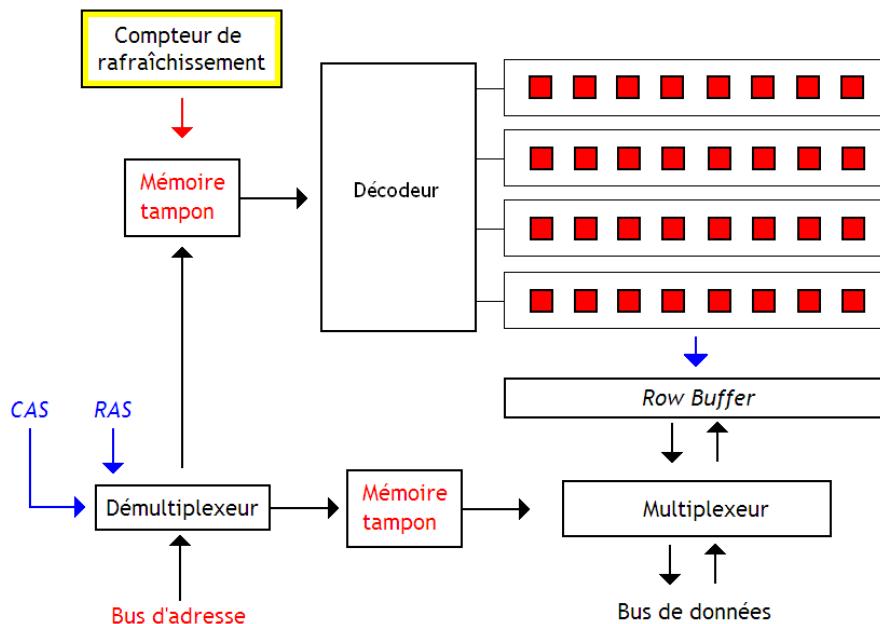
processeur. Cette commande faisait passer le CAS à 0 avant le RAS.



On peut noter qu'il est possible de déclencher plusieurs rafraîchissements à la suite en laissant le signal CAS dans le même état. Ce genre de choses pouvait avoir lieu après une lecture : on pouvait profiter du fait que le CAS soit mis à zéro par la lecture ou l'écriture pour ensuite effectuer des rafraîchissements en touchant au signal RAS. Dans cette situation, la donnée lue était maintenue sur la sortie durant les différents rafraîchissements.

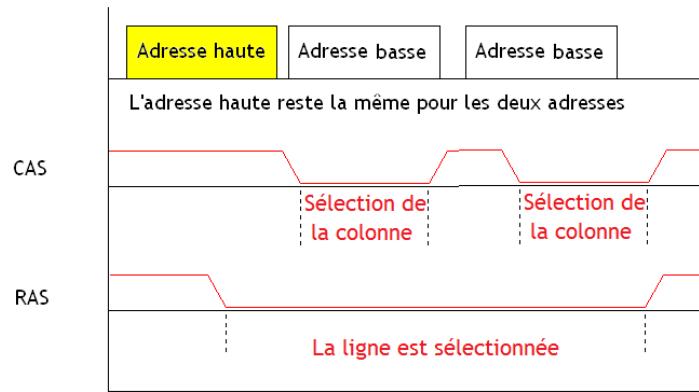


Rapidement, les constructeurs de mémoire se sont dits qu'il valait mieux gérer ce rafraîchissement de façon automatique, sans faire intervenir le contrôleur mémoire intégré à la carte mère. Ce rafraîchissement a alors été délégué au contrôleur mémoire intégré à la barrette de mémoire, et est maintenant géré par des circuits spécialisés. Ce circuit de rafraîchissement automatique n'est rien d'autre qu'un compteur, qui contient un numéro de ligne (celle à rafraîchir).

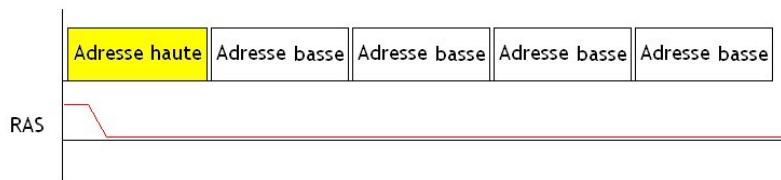


Mémoires FPM et EDO

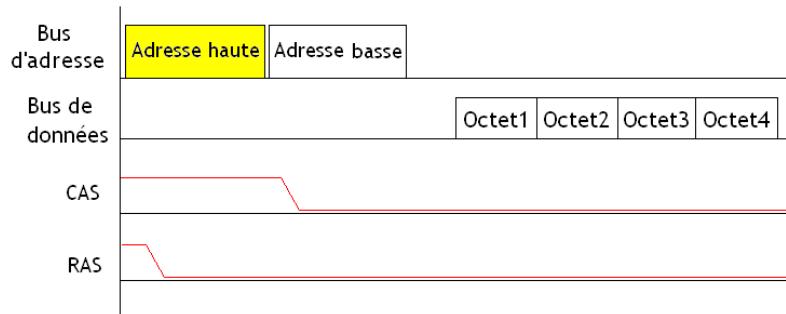
Les mémoires FPM et EDO fonctionnaient de façon asynchrone, avec une amélioration comparée aux autres mémoires. Il n'y avait plus besoin de préciser deux fois la ligne si celle-ci ne changeait pas lors de deux accès consécutifs : on pouvait garder la ligne sélectionnée durant plusieurs accès.



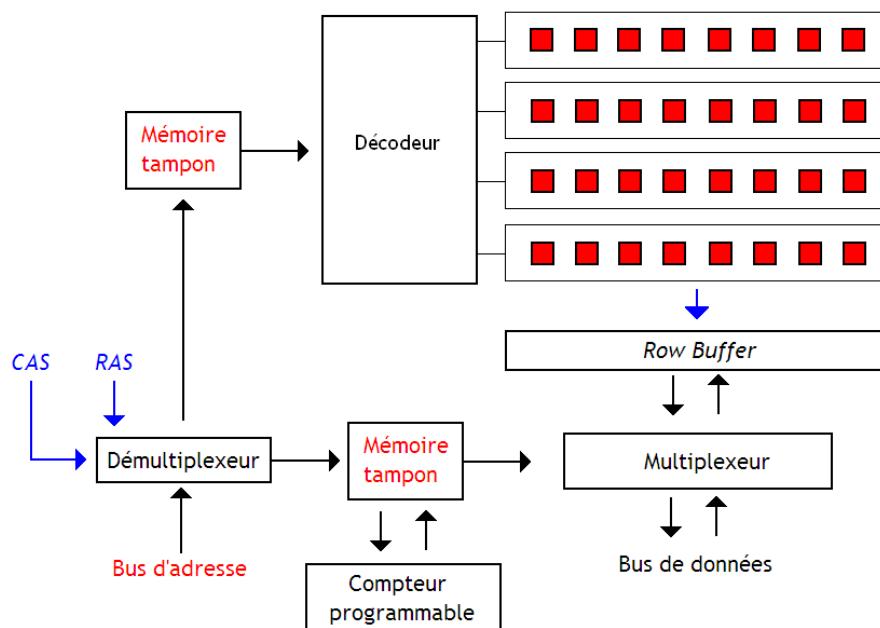
Certaines FPM se passaient du signal CAS : le changement de l'adresse de colonne était détecté automatiquement par la mémoire et suffisait à passer à la colonne suivante. Ces mémoires sont dites à colonne statique. Dans ces conditions, un délai supplémentaire a fait son apparition : le tCAS-to-CAS. C'est le temps minimum entre deux sélections de deux colonnes différentes.



L'EDO-RAM a été inventée quelques années après la mémoire FPM. Elle a été déclinée en deux versions : la EDO simple, et la EDO en rafale. L'EDO simple n'apportait que de faibles améliorations vraiment mineures, aussi je me permets de la passer sous silence. Les EDO en rafale effectuent les accès à 4 octets consécutifs automatiquement : il suffit d'adresser le premier octet à lire. Les 4 octets étaient envoyés sur le bus les uns après les autres, au rythme d'un par cycle d'horloge : ce genre d'accès mémoire s'appelle un accès en rafale.



Implémenter cette technique nécessite d'ajouter des circuits dans notre mémoire. Il faut notamment rajouter un compteur, capable de faire passer d'une colonne à une autre quand on lui demande. Le tout était accompagné de quelques circuits pour gérer le tout.



SDRAM

Les mémoires asynchrones ont laissé la place aux mémoires SDRAM, qui sont synchronisées avec le bus par une horloge. L'utilisation d'une horloge a comme avantage des temps d'accès fixes : le processeur sait qu'un accès mémoire prendra un nombre déterminé de cycles d'horloge et peut faire ce qu'il veut dans son coin durant ce temps. Avec les mémoires asynchrones, le processeur ne pouvait pas prévoir quand la donnée serait disponible et ne faisait rien tant que la mémoire n'avait pas répondu : il exécutait ce qu'on appelle des wait states en attendant que la mémoire ait fini.

Mode rafale

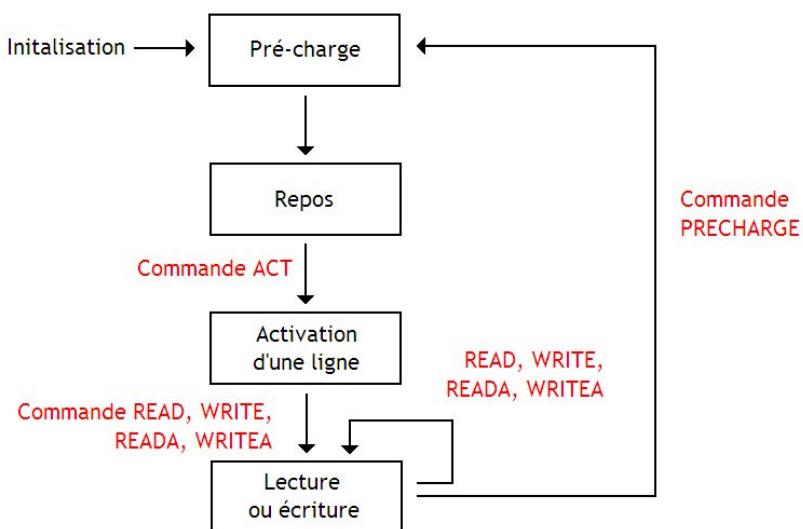
Certains paramètres de la mémoire, qui ont trait au mode rafale, sont modifiables : le contrôleur mémoire interne de la SDRAM mémorise ces informations dans un registre de 10 bits, le registre de mode. Cela permet notamment d'avoir un mode rafale programmable. Pour cela, le registre de mode contient un bit qui permet de préciser s'il faut effectuer des accès normaux ou des accès en rafale. On peut aussi spécifier le nombre d'octets consécutifs à lire ou écrire : on peut ainsi accéder à 1, 2, 4, ou 8 octets en une seule fois, alors que les EDO ne permettaient que des accès à 4 octets consécutifs.

Qui plus est, il existe deux types d'accès en rafale sur les SDRAM : l'accès entrelacé, et l'accès séquentiel. Le mode séquentiel est le mode rafale normal : on accède à des octets consécutifs les uns après les autres. Le mode entrelacé utilise un ordre différent. Avec ce mode de rafale, le contrôleur mémoire effectue un XOR bit à bit entre un compteur (incrémenté à chaque accès) et l'adresse de départ pour calculer la prochaine adresse de la rafale.

Un accès en rafale parcourt un mot (un bloc de mots mémoires de même taille que le bus). Un accès en rafale ne commence pas forcément au début du mot, mais peut commencer à n'importe quel mot mémoire dans le mot. Dans ce cas, la rafale reprend au premier mot mémoire une fois arrivé au bout d'un mot.

Commandes SDRAM

Une SDRAM est commandée par un bus aux fils d'une utilité bien précise. On trouve un signal d'horloge, pour cadencer la mémoire, mais pas que. En tout, 18 fils permettent de commander la SDRAM. Configurer le bus permet d'envoyer des commandes à la mémoire, commandes qui vont effectuer une lecture, une écriture, ou autre chose dans le genre. Le fonctionnement simplifié d'une SDRAM peut se résumer dans ce diagramme :



On remarque que les commandes READ et WRITE ne peuvent se faire qu'une fois que la banque a été activée par une commande ACT. Une fois la banque activée par une commande ACT, il est possible d'envoyer plusieurs commandes READ ou WRITE successives. Ces lectures ou écritures accéderont à la même ligne, mais à des colonnes différentes. Ces commandes se font à partir de l'état de repos, l'état où toutes les banques sont préchargées. Il faut donc noter que les commandes MODE REGISTER SET et AUTO REFRESH ne peuvent se faire que si toutes les banques sont désactivées.

Délais mémoires

Il faut un certain temps pour sélectionner une ligne ou une colonne, sans compter qu'une SDRAM doit gérer d'autres temps d'attente plus ou moins bien connus : ces temps d'attente sont appelés des délais mémoires. La façon de mesurer ces délais varie : sur les mémoires FPM et EDO, on les mesure en unités de temps (secondes, millisecondes, micro-secondes, etc.), tandis qu'en les mesure en cycles d'horloge sur les mémoires SDRAM.

Timing	Description
tRAS	Temps mis pour sélectionner une ligne.
tCAS	Temps mis pour sélectionner une colonne.
tRP	Temps mis pour réinitialiser le row buffer et décharger la ligne.
tRCD	Temps entre la fin de la sélection d'une ligne, et le moment où l'on peut commencer à sélectionner la colonne.
tWTR	Temps entre une lecture et une écriture consécutives.
tCAS-to-CAS	Temps minimum entre deux sélections de deux colonnes différentes.

Mémoires DDR

Les mémoires SDRAM récentes sont des mémoires de type dual data rate, voire quad data rate (voir le chapitre précédent pour ceux qui ont oublié) : elles portent ainsi le nom de mémoires DDR. Plus précisément, le plan mémoire des DDR est deux fois plus large que le bus mémoire, même si les deux sont commandés par un même signal d'horloge : là où les transferts avec le plan mémoire ont lieu sur front montant, les transferts de données sur le bus ont lieu sur les fronts montants et descendants de l'horloge. Il y a donc deux transferts de données sur le bus pour chaque cycle d'horloge, ce qui permet de doubler le débit sans toucher à la fréquence. D'autres différences mineures existent entre les SDRAM et les mémoires DDR. Par exemple, la tension d'alimentation des mémoires DDR est plus faible que pour les SDRAM.

Les mémoires DDR sont standardisées par un organisme international, le JEDEC, et ont été déclinées en versions DDR1, DDR2, DDR3, et DDR4. Il existe enfin d'autres types de mémoires DDR, non-standardisées par le JEDEC : les mémoires GDDR, pour graphics double data rate, utilisées

presque exclusivement sur les cartes graphiques. Il en existe plusieurs types pendant que j'écris ce tutoriel : GDDR, GDDR2, GDDR3, GDDR4, et GDDR5. Mais attention, il y a des différences avec les DDR normales : les GDDR sont des mémoires multiports et elles ont une fréquence plus élevée que les DDR normales, avec des temps d'accès plus élevés (sauf pour le tCAS).

Il existe quatre types de mémoires DDR1 officialisés par le JEDEC.

Nom standard	Nom des modules	Fréquence du bus	Débit	Tension d'alimentation
DDR 200	PC-1600	100 Mhz	1,6 gibioctets seconde	2,5 Volts
DDR 266	PC-2100	133 Mhz	2,1 gibioctets seconde	2,5 Volts
DDR 333	PC-2700	166 Mhz	2,7 gibioctets seconde	2,5 Volts
DDR 400	PC-3200	200 Mhz	3,2 gibioctets seconde	2,6 Volts

Avec les mémoires DDR2, 5 types de mémoires sont officialisées par le JEDEC. Diverses améliorations ont été apportées sur les mémoires DDR2 : la tension d'alimentation est notamment passée de 2,5/2,6 Volts à 1,8 Volts.

Nom standard	Nom des modules	Fréquence du bus	Débit
DDR2 400	PC2-3200	100 Mhz	3,2 gibioctets par seconde
DDR2 533	PC2-4200	133 Mhz	4,2 gibioctets par seconde
DDR2 667	PC2-5300	166 Mhz	5,3 gibioctets par seconde
DDR2 800	PC2-6400	200 Mhz	6,4 gibioctets par seconde
DDR2 1066	PC2-8500	266 Mhz	8,5 gibioctets par seconde

Avec les mémoires DDR3, 6 types de mémoires sont officialisées par le JEDEC. Diverses améliorations ont été apportées sur les mémoires DDR3 : la tension d'alimentation est notamment passée à 1,5 Volts.

Nom standard	Nom des modules	Fréquence du bus	Débit
DDR3 800	PC2-6400	100 Mhz	6,4 gibioctets par seconde
DDR3 1066	PC2-8500	133 Mhz	8,5 gibioctets par seconde
DDR3 1333	PC2-10600	166 Mhz	10,6 gibioctets par seconde
DDR3 1600	PC2-12800	200 Mhz	12,8 gibioctets par seconde
DDR3 1866	PC2-14900	233 Mhz	14,9 gibioctets par seconde
DDR3 2133	PC2-17000	266 Mhz	17 gibioctets par seconde

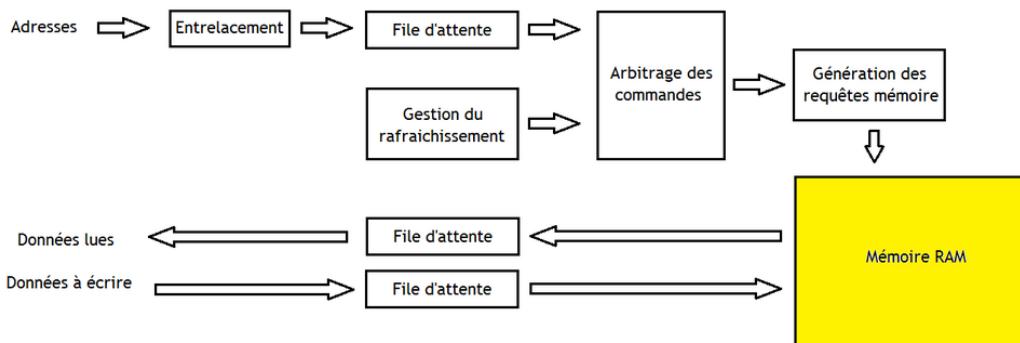
Contrôleur mémoire externe

Pour interfaçer le processeur à la mémoire, on utilise un circuit spécialisé : le contrôleur mémoire externe. Celui-ci est placé sur la carte mère ou dans le processeur, et ne doit pas être confondu avec le contrôleur mémoire intégré dans la mémoire. Ce contrôleur mémoire externe gère :

- l'entrelacement ;
- le rafraîchissement mémoire ;
- le séquencement des accès mémoires ;
- la traduction des accès mémoires en suite de commandes (ACT, sélection de ligne, etc.) ;
- l'interfaçage électrique.

Architecture du contrôleur

Ce contrôleur mémoire externe est schématiquement composé de quatre modules séparés. Le **module d'interface avec le processeur** gère l'entrelacement. Le **module d'ordonnancement des commandes** contrôle le rafraîchissement et le séquencement des accès mémoires (dans le cas où ils ne sont pas envoyés dans l'ordre à la mémoire). Ce module est composé de plusieurs circuits. Il contient notamment des FIFO pour conserver l'ordre des transferts de données, ainsi qu'une mémoire pour mettre en attente les accès mémoires. Il possède aussi un circuit d'arbitrage, qui décide quelle requête envoyer à la mémoire. Enfin, n'oublions pas de mentionner le contrôleur de rafraîchissement. Le **module de décodage des commandes** traduit les accès mémoires en une suite de commandes ACT, READ, PRECHARGE, etc. C'est ce module qui tient compte des délais mémoires et gère l'envoi des commandes au cycle près. Le **module d'interface mémoire** sert quand la mémoire et le processeur n'utilisent pas les mêmes tensions pour coder un bit : ce module se charge de la conversion de tension. Il prend aussi en charge la détection et la correction des erreurs.



Si la mémoire (et donc le contrôleur) est partagée entre plusieurs processeurs, le module d'interface hôte est dupliqué en autant d'exemplaires qu'il y a de processeurs. Le module de commande contient un circuit pour que les processeurs ne se marchent pas sur les pieds pour l'accès à la mémoire. Ce circuit fait en sorte que chaque processeur ait sa part, et puisse accéder à la mémoire sans trop d'attente : cela évite qu'un processeur monopolise la mémoire pour lui tout seul.

Politique de gestion du row buffer

Le contrôleur mémoire décide quand envoyer les commandes PRECHARGE, qui pré-chargent les bitlines et vident le row buffer. Il peut gérer cet envoi des commandes PRECHARGE de diverses manières nommées respectivement politique de la page fermée, politique de la page ouverte et politique hybride.

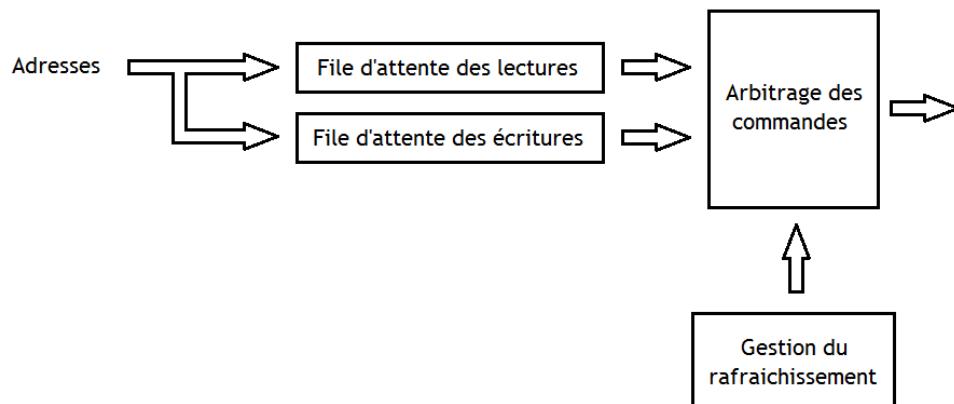
Politique de la page fermée

Dans le premier cas, le contrôleur ferme systématiquement toute ligne ouverte par un accès mémoire : chaque accès est suivi d'une commande PRECHARGE. Cette méthode est adaptée à des accès aléatoires, mais est peu performante pour les accès à des adresses consécutives. On appelle cette méthode la close page autoprecharge, ou encore politique de la page fermée.

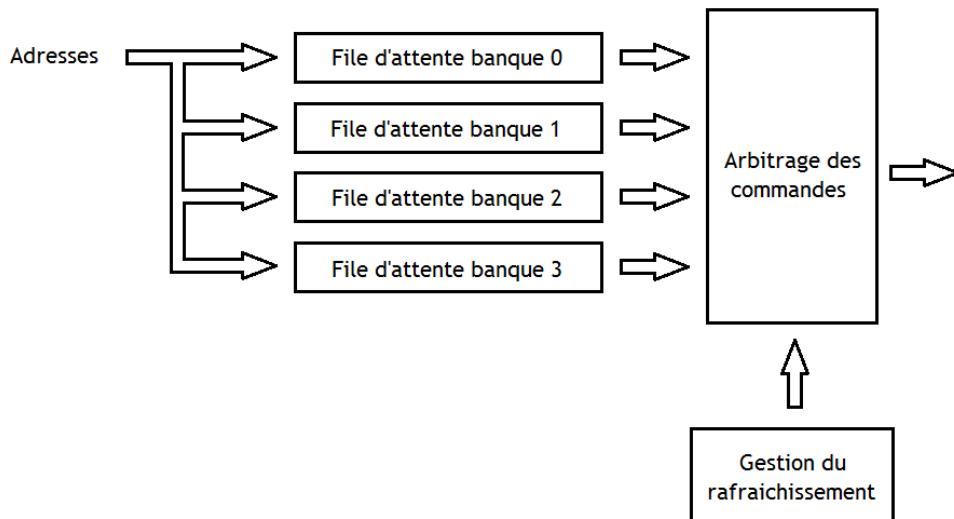
Politique de la page ouverte

Avec des accès consécutifs, les données ont de fortes chances d'être placées sur la même ligne. En conséquence, fermer celle-ci pour la réactiver au cycle suivant est contreproductif : il vaut mieux garder la ligne active et ne la fermer que lors d'un accès à une autre ligne. Cette politique porte le nom d'open page autoprecharge, ou encore politique de la page ouverte. Lors d'un accès, la commande à envoyer peut faire face à deux situations : soit la nouvelle requête accède à la ligne ouverte, soit elle accède à une autre ligne. Dans le premier cas, on doit juste changer de colonne : c'est un row buffer hit. Le temps nécessaire pour accéder à la donnée est donc égal au temps nécessaire pour sélectionner une colonne avec une commande READ, WRITE, WRITA, READA : on observe donc un gain significatif comparé à la politique de la page fermée. Dans le second cas, c'est un row buffer miss et il faut procéder comme avec la politique de la page fermée, à savoir vider le row buffer avec une commande PRECHARGE, sélectionner la ligne avec une commande ACT, avant de sélectionner la colonne avec une commande READ, WRITE, WRITA, READA. Déceler un row buffer miss ou un row buffer hit n'est pas très compliqué. Le contrôleur a juste à se souvenir des lignes et banques actives avec une petite mémoire : la table des banques. Pour détecter un hit ou un miss, le contrôleur doit simplement extraire la ligne de l'adresse à lire ou écrire, et vérifier si celle-ci est ouverte : si c'est le cas, c'est un hit, un miss sinon.

Le contrôleur mémoire peut optimiser l'utilisation de la politique de la page ouverte en changeant l'ordre des requêtes mémoires pour regrouper les accès à la même ligne/banque. Une première solution consiste à mettre en attente les écritures et effectuer les lectures en priorité. Il suffit d'utiliser des files d'attente séparées pour les lectures et écritures. Si une lecture accède à une donnée qui n'a pas encore été écrite dans la mémoire (car mise en attente), la donnée est lue directement dans la file d'attente des écritures. Cela demande de comparer toute adresse à lire avec celles des écritures en attente : la file d'attente est donc une mémoire associative.



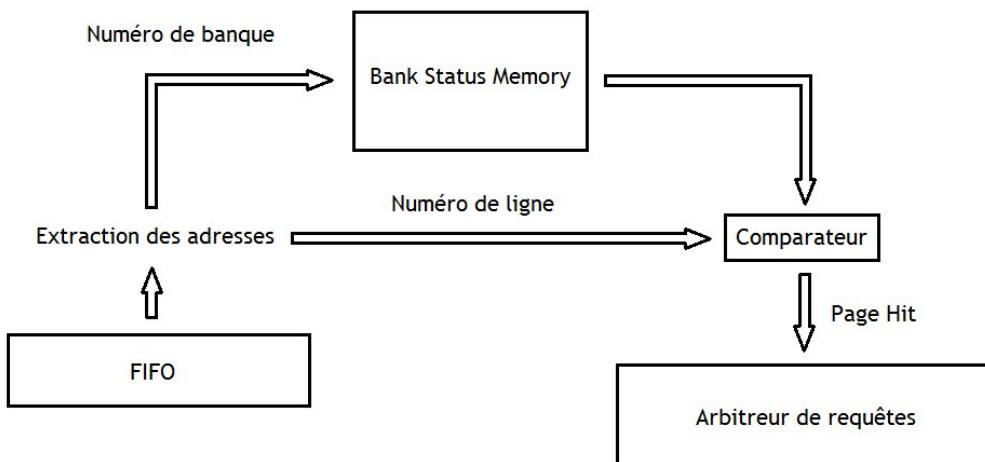
Une autre solution répartit les accès mémoire sur plusieurs banques en parallèle. Ainsi, on commence par servir la première requête de la banque 0, puis la première requête de la banque 1, et ainsi de suite. Cela demande de trier les requêtes de lecture ou d'écriture suivant la banque de destination : on se retrouve avec une file d'attente pour chaque banque.



Une autre solution consiste à regrouper plusieurs accès à des données successives en un seul accès en rafale.

Politiques dynamiques

Pour gagner en performances et diminuer la consommation énergétique de la mémoire, il existe des techniques hybrides qui alternent entre la politique de la page fermée et la politique de la page ouverte en fonction des besoins. La plus simple décide s'il faut maintenir ouverte une ligne accédée récemment. Il suffit de regarder si les accès mémoires mis en attente vont accéder à la page : la ligne est laissée ouverte si au moins un accès en attente y accède. Une autre politique laisse la ligne ouverte, sauf si un accès en attente accède à une ligne différente de la même banque. Avec l'algorithme FR-FCFS (First Ready, First-Come First-Service), les accès mémoires qui accèdent à une ligne ouverte sont exécutés en priorité, et les autres sont mis en attente. Dans le cas où aucun accès mémoire ne lit une ligne ouverte, le contrôleur prend l'accès le plus ancien, celui qui attend depuis le plus longtemps comparé aux autres. Pour implémenter ces techniques, le contrôleur compare la ligne ouverte dans le row buffer et les lignes des accès en attente. Ces techniques demandent de mémoriser la ligne ouverte, pour chaque banque, dans une mémoire RAM : la bank status memory. Le contrôleur extrait les numéros de banque et de ligne des accès en attente pour adresser la bank status memory, le résultat étant comparé avec le numéro de ligne de l'accès.



Politiques prédictives

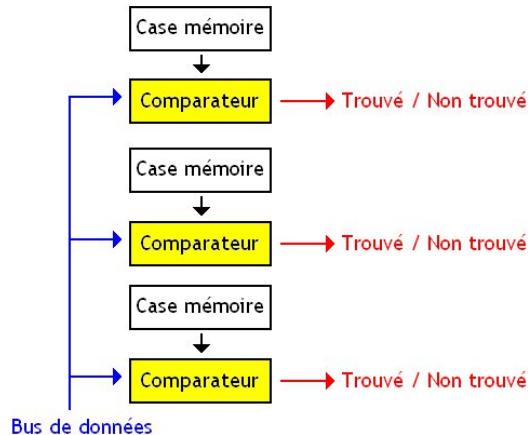
Certaines techniques prédisent s'ils faut fermer ou laisser ouvertes les pages ouvertes. La méthode la plus simple consiste à laisser ouverte chaque ligne durant un temps pré-déterminé avant de la fermer. Une autre solution consiste à effectuer ou non la pré-charge en fonction du type d'accès mémoire effectué par le dernier accès : on peut très bien décider de laisser la ligne ouverte si l'accès mémoire précédent était une rafale, et fermer sinon. Une autre solution consiste à mémoriser les N derniers accès et en déduire s'il faut fermer ou non la prochaine ligne. On peut mémoriser si l'accès en question a causé la fermeture d'une ligne avec un bit : mémoriser les N derniers accès demande d'utiliser un simple registre à décalage, un registre couplé à un décaleur par 1. Pour chaque valeur de ce registre, il faut prédire si le prochain accès demandera une ouverture ou une fermeture. Une première solution consiste à faire la moyenne des bits à 1 dans ce registre : si plus de la moitié des bits est à 1, on laisse la ligne ouverte et on ferme sinon. Pour améliorer l'algorithme, on peut faire en sorte que les bits des accès mémoires les plus récents aient plus de poids dans le calcul de la moyenne. Mais rien de descendant. Une autre technique consiste à détecter les cycles d'ouverture et de fermeture de page potentiels. Pour cela, le contrôleur mémoire associe un compteur pour chaque valeur du registre. En cas de fermeture du row buffer, ce compteur est décrémenté, alors qu'une non-fermeture va l'incrémenter : suivant la valeur de ce compteur, on sait si l'accès a plus de chance de fermer une page ou non.

Les mémoires associatives

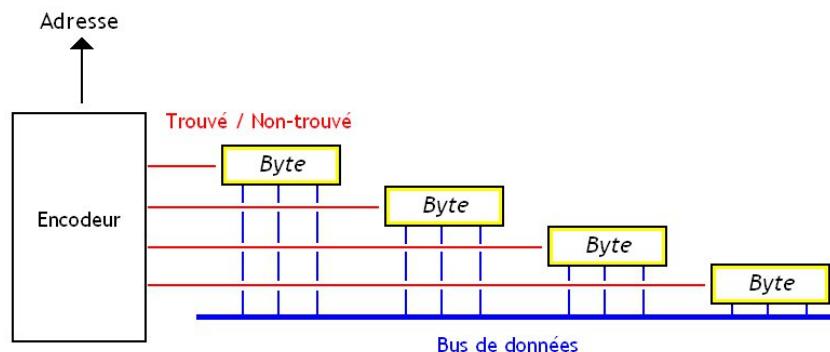
Les mémoires associatives servent à accélérer la recherche d'une donnée dans un ensemble de données. Leur fonctionnement est totalement opposé aux mémoires adressables normales : au lieu d'envoyer l'adresse pour accéder à la donnée, on envoie la donnée pour récupérer son adresse. Évidemment, il se peut que la donnée demandée ne soit pas présente en mémoire, qui doit préciser si elle a trouvé la donnée recherchée avec un signal dédié. Si la donnée est présente en plusieurs exemplaires en mémoire, la mémoire renvoie le premier exemplaire rencontré (celui dont l'adresse est la plus petite). Certaines mémoires renvoient aussi les autres exemplaires, qui sont envoyées une par une au processeur.

Mémoires associatives

Le plan mémoire d'un mémoire associative a deux fonctions : mémoriser des mots mémoire et vérifier la présence d'une donnée dans chaque case mémoire (effectuer une comparaison, donc). Dans une mémoire associative, la donnée à rechercher dans la mémoire est envoyée à toutes les cases mémoires simultanément et toutes les comparaisons sont effectuées en parallèle.

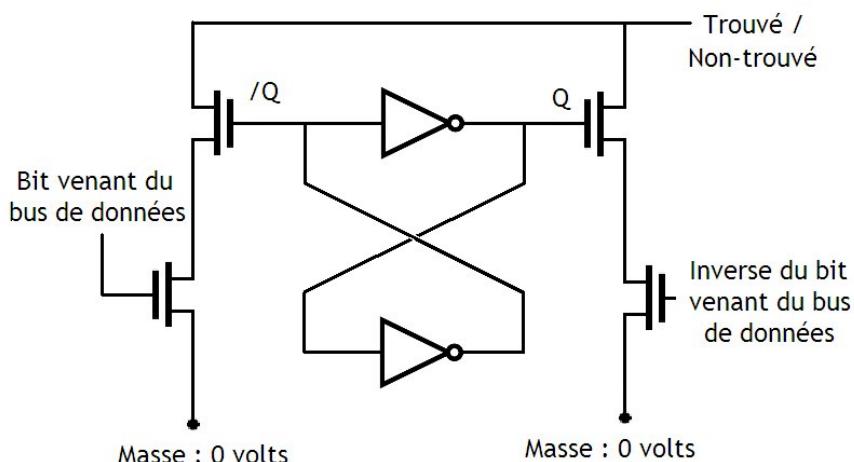


Une fois la case mémoire qui contient la donnée identifiée, il faut déduire son adresse. Si vous regardez bien, le problème qui est posé est l'exact inverse de celui qu'on trouve dans une mémoire adressable : on n'a pas l'adresse, mais les signaux sont là. La traduction va donc devoir se faire par un circuit assez semblable au décodeur : l'**encodeur**. Dans le cas où la mémoire doit gérer le cas où plusieurs mots contiennent la donnée demandée, la solution la plus simple est d'en sélectionner un seul, généralement celui qui a l'adresse la plus petite (ou la plus grande). Pour cela, l'encodeur doit subir quelques modifications et devenir un encodeur à priorité.

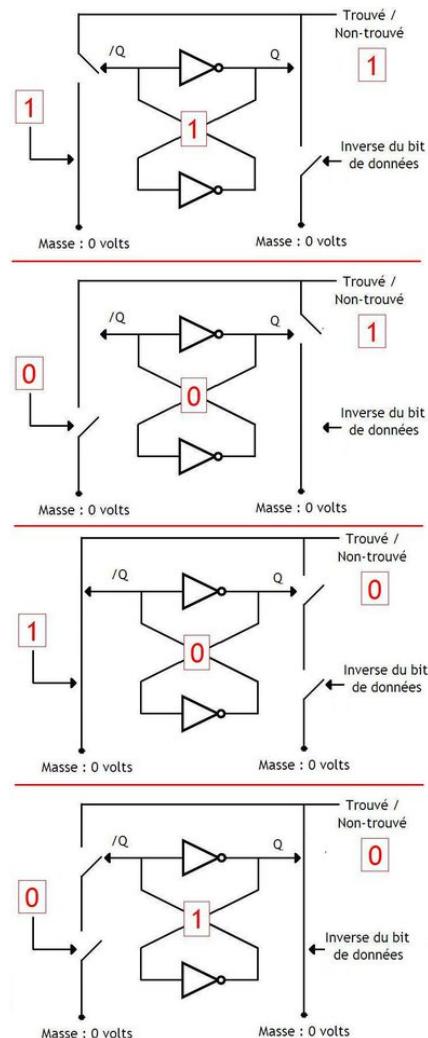


Cellules NOR

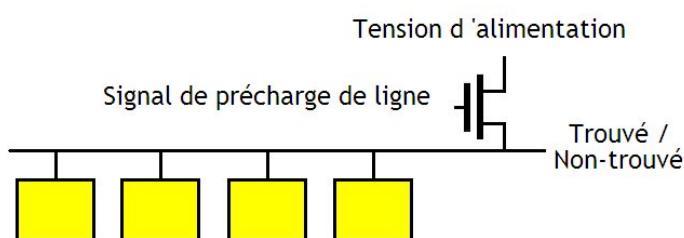
Pour limiter le nombre de portes logiques utilisées pour le comparateur, les circuits de comparaison sont fusionnés avec les cellules mémoires. La première possibilité d'intégration est celle-ci :



Pour comprendre son fonctionnement, il suffit de se rappeler que les transistors se ferment quand on place un 1 sur la grille. De plus, le signal trouvé est mis à 1 par un stratagème technique si les transistors du circuit ne le connectent pas à la masse. Ce qui donne ceci :

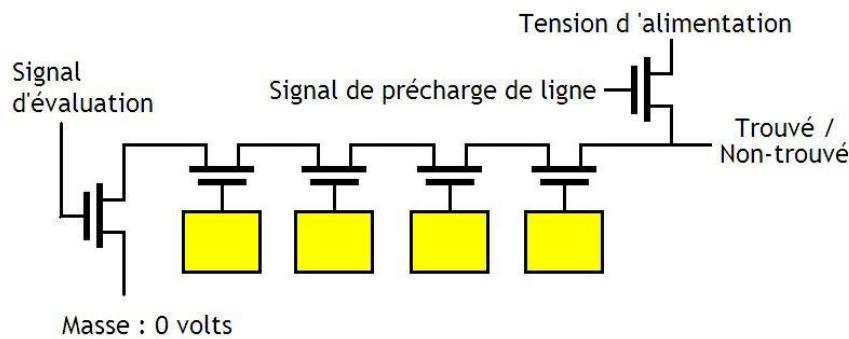


Les signaux trouvé/non-trouvé des différents bits d'un seul mot sont alors reliés ensemble comme ci-dessous. Le transistor du dessus sert à mettre notre signal trouvé/non-trouvé de notre mot mémoire à 1. Il s'ouvre durant un moment afin de précharger le fil qui transportera le signal. Le signal sera relié au zéro volt si une seule case mémoire a une sortie à zéro. Dans le cas contraire, le fil contiendra encore une tension correspondant à un 1 sur la sortie.

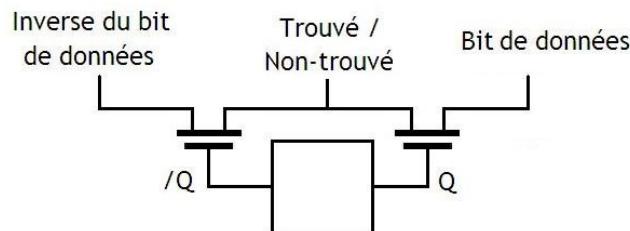


Cellule NAND

Une autre manière consiste à relier les cellules d'un mot mémoire ensemble comme indiqué dans le schéma ci-dessous. Chaque cellule mémoire va avoir un signal trouvé/non-trouvé inversé : il vaut 0 quand le bit de la cellule et le bit du bus de données sont identiques, et vaut 1 sinon. Si toutes les cellules sont identiques aux bits correspondants du bus de données, le singal final sera mis à la masse (à zéro). Et inversement, il suffit qu'une seule cellule ouvre son transistor pour que le signal soit relié à la tension d'alimentation. En somme, le signal obtenu vaut zéro si jamais la donnée et le contenu du mot sont identiques, et 1 sinon.



Il est alors possible de câbler la cellule mémoire d'une autre façon, indiquée sur le schéma ci-dessous :

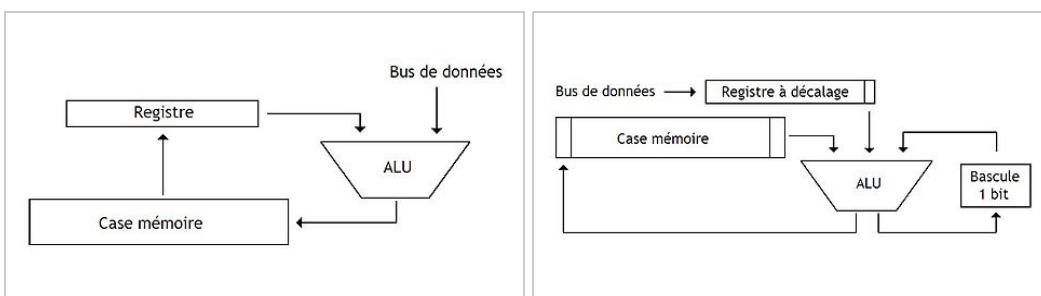


Architectures associatives

Il est possible d'utiliser une mémoire associative en supplément d'une RAM normale, mais aussi comme mémoire principale sur une architecture Harvard. Toutefois, ces architectures ne sont pas les seules à utiliser des mémoires associatives. Il est maintenant temps de voir les **processeurs associatifs**, des sortes de mémoires associatives sous stéroïdes. Sur une mémoire associative, on peut comparer la donnée passée en entrée avec chaque mot mémoire. Avec un processeur associatif, les mots sélectionnés par une comparaison vont rester sélectionnés durant un ou plusieurs cycles. Pendant ce temps, on pourra demander à notre processeur d'effectuer des calculs ou des branchements dessus. En gros, on peut effectuer des opérations identiques sur un grand nombre de mots en parallèle.

Encore mieux : il est possible pour notre mémoire de sélectionner les bits à prendre en compte dans une comparaison, et de laisser tranquille les autres. Pour cela, il suffit d'envoyer à la mémoire un **masque**, qui permettra de savoir quels bits sont à prendre en compte dans chaque donnée : si son i-ème bit est à 1, le bit correspondant de la donnée est à prendre en compte dans notre comparaison. Ce masque est envoyé à la mémoire via un second bus, séparé du bus de données.

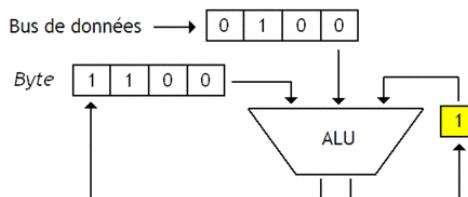
On peut signaler qu'il existe plusieurs types de processeurs associatifs. Le premier type correspond à ce qu'on appelle les processeurs **totalement parallèles**, où l'ALU est capable de traiter tous les bits du mot d'un seul coup. Pour économiser des circuits, on peut castrer l'ALU qui ne peut plus traiter qu'un seul bit à la fois : on rentre alors dans la catégorie des architecture dites **sérielles**. Pour pouvoir traiter un bit à la fois, le mot mémoire est implémenté avec un registre à décalage.



Case mémoire d'un processeur associatif totalement parallèle.

Case mémoire d'un processeur associatif bit serial avec une bascule.

Pour donner un exemple d'opération sur une architecture bit-séquentielle, je vais prendre l'exemple d'un test d'égalité entre un Byte qui contient la valeur 1100, et une donnée qui vaut 0100. Pour effectuer ce test, notre processeur va devoir comparer chaque bit un par un en utilisant une porte NXOR, tout en prenant en compte le bit indiquant le résultats des comparaisons précédentes avec une porte ET. Le résultat de la comparaison est disponible dans notre bascule de 1 bit une fois la comparaison terminée. Notre bascule sera donc reliée à la sortie sur laquelle on envoie le signal Trouvé / Non-trouvé. Mais cette bascule peut aussi servir dans d'autres opérations. Par exemple, si notre processeur implémente des opérations d'addition, notre addition peut se faire bit par bit. Notre bascule sert alors à stocker la retenue de l'addition des deux bits précédents.



Les disques durs

Nous allons maintenant étudier la plus célèbre des mémoires de masse : le disque dur ! L'intérieur d'un disque dur contient beaucoup de composants, le premier d'entre eux étant le **support de mémorisation**. Il est composé de plateaux fabriqués dans une couche de matériau magnétique. Ensuite, on trouve une **tête de lecture-écriture** pour lire ou écrire sur le support de mémorisation. Ensuite, on trouve le **contrôleur de disque** qui gère le déplacement de la tête de lecture. Et enfin, on trouve un circuit qui se charge de gérer les échanges de données entre le disque dur et le bus : le **contrôleur de bus**.

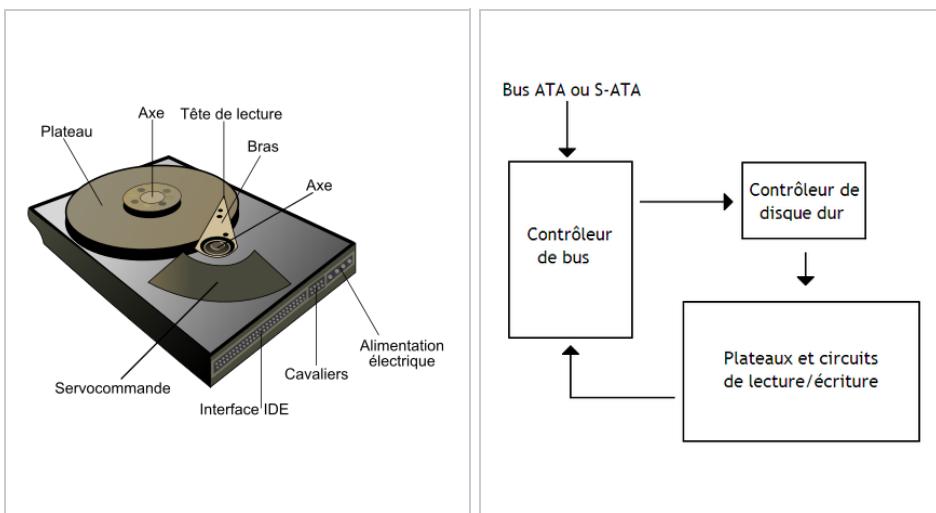
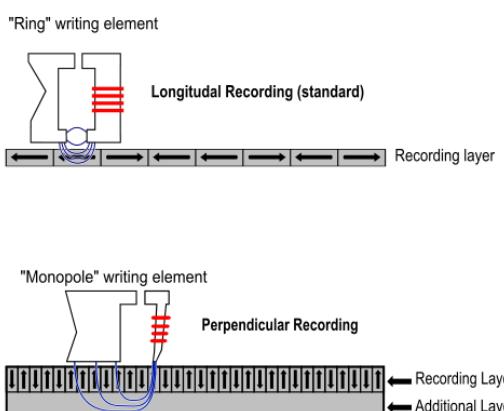


Schéma de l'intérieur d'un disque dur.

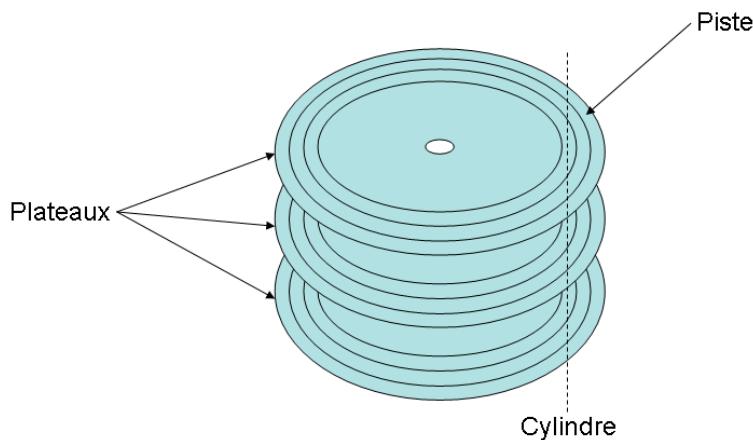
Schéma de l'intérieur d'un disque dur, version conceptuelle.

Plateaux

Les **plateaux** sont composés d'une plaque fabriquée dans un matériau peu magnétisable, recouvert de deux couches de matériau magnétique : une couche sur chaque face. Ainsi, les deux faces d'un plateau sont utilisées pour stocker des données. Sur les anciens disques durs, chaque couche de matériau magnétique est découpée en cellules de un bit. Le stockage d'un bit se fait en aimantant la cellule dans une direction pour stocker un 1 et dans l'autre sens pour un 0. Les nouveaux disques durs utilisent deux cellules pour stocker un bit. Si ces deux cellules sont aimantées dans le même sens, c'est un zéro, et sinon, c'est un 1. Certains aimantent ces blocs à la verticale, et d'autres à l'horizontale, les disques durs récents utilisant l'aimantation verticale pour diverses raisons techniques. Les disques durs basés sur ce principe permettent de stocker plus de données à surface égale.



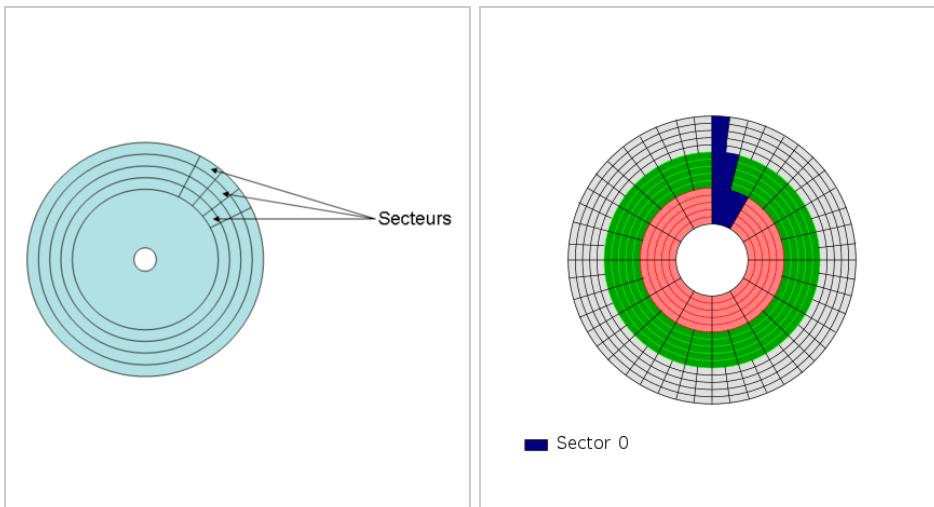
Ces plateaux entourent un axe central autour duquel les plateaux vont tourner à une vitesse précise. Plus la vitesse de rotation des plateaux est forte, plus le disque dur sera rapide. Les disquettes fonctionnent sur un principe semblable à celui du disque dur, à une différence près : il n'y a qu'un seul plateau. Les bits sont regroupés sur une face d'un plateau en cercles concentriques qu'on nomme des **pistes**. Les pistes d'une face d'un plateau sont numérotées, mais attention : deux pistes peuvent avoir le même numéro si celles-ci sont sur des faces ou des plateaux différents. Ces pistes de même numéro sont à la verticale les unes des autres : elles forment ce qu'on appelle un **cylindre**.



Ces pistes sont découpées en blocs de taille fixe qu'on appelle des **secteurs**, qui correspondent aux bytes des disques durs. Ces secteurs sont numérotés. Le début de chaque secteur est identifié par un préambule, qui permet de délimiter le secteur sur une piste, suivi des données du secteur proprement dit, puis de bits de correction d'erreur.

Préambule	Données	ECC
Contenu d'un secteur		

Le découpage des pistes en secteurs dépend du disque dur. Les anciens disques durs découpaient les pistes en secteurs de taille différentes, de sorte que toutes les pistes aient le même nombre de secteurs. De nos jours, on préfère utiliser des secteurs de taille fixe, quitte à ce que toutes les pistes aient un nombre de secteurs différent. Ces deux organisations ont cependant un problème lors de la lecture de deux pistes consécutives. Le passage d'une piste à l'autre prend toujours un peu de temps : il faut bien déplacer la tête de lecture. Or, durant ce temps, le disque dur a tourné et on doit attendre que le disque dur finisse son tour avant de retomber sur le bon secteur. Pour résoudre ce problème, les concepteurs de disque dur décalent les secteurs de même numéro dans le sens de rotation du disque. C'est ce que l'on appelle le **cylinder skewing**.



Organisation avec nombre de secteurs par piste constant.

Organisation avec taille constante des secteurs.

Têtes de lecture-écriture

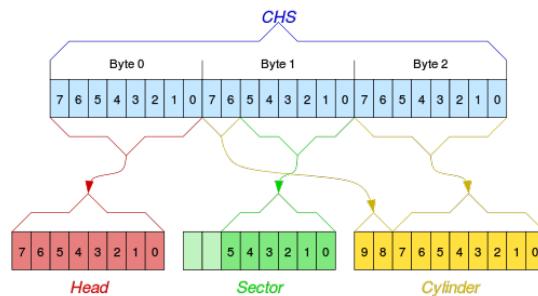
Un disque dur contient de petits dispositifs capables de lire ou écrire un bit sur le plateau : les **têtes de lecture-écriture**. Généralement, on trouve une tête de lecture sur chaque face, parfois plus (ce qui permet de lire ou d'écrire à des secteurs différents en même temps). Chaque tête de lecture-écriture est un bras mécanique dans lequel passe un fil électrique. Ce fil électrique affleure légèrement au bout du bras en formant un électroaimant qui va servir à lire ou écrire sur le plateau. Pour écrire, il suffit d'envoyer un courant électrique dans le fil de notre tête de lecture : cela créera un champ magnétique autour de l'électroaimant, qui aimera le plateau. Pour lire, il suffira d'approcher la tête de la cellule : le champ magnétique de la cellule va créer une tension dans l'électroaimant et le fil, qu'on pourra interpréter comme un 0 ou un 1.

Ces têtes de lecture sont entraînées par un moteur qui les fait tourner au-dessus des plateaux, afin de les placer au dessus des données à lire ou écrire. À l'arrêt, les têtes de lecture sont rangées bien sagement dans un emplacement bien particulier : pas question de les arrêter sur place ! Si une tête de lecture-écriture touche la couche magnétique, la surface de contact est définitivement endommagée.

Le disque dur contient deux moteurs : un pour déplacer les têtes de lecture-écriture, et un autre pour faire tourner les plateaux. Quand les moteurs tournent, cela fait un peu de bruit, ce qui explique que votre disque dur "gratte" quand il est soumis à une forte charge de travail. Pour limiter ce bruit, certains fabricants utilisent une technologie nommée **automatic acoustic management** ou AAM, qui rend le déplacement des têtes de lecture plus progressif. Cela peut avoir un impact sur les performances : la tête de lecture met plus de temps à se mettre en place, augmentant le temps d'accès à un secteur. On peut configurer le lissage de l'accélération via un nombre codé sur 8 bits (seules celles comprises entre 128 et 254 sont permises). Cette valeur est stockée dans une mémoire EEPROM sur le disque dur, et est chargée à chaque démarrage du disque dur. Pour configurer cette valeur, le pilote de disque dur peut envoyer une commande au contrôleur, qui mettra à jour la valeur d'AAM.

Contrôleur de disque

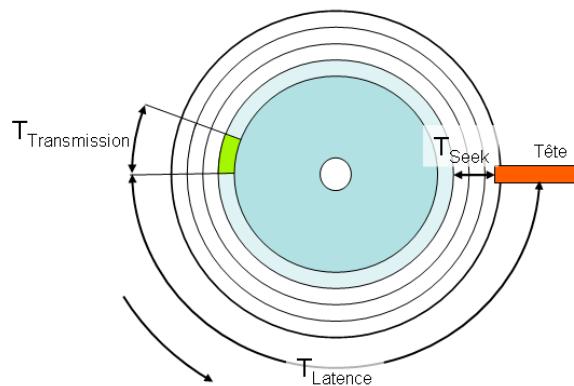
Outre la gestion des moteurs, le contrôleur de disque dur doit aussi gérer l'adressage des secteurs. Avec l'**adressage CHS**, l'adresse est composée du numéro de plateau, du numéro du cylindre et le numéro du secteur. En comparaison, l'**adressage LBA** numérote chaque secteur sans se préoccuper de son numéro de tête, de cylindre ou de secteur.



Le contrôleur de disque est aussi relié à toute une série de capteurs, qui permettent de mesurer en temps réel ce qui se passe sur le disque dur. Tout disque dur possédant la **technologie SMART** peut ainsi détecter en temps réel les erreurs, qui sont alors mémorisées dans une ROM accessible par différents logiciels. Elle est utilisée pour savoir si le disque dur est pas loin de lâcher ou s'il lui reste du temps avant sa mort. Parmi ces erreurs, il arrive que certains secteurs du disque deviennent inutilisables, à la suite d'un défaut mécanique. Pour résoudre tout problème lié à ces secteurs, le contrôleur de disque mémorise la liste des secteurs défectueux dans une mémoire, généralement une EEPROM.

Performance des disques durs

La performance des disques durs dépend de leur temps d'accès et de son débit. Le temps d'accès d'un secteur dépend de plusieurs sous-temps d'accès. Tout d'abord, il faut que la tête de lecture se déplace sur la piste adéquate : c'est le **seek time**. Ensuite, il faut que la tête se positionne au-dessus du secteur à lire/écrire, ce qui prend un certain temps appelé **temps de balayage**. Puis, la lecture/écriture du secteur prend en certain temps, appelé **temps de transmission**.



Le temps de balayage et le temps de transmission dépendent fortement de la vitesse de rotation du disque dur : plus le plateau tourne vite, plus vite le secteur se positionnera sous la tête de lecture et plus vite le secteur sera balayé par celle-ci. Le temps de transmission dépend aussi de la taille des secteurs : plus un secteur est grand, plus la tête devra rester longtemps sur la surface magnétique pour lire le secteur au complet. Pour le seek time, celui-ci dépend surtout du rayon du disque, lui-même proportionnel au nombre de pistes par plateau. La vitesse de rotation permet de calculer le temps que met la tête de lecture pour balayer une piste complète, aussi appelé **temps de rotation**. Le temps de balayage et de transmission sont bornés par ce temps de rotation. Dans le pire des cas, si le secteur est vraiment mal placé, la somme temps de balayage + temps de lecture/écriture est égale au temps de rotation. Le temps d'accès moyen est plus ou moins égal à la moitié du temps de rotation, sans optimisations particulières.

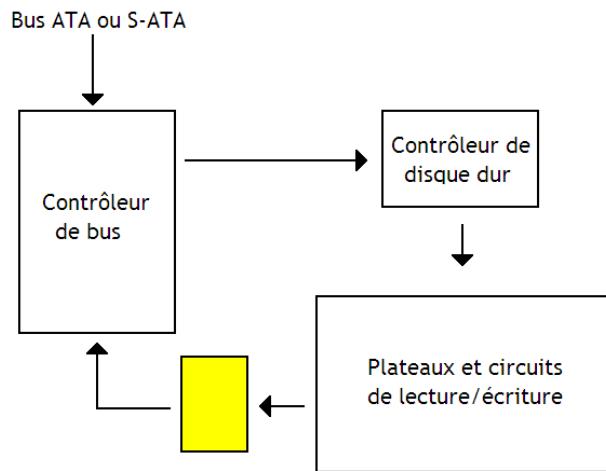
Vitesse de rotation en tours par minutes Temps de balayage, en millisecondes

15,000	2
10,000	3
7,200	4.16
5,400	5.55
4,800	6.25

Tout HDD contient diverses optimisations pour limiter l'influence de ces temps d'accès sur la performance. Le but de ces optimisations peut être de diminuer le temps d'accès moyen, ou d'augmenter le débit. Pour rappel, le débit est le produit de la taille des secteurs par le nombre de lectures/écritures de secteurs par secondes. Augmenter le débit demanderait d'augmenter soit la taille des secteurs, soit de gérer plus de lectures/écritures. Diminuer le temps d'accès demanderait de jouer sur le mouvement des têtes de lecture ou d'éviter des accès au disque. Les optimisations suivantes servent généralement à cela.

Tampon de lecture/écriture

Le disque dur est plus lent que le processeur et la RAM, les transferts entre eux étant rarement synchronisés. Il arrive souvent que, lorsqu'un disque dur a terminé de lire une donnée, le processeur ne soit pas disponible immédiatement. De même, le processeur peut déclencher une écriture alors que le disque dur n'est pas prêt (parce qu'il est en pleine lecture, par exemple). Pour faciliter cette synchronisation, les disques durs modernes incorporent des mémoires tampons pour mettre en attente les lectures ou écritures, le temps que le processeur ou le disque soient disponibles. Ces tampons sont souvent des mémoires FIFO qui accumulent les données lues, ou à écrire.

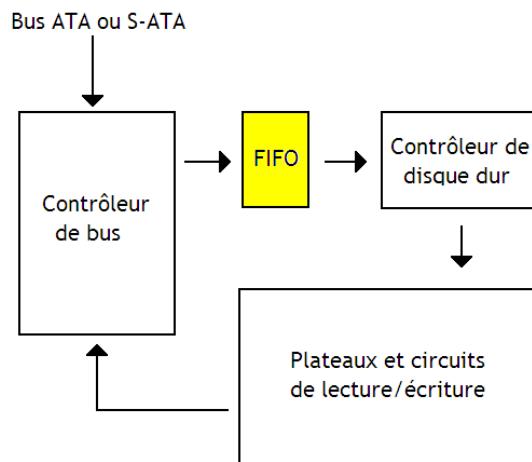


Cache disque

De nos jours, le tampon de lecture/écriture est parfois remplacé par une mémoire cache. Ce cache peut mémoriser les données lues récemment, pour profiter de la localité temporelle, ou précharger les secteurs proches de ceux récemment accédés, pour profiter de la localité spatiale (sur certains disques durs, c'est carrément toute la piste qui est chargé lors de l'accès à un secteur). Autre utilité de ce cache : mettre en attente les écritures tant que la tête de lecture-écriture n'est pas libre. Ce cache en écriture pose un léger problème. Les données à écrire vont attendre durant un moment dedans avant que les plateaux soient libres pour démarrer l'écriture. Si jamais une coupure de courant se produit, les données présentes dans la mémoire tampon, mais pas encore écrites sur le disque dur sont perdues.

Tampon de requêtes

Sur les disques durs anciens, on devait attendre qu'une requête soit terminée avant d'en envoyer une autre, ce qui limitait le nombre de requêtes d'entrée-sortie par seconde. Pour limiter la casse, certains disques durs reçoivent des requêtes même si les précédentes ne sont pas terminées. Ces requêtes anticipées sont mises en attente dans une mémoire tampon et sont traitées quand le disque dur est libre.



Avec cette optimisation, le nombre d'IOPS augmente, en même temps que le temps d'accès augmente. Le nombre d'IOPS augmente pour la simple raison que le disque dur peut en mettre en attente les opérations d'entrée-sortie en attendant leur traitement. Mais le temps d'attente dans le tampon de requêtes est techniquement compté dans le temps d'accès, en plus du temps de transmission et de rotation. Le temps d'accès total augmente donc, au prix d'un débit plus élevé. Plus la file d'attente sera longue (peut mettre en attente un grand nombre de requêtes), plus le débit sera important, de même que le temps d'accès. Il est possible de formaliser l'amélioration induite par ce tampon de requêtes sur le débit et le temps d'accès avec quelques principes mathématiques relativement simples.

Native Command and Queuing

Les disques durs S-ATA récents dotés de la bonne carte-mère permettent de changer l'ordre de traitement des requêtes : au lieu de faire sans cesse des allers et retours, notre disque dur peut tenter d'accéder de préférence à des données proches. Cela s'appelle du **Native Command Queuing**. Cette technique permet de diminuer le temps d'accès moyen, en économisant sur le temps de recherche et de balayage. Cette technique demande toutefois de remettre en ordre les données lues ou écrites. Pour cela, on utilise les mémoires tampons vues précédemment, pour que les données quittent le disque dur dans l'ordre demandé par l'OS.

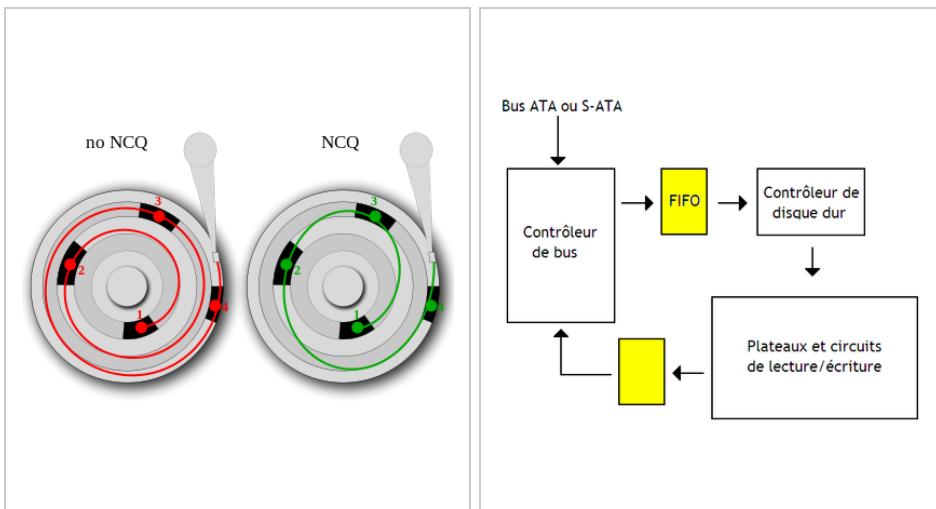


Illustration du principe du NCQ.

Mémoires tampons internes au disque dur, dans le cadre du NCQ.

On pourrait croire que chercher à toujours accéder au secteur le plus proche serait une bonne idée. Mais en faisant ainsi, certaines situations peuvent poser problème. Si de nouvelles requêtes sur des secteurs proches arrivent sans cesse, le disque dur les traite systématiquement avant les anciennes. Pour éviter cela, les requêtes sont traitées dans le sens de rotation du bras de la tête de lecture. Si celle-ci se dirige vers l'extérieur du disque dur, celui-ci traite les requêtes qui lui demandent d'aller lire ou écrire les secteurs sur la même piste ou encore plus vers l'extérieur. Et réciproquement si la tête de lecture se dirige vers l'intérieur. Elle ne change de sens que quand toutes les requêtes en attente portent sur des secteurs situés plus à l'intérieur. D'autres algorithmes d'ordonnancement des requêtes d'accès disque existent, mais faire une liste serait beaucoup trop long.

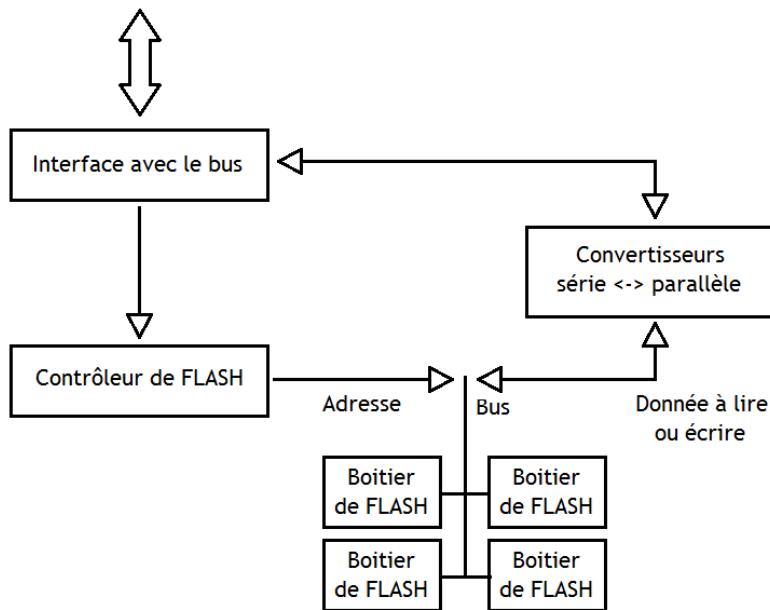
Consommation énergétique des disques durs

La majorité de la consommation électrique des disques durs est utilisée pour faire tourner les plateaux, le reste étant négligeable. Rien d'étonnant à cela, faire tourner de lourds plateaux en métal consomme naturellement plus que l'électronique embarquée du disque. Naturellement, plus la vitesse de rotation est importante, plus la consommation l'est. Même chose pour le nombre de plateaux : faire tourner 5 plateaux demandera plus d'efforts que d'en faire tourner un seul. Et enfin, plus les plateaux sont massifs, plus il faudra consommer d'énergie pour les faire tourner à une vitesse déterminée. L'épaisseur des plateaux étant plus ou moins la même sur tous les disques durs, ce n'est pas forcément le cas du diamètre des plateaux. Ce diamètre est un bon indice de la masse des plateaux, et donc de la consommation. Pour résumer, la consommation électrique d'un disque dur est approximativement égal à : $\text{Diamètre} * \text{Nombre de plateaux} * \text{Vitesse de rotation}$. La conséquence est que la capacité influence la consommation, un grand nombre de plateaux étant souvent synonyme de disque de forte capacité, de même que des plateaux larges auront une plus grande capacité que des plateaux petits. On peut donc résumer en disant que la consommation électrique du disque est approximativement égale au produit Capacité * Vitesse de rotation.

Limiter la consommation énergétique est donc difficile, et ne peut passer que par des moyens indirects. La majorité des contrôleurs de disque utilise des algorithmes pour mettre en pause les têtes de lecture ou cesser de faire tourner les plateaux en l'absence d'accès au disque. La file d'attente des requêtes permet d'implémenter ces algorithmes. Ces algorithmes vont ralentir ou accélérer la vitesse de rotation des plateaux suivant la charge de travail du disque dur. Si le disque dur a beaucoup de travail, la vitesse des plateaux augmentera jusqu'à une valeur maximale. Dans le cas contraire, la vitesse des plateaux diminuera jusqu'à une valeur minimale. Pour estimer la charge de travail, le disque dur se base sur le remplissage de la mémoire tampon : plus la file d'attente est remplie, plus les plateaux tourneront vite.

Les solid-state drives

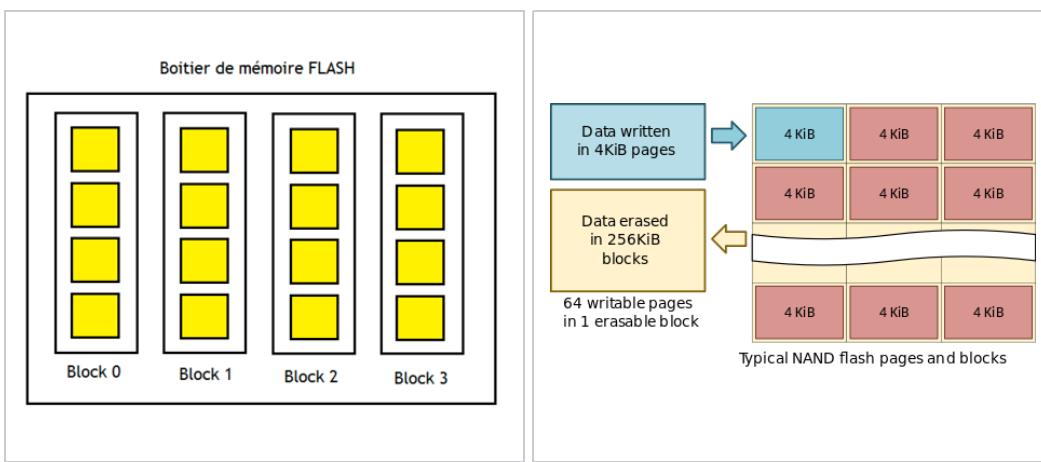
De nos jours, les disques durs tendent à être remplacés par des mémoires de masse dont le support de mémorisation est électronique : les **solid-state drives**, ou SSD. Certains SSD assez anciens utilisaient de la mémoire DDR-SDRAM, comme on en trouve dans nos PC, mais les SSD actuels sont fabriqués avec de la mémoire Flash. Il s'agit d'une forme évoluée d'EEPROM, déclinée en plusieurs versions : NOR et NAND. Les mémoires Flash sont accessibles via un bus série (de 1 bit), sauf pour quelques exceptions. A part le support de mémorisation et les circuits de conversion série-parallèle, l'organisation interne d'un SSD est similaire à celle d'un mémoire adressable.



Support de mémorisation

L'interface d'une Flash autorise plusieurs opérations : la lecture, l'écriture (ou reprogrammation, vu qu'il s'agit d'une mémoire EEPROM) et l'effacement. Si la lecture d'un byte individuel est possible, l'effacement s'effectue par paquets de plusieurs bytes, nommés **blocs**.

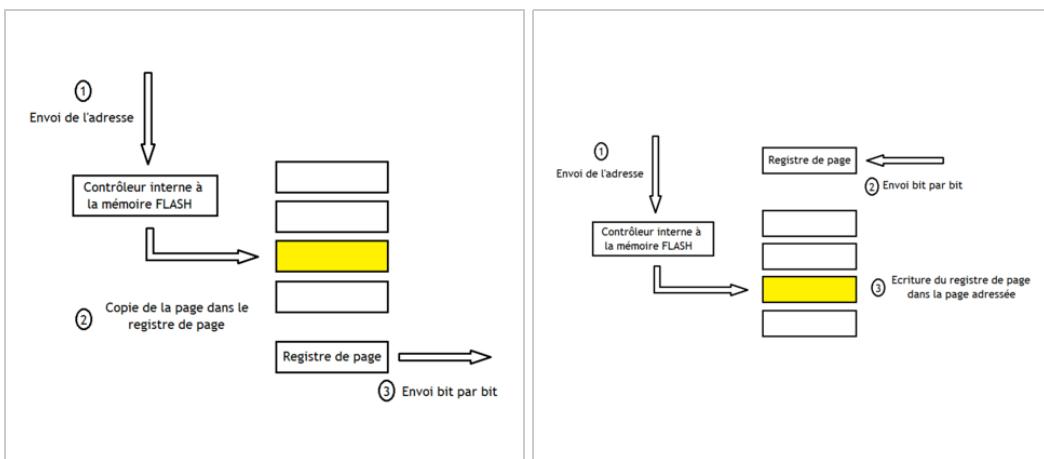
Dans les mémoires NOR, il est possible de lire ou d'écrire octet par octet, chaque octet ayant une adresse. Mais sur les NAND, on ne peut pas : on est obligé de lire ou d'écrire par paquets d'octets relativement importants. Même si vous ne voulez modifier qu'un seul bit ou un seul secteur, vous allez devoir modifier un paquet de 512 à 4096 octets. Ces paquets sont appelés des **pages**. Si une page est vide, on peut écrire dans celle-ci sans problème, même si les autres pages du bloc sont occupées. Mais si la page contient déjà des données, on est obligé d'effacer tout le bloc. Ce problème est géré différemment suivant le SSD, comme on le verra dans la suite du cours.



Pages - microarchitecture d'un SSD.

Pages et blocs.

Vu que les pages font plusieurs kibioctets, il est évident que les Flash NAND ne peuvent pas utiliser une interface parallèle : vous imaginez des bus de 4096 bits ? À la place, elles sont obligées d'utiliser des bus série pour envoyer ou récupérer les pages bit par bit. Pour faire la conversion série-parallèle, chaque plane ou bloc contient un registre à décalage série-parallèle : le **registre de page**.



Lecture sur une Flash.

Écriture sur une Flash.

Le support de mémorisation peut être optimisé de manière à gagner en performances, les optimisations étant souvent les mêmes que celles utilisées sur les mémoires RAM : entrelacement, pipeline, etc. Pour ce qui est du pipeline, les Flash se limitent à recevoir une requête de lecture/écriture pendant que la Flash effectue la précédente. Pour cela, le contrôleur de la Flash doit être modifié et chaque plane/bloc se voit attribuer un **registre de cache**, pour mettre en attente une écriture pendant que la précédente utilise le registre de page. La donnée en cours d'écriture est copiée du registre de page sur le support de mémorisation, tandis que le registre de cache accumule les bits de la prochaine donnée à écrire. Pour une lecture, c'est la même chose : le contenu du registre de cache peut être envoyé sur la sortie série, pendant qu'une autre donnée est copiée du support de mémorisation vers le registre de page. Évidemment, le contenu des registres de page et de cache est échangé à chaque début de lecture/écriture. Avec quelques optimisations, on peut se passer de recopies en échangeant le rôle des registres : le registre de page devient le registre de cache, et vice versa.

Certaines Flash permettent d'effectuer des **copies rapides entre pages**, opérations très courantes sur les SSD, sans passer par le registre de page. Mais cela a un cout : il faut que les pages en question soient reliées avec un lien série pour transférer les données de page en page. Et plus une mémoire Flash contient de pages, plus le nombre d'interconnexions augmente. Pour limiter le nombre d'interconnexions, ces copies rapides entre page ne sont possibles qu'à l'intérieur d'une plane ou d'un die : impossible de copier rapidement des données mémorisées dans des dies ou planes différents.

Flash transaction layer

Le système d'exploitation transmet des adresses LBA au SSD, qui sont traduites en adresses de mémoire FLASH par le **contrôleur de SSD**. La majorité des SSD (si ce n'est tous) partent du principe qu'un secteur est égal à une page. Dans le cas le plus simple, la correspondance entre adresse LBA et adresse de mémoire Flash est fixée une fois pour toutes à la construction du SSD : on parle alors de **correspondance statique**. Cette correspondance statique effectue les écritures en place, à savoir que toute réécriture d'une page demande d'effacer le bloc complet. On doit donc sauvegarder temporairement le bloc, mettre à jour la page à écrire dans le bloc, effacer le bloc et réécrire le résultat. Pour cela, le registre de page est agrandi pour pouvoir stocker un bloc complet. Or, les cellules de mémoire Flash ne peuvent supporter qu'un nombre limité d'effacements. Or, avec une correspondance statique, certaines cellules seront nettement plus souvent réécrites que d'autres : après tout, certains fichiers sont très souvent accédés, tandis que d'autres sont laissés à l'abandon. Certaines cellules vont donc tomber en panne rapidement, ce qui réduit la durée de vie du SSD.

Pour régler ces problèmes, certains contrôleurs utilisent une **correspondance dynamique** : ils font ce que l'on appelle du relocate on write. Avec cette correspondance dynamique, la page à écrire est écrite dans un bloc vide, et l'ancienne page est marquée invalide. L'effacement des blocs invalides est effectué plus tard, quand le SSD est inutilisé. Pour cela, le SSD possède une liste des pages vierges et des blocs en attente d'effacement. La position d'une adresse LBA dans la mémoire Flash change donc à chaque écriture. Le SSD doit donc se souvenir à quelle page ou bloc correspond chaque adresse LBA, grâce à une table de correspondance mémorisée soit dans une mémoire non volatile, soit dans les premiers secteurs du SSD. Reste que cette correspondance d'adresse peut se faire de trois manières :

- soit le contrôleur travaille au niveau du bloc ;
- soit il travaille au niveau de la page ;
- soit il utilise une technique hybride.

Certains contrôleurs fonctionnent par page : un secteur peut se voir attribuer n'importe quelle page, peu importe le bloc où se situe la page. Avec d'autres contrôleurs, des secteurs consécutifs sont placés dans le même bloc, dans des pages différentes. La position de la page dans un bloc est déterminée statiquement, mais le bloc dans lequel placer la donnée change. Vu qu'il y a plusieurs pages par bloc, la table prend donc nettement moins de place que son équivalent par page. Il existe des techniques hybrides, qui vont utiliser une table de correspondances par bloc pour les lectures, avec un mise à jour par page lors de l'écriture.

Wear leveling

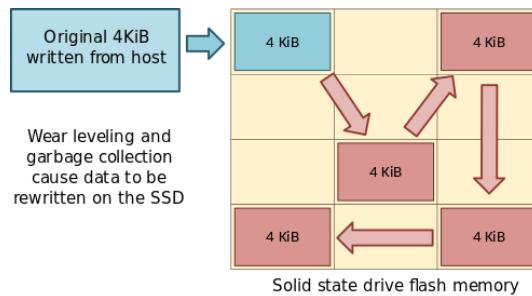
Certains contrôleurs de SSD utilisent une technique pour augmenter la durée de vie du SSD : le wear leveling, ou égalisation d'usure. L'idée est simple : on va chercher à répartir les écritures de manière à ce que tous les blocs soient « usés » de la même façon : on va faire en sorte que le nombre d'écritures soit identique dans tous les blocs. Pour cela, il suffit de déplacer les données fortement utilisées dans des blocs pas vraiment usés. Vu que les données fréquemment utilisées sont présentes sur des blocs très usés, qui ont un grand nombre d'écriture, on peut considérer que le wear leveling consiste à déplacer les données des blocs usés dans les blocs relativement intacts.

Le plus simple est le **wear leveling dynamique**, où les blocs sont déplacés lorsqu'ils sont réécrits. Quand on veut réécrire une donnée, on va choisir un bloc qui a relativement peu d'écritures à son actif. Pour cela, le contrôleur de SSD doit mémoriser le nombre d'écritures de chaque bloc dans une mémoire non-volatile. Avec ce wear leveling dynamique, les données qui ne sont pas réécrites souvent ne bougent quasiment jamais du bloc qui leur est assigné, ce qui est loin d'être optimal. Il vaut mieux déplacer des telles données sur des blocs très usés, et allouer leur emplacement initial à une donnée fréquemment mise à jour. C'est ce principe qui se cache derrière le **wear leveling statique**. Il va de soi que le contrôleur doit alors mémoriser la liste des blocs peu utilisés, en plus de la liste des blocs vides.

L'amplification d'écriture

Avec correspondance dynamique, chaque écriture gaspille un bloc ou une page, qui contient l'ancienne version de la donnée écrite. Si un secteur est écrit six fois de suite, six blocs ou six pages seront utilisés pour mémoriser ce secteur dont un seul contiendra une donnée valide : ce qui fait un gâchis de cinq blocs/pages. De plus, le SSD peut aussi profiter des périodes d'inactivité du SSD pour réorganiser les données sur le disque dur, de

manière à obtenir des blocs totalement vides qu'il pourra effacer. Ce ramasse-miettes a toutefois un défaut : il est obligé de migrer des pages dans la mémoire Flash, ce qui est à l'origine d'écritures supplémentaires. De plus, le wear leveling ajoute des écritures supplémentaires. Ce phénomène s'appelle l'**amplification d'écriture**.



Pour limiter la casse, certains SSD contiennent des Flash en rab, invisibles du point de vue du système d'exploitation : on parle de **sur-approvisionnement**. Ces Flash en rab peuvent aussi servir si jamais une mémoire Flash tombe en panne après avoir été trop souvent écrite. Le remplacement s'effectue simplement en mettant à jour la table de correspondances.

L'amplification d'écriture est à l'origine d'un autre problème, qui concerne les situations où l'on veut effacer des données sans que ce soit nécessaire de les récupérer. Avec des technologies de pointes, il est parfois possible de récupérer physiquement des données présentes sur un disque dur, même si celui-ci est partiellement détruit ou que les données en question ont été écrasées. Pour éviter cela, il vous suffit de réécrire des données aléatoires ou des zéros sur les secteurs concernés plusieurs centaines ou milliers de fois de suite. Mais si vous essayez de faire cela avec des SSD, cela n'effacera pas les données : cela se contentera de prendre des blocs vides pour écrire les données aléatoires dedans. Pour régler ce problème, certains SSD modernes possèdent des commandes qui permettent de remettre à zéro l'intégralité du disque dur, ou d'effacer des pages ou blocs bien précis.

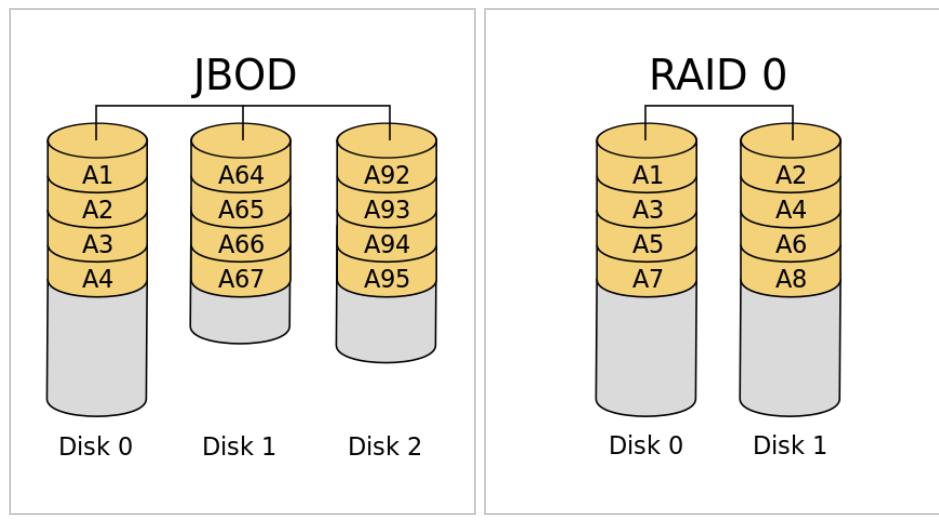
Les technologies RAID

Les technologies RAID (Redundant Array of Inexpensive Disks) sont des technologies qui permettent d'utiliser plusieurs disques durs de manière à gagner en performances, en espace disque ou en fiabilité. Ces technologies sont apparues dans les années 70, sur des ordinateurs devant supporter des pannes de disque dur et sont toujours utilisées dans ce cas de figure à l'heure actuelle, notamment sur les serveurs dits à haute disponibilité. Ces techniques peuvent être prises en charge aussi bien par le logiciel que par le matériel. Dans le premier cas, c'est le système d'exploitation (et plus précisément le système de fichiers) qui se charge de la gestion des disques durs et de la répartition des données sur ceux-ci. Dans l'autre cas, le RAID est pris en charge par un contrôleur spécialisé, intégré à la carte mère ou présent sous la forme d'une carte d'extension. La dernière méthode est évidemment plus rapide, les calculs étant déportés sur une carte spécialisée au lieu d'être pris en charge par le processeur. Il existe plusieurs types de RAID, qui ont des avantages et leurs inconvénients bien précis. Dans les grandes lignes, on peut classer les différentes techniques de RAID dans deux catégories : les **RAID standards** et les **RAID combinés**.

RAID standards

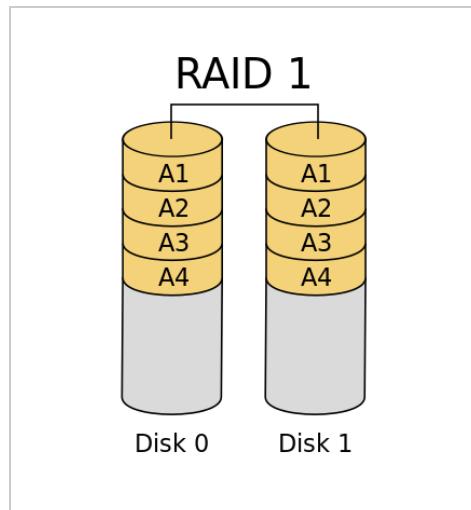
Les techniques de RAID dites standards ont toutes été inventées en premier. Il s'agit des techniques de JBOD, RAID 0, RAID 1, RAID2, RAID3, RAID 4, RAID5 et RAID6. Les trois premières sont les plus simples, dans le sens où elles ne font pas appel à des bits de parité.

- La **technologie JBOD** permet de regrouper plusieurs disques durs en une seule partition. JBOD est l'acronyme de l'expression *Just a Bunch Of Disks*. Avec cette technique, on attend qu'un disque dur soit rempli pour commencer à entamer le suivant. Les capacités totales des disques durs s'additionnent. Il est parfaitement possible d'utiliser des disques durs de taille différentes.
- Avec le **RAID 0**, des données consécutives sont réparties sur des disques durs différents. Là encore, le système RAID contient plusieurs disques durs, mais est reconnu comme une seule et unique partition par le système d'exploitation. La capacité totale du RAID 0 est égale à la somme des capacités de chaque disque dur, comme avec le JBOD. La différence avec le JBOD tient dans le fait que les données d'un même fichier étant systématiquement réparties sur plusieurs HDDs, ce que ne faisait pas le JBOD. Cela permet une amélioration des performances lors de l'accès à des données consécutives, celles-ci étant lues/écrites depuis plusieurs disques durs en parallèle.
- Avec le **RAID 1**, les données sont copiées à l'identique sur tous les disques durs. Chaque disque dur est ainsi une copie de tous les autres, chaque disque dur ayant exactement le même contenu. L'avantage de cette technique réside dans la résistance aux pannes : cela permet de résister à une panne qui touche un grand nombre de disque dur, tant qu'au moins un disque dur est épargné. Par contre, cette technique ne permet pas de gagner en espace disque : l'ensemble des disques durs est vu comme un unique disque de même capacité qu'un disque individuel. Des gains en performances sont possibles, vu que des données consécutives peuvent être lues depuis plusieurs disques durs. Mais cette optimisation entraîne une baisse des performances en écriture, ce qui fait que peu de contrôleurs RAID l'utilisent.



JBOD

RAID 0

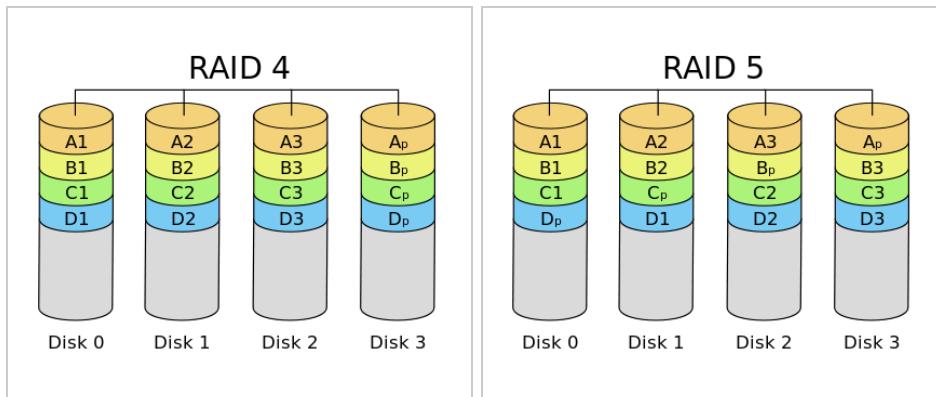


RAID 1

Les RAID 2, 3, 4, 5 et 6 font appel à des bits de parité. Cela leur permet d'obtenir une résilience aux pannes plus attractive que le RAID1. Ces techniques ont tendance à utiliser un seul HDD pour stocker des données redondantes, soit nettement moins que le RAID1. En contrepartie, la

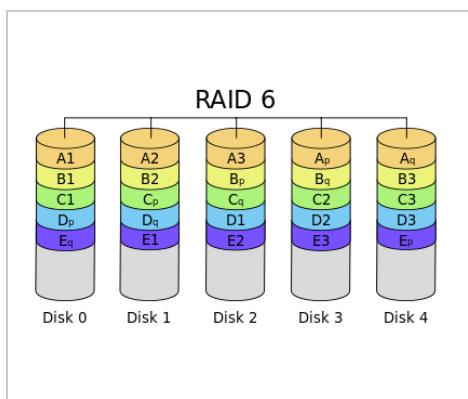
récupération suite à une panne est plus lente, vu qu'il faut reconstituer les données originelles via un calcul qui fait intervenir les données de tous les disques durs, ainsi que les données de parité.

- Les **RAID 2, 3 et 4** peuvent être vu comme une sorte de RAID 0 amélioré. Il est amélioré dans le sens où on ajoute un disque pour les données de parité à un RAID 0. L'idée est simplement de calculer, pour tous les secteurs ayant la même adresse, un secteur de parité. Les octets des différents disques du RAID 0 seront utilisés pour calculer un octet de parité, qui sera enregistré sur un disque de parité à part. Il faut noter que certaines formes améliorées du RAID 4 dupliquent les disques de parité, ce qui permet de résister à plus d'une panne de disque dur (autant qu'il y a de disques de parité). Mais ces améliorations ne font pas partie des niveaux de RAID standard.
- Le **RAID 5** est similaire au RAID 4, sauf que les secteurs de parité sont répartis sur les différents disques dur, afin de gagner en performances.
- Le **RAID 6** est une amélioration du RAID 5 où les données de parité sont elles-mêmes dupliquées en plusieurs exemplaires. Cela permet de résister à la défaillance de plus d'un HDD.



RAID 4

RAID 5

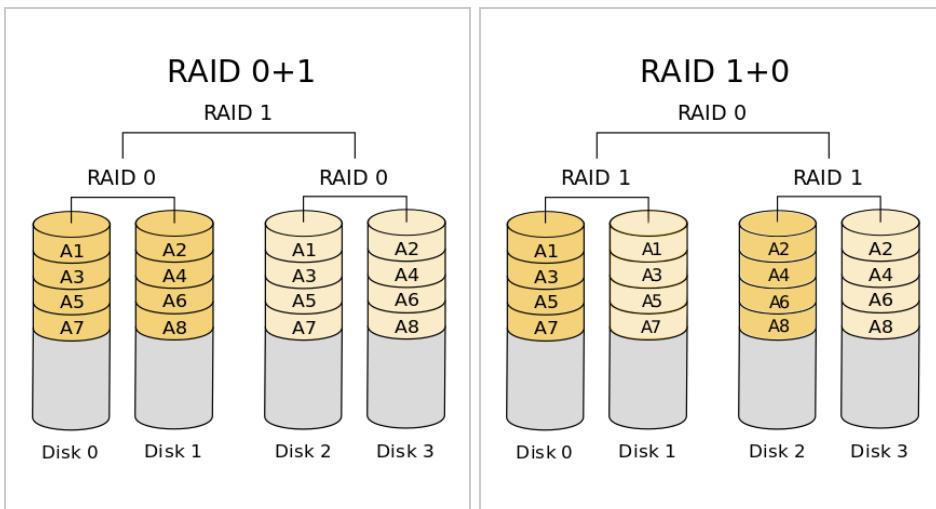


RAID 6

Type de RAID	JBOD	RAID 0	RAID 1	RAID 2, 3 et 4	RAID 5	RAID 6
Performances	Pas d'amélioration	Amélioration en lecture et écriture	Amélioration en lecture, sous conditions	Amélioration en lecture et écriture, inférieure à celle obtenue avec le RAID 0, mais supérieure à celle obtenue avec le RAID 4	Amélioration en lecture et écriture, inférieure à celle obtenue avec le RAID 0, mais supérieure à celle obtenue avec le RAID 4	Amélioration en lecture et écriture, inférieure à celle obtenue avec le RAID 0, mais supérieure à celle obtenue avec le RAID 4
Espace disque	Somme des capacités des HDD	Somme des capacités des HDD	Pas d'amélioration	Somme des capacités de tous les HDD, sauf un	Somme des capacités de tous les HDD, sauf un	Somme des capacités de tous les HDD, sauf un
Résilience aux pannes	Pas d'amélioration	Pas d'amélioration	Excellent : survit à autant de pannes qu'il y a de HDD, sauf un	Minimale : survit à la panne d'un seul HDD	Minimale : survit à la panne d'un seul HDD	Intermédiaire : survit à la panne de plusieurs HDD (autant qu'il y a de duplication des données de parité)

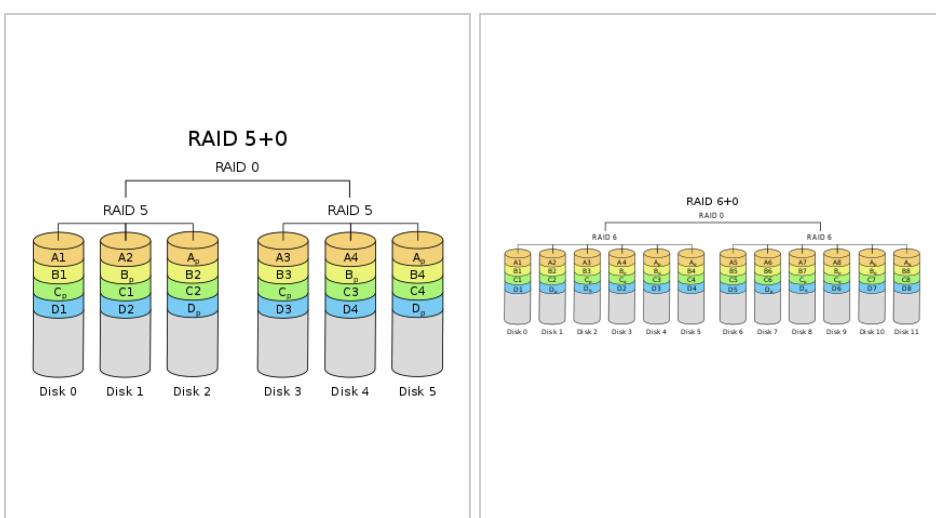
RAID combinés

Il est possible de combiner des disques durs en utilisant diverses techniques de RAID. Par exemple, on peut utiliser un RAID1 de disques en RAID 0 : on parle alors de **RAID 01**. Il est aussi possible d'utiliser un RAID 0 de disques en RAID 1 : on parle alors de **RAID 10**. De même, utiliser un RAID 0 de RAID 3, 4, 5 ou 6 est possible : on parle respectivement de RAID 30, 40, 50, 60.



RAID 01

RAID 10



RAID 50

RAID 60

Langage machine et assembleur

Ce chapitre va aborder le langage machine d'un processeur, à savoir un standard qui définit les instructions du processeur, le nombre de registres, etc. Dans ce chapitre, on considérera que le processeur est une boîte noire au fonctionnement interne inconnu. Nous verrons le fonctionnement interne d'un processeur dans quelques chapitres. La majorité des concepts qui seront vus dans ce chapitre ne sont rien d'autre que les bases nécessaires pour apprendre l'assembleur.

Instructions machine

Pour simplifier, on peut classer les instructions en trois grands types :

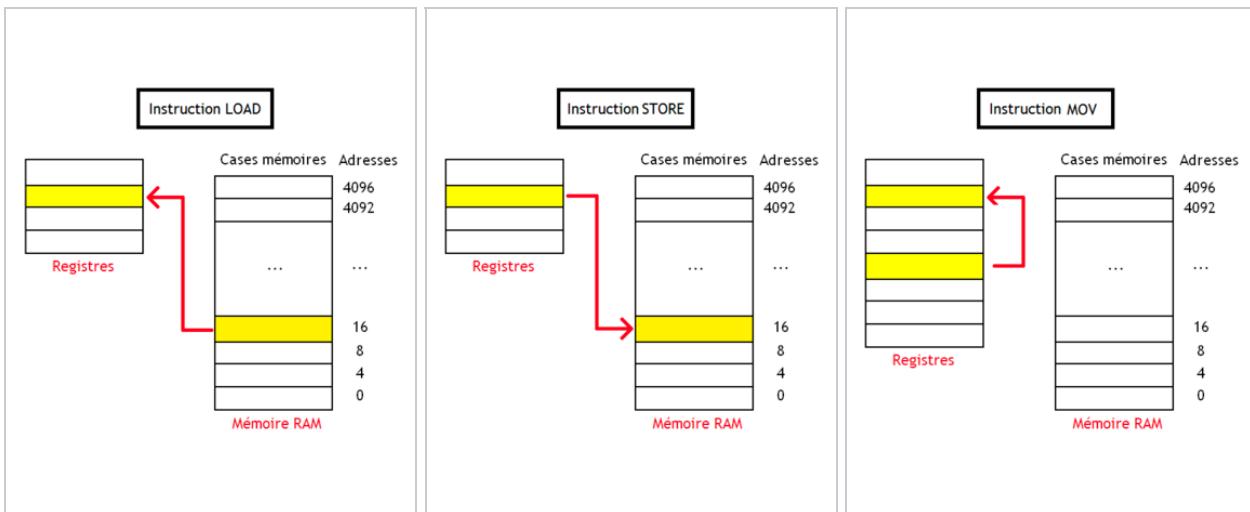
- les échanges de données entre mémoires ;
- les calculs et autres opérations arithmétiques ;
- les instructions de comparaison ;
- les instructions de branchement.

À côté de ceux-ci, on peut trouver d'autres types d'instructions plus exotiques, pour gérer du texte, pour modifier la consommation en électricité de l'ordinateur, pour chiffrer ou déchiffrer des données de taille fixe, générer des nombres aléatoires, etc.

Accès mémoire

Les instructions d'accès mémoire permettent de copier ou d'échanger des données entre le processeur et la mémoire. On peut ainsi copier le contenu d'un registre en mémoire, charger une donnée de la RAM dans un registre, initialiser un registre à une valeur bien précise, etc. Il en existe plusieurs, les plus connues étant les instructions : LOAD, STORE et MOV.

- LOAD est une instruction de lecture : elle copie le contenu d'un ou plusieurs mots mémoire consécutifs dans un registre. Le contenu du registre est remplacé par le contenu des mots mémoire de la mémoire RAM. Par exemple...
- STORE fait l'inverse : elle copie le contenu d'un registre dans un ou plusieurs mots mémoire consécutifs en mémoire RAM.
- MOV copie le contenu d'un registre dans un autre, sans passer par la mémoire (le contenu du registre de destination est perdu).



Instruction LOAD.

Instruction STORE.

Instruction MOV.

Instructions de calcul

Les **instructions arithmétiques** sont les plus courantes et comprennent au minimum l'addition, la soustraction, la multiplication, éventuellement la division, parfois les opérations plus complexes comme la racine carrée. Ces instructions dépendent de la représentation utilisée pour coder ces nombres : on ne manipule pas de la même façon des nombres signés, des nombres codés en complément à 1, des flottants simple précision, des flottants double précision, etc. Le processeur dispose souvent d'une instruction par type à manipuler : on peut avoir une instruction de multiplication pour les flottants, une autre pour les entiers codés en complément à deux, etc. Sur d'autres machines assez anciennes, on stockait le type de la donnée (est-ce un flottant, un entier codé en BCD, etc.) dans la mémoire. Chaque nombre manipulé par le processeur incorporait un tag, une petite suite de bits qui permettait de préciser son type. Le processeur ne possédait pas d'instructions en plusieurs exemplaires pour faire la même chose, et utilisait le tag pour déduire comment faire ses calculs. Des processeurs de ce type s'appellent des architectures à tags, ou **tagged architectures**.

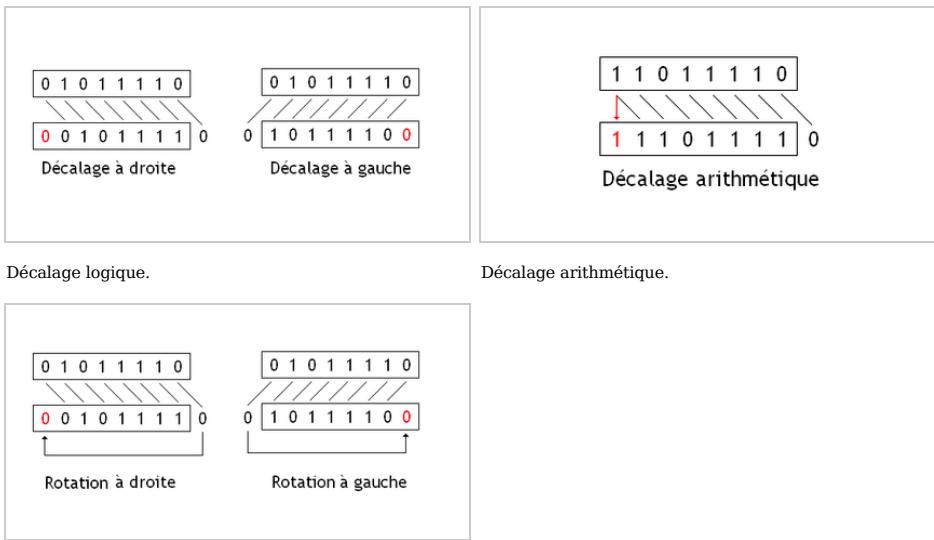
Les processeurs peuvent parfaitement gérer les calculs sur des **nombre flottants**. Il faut néanmoins préciser que le support de la norme IEEE 754 par la FPU/le jeu d'instruction n'est pas une obligation : certains processeurs s'en moquent royalement. Dans certaines applications, les programmeurs ont souvent besoin d'avoir des calculs qui s'effectuent rapidement et se contentent très bien d'un résultat approché. Dans ces situations, on peut utiliser des formats de flottants différents de la norme IEEE 754 et les circuits de la FPU sont simplifiés pour être plus rapides. Par exemple, certains circuits ne gèrent pas les underflow, overflow, les NaN ou les infinis, voire utilisent des formats de flottants exotiques. À côté, certaines architectures ont besoin que le compilateur ou les programmes fassent quelques manipulations pour que les calculs effectués avec des flottants respectent la norme IEEE 754. Tout dépend des instructions machines que le compilateur utilise. Par exemple, certains processeurs implémentent non seulement les instructions de la norme, mais aussi d'autres instructions sur les flottants qui ne sont pas supportées par la norme IEEE 754. Par exemple, certaines fonctions mathématiques telles que sinus, cosinus, tangente, arctangente et d'autres, sont supportées par certaines FPU. Le seul problème, c'est que ces instructions peuvent mener à des erreurs de calcul. Or, certaines de ces instructions sont parfois utilisées par certains compilateurs, ce qui peut produire des erreurs de calcul incompatibles avec la norme IEEE 754.

Les tous premiers ordinateurs pouvaient manipuler des données de taille arbitraire. Alors certes, ces processeurs n'utilisaient pas vraiment les encodages de nombres qu'on a vus au premier chapitre. À la place, ils stockaient leurs nombres dans des chaînes de caractères ou des tableaux encodés en BCD. De nos jours, les ordinateurs utilisent des entiers de taille fixe. La taille des données à manipuler peut dépendre de l'instruction. Ainsi, un processeur peut avoir des instructions pour traiter des nombres entiers de 8 bits, et d'autres instructions pour traiter des nombres entiers de 32 bits, par exemple. On peut aussi citer le cas des flottants : il faut bien faire la différence entre flottants simple précision et double précision !

A coté des instructions de calcul, on trouve des **instructions logiques** qui travaillent sur des bits ou des groupes de bits. Nous allons détailler ces instructions avant de passer à la suite. La négation, ou encore le NON bit à bit, inverse tous les bits d'un nombre : les 1 deviennent des 0 et les 0

deviennent des 1. Vient ensuite le ET bit à bit, qui agit sur deux nombres : il va prendre les bits qui sont à la même place et va effectuer un ET (l'opération effectuée par la porte logique ET). Exemple : $1100 \cdot 1010 = 1000$. Il existe des instructions similaires avec le OU ou le XOR.

Les **instructions de décalage** décalent tous les bits d'un nombre vers la gauche ou la droite. Il existe plusieurs types de décalages, dont deux vont nous intéresser particulièrement : les décalages logiques, et les décalages arithmétiques. Le **décalage logique**, ou logical shift, décale tout ses chiffres d'un ou plusieurs crans vers la gauche ou la droite, et remplit les vides par des zéros. Grâce à ce remplissage par des zéros, un décalage de n rangs vers la droite/gauche correspond à une multiplication/division entière par 2^n , pour les entiers non signés. Mais lorsqu'on effectue un décalage à droite, certains bits vont sortir du résultat et être perdus : le résultat est tronqué ou arrondi vers zéro. Pour pouvoir effectuer des divisions par 2^n sur des nombres négatifs avec un décalage, on a inventé les **décalages arithmétiques** ou arithmetical shift. Ils sont similaires aux décalages logiques, si ce n'est que les vides laissés par le décalage sont remplis avec le bit de signe, et non pas des zéros. Ces instructions sont équivalentes à une multiplication/division par 2^n , que le nombre soit signé ou non. La différence avec un décalage logique est que les nombres négatifs sont arrondis vers moins l'infini. Pour donner un exemple, 92 sera arrondi en 4, tandis que -92 sera arrondi en -5. Les **instructions de rotation** sont similaires aux décalages, à part que les bits qui sortent du nombre d'un côté rentrent de l'autre et servent à boucher les trous.



Rotation de bits.

Instructions de test

Les **instructions de test** comparent deux valeurs (des adresses, ou des nombres entiers ou à virgule flottante). Sur la majorité des processeurs, ces instructions ne font qu'une comparaison à la fois. D'autres processeurs ont des instructions de test qui effectuent plusieurs comparaisons en même temps et fournissent plusieurs résultats. Par exemple, un processeur x86 possède une instruction CMP qui vérifie simultanément si deux valeurs A et B sont égales, différentes, si A est inférieur à B, si A est supérieur à B, etc. Les instructions de comparaison permettent souvent d'effectuer les comparaisons suivantes :

- A > B (est-ce que A est supérieur à B ?) ;
- A < B (est-ce que A est inférieur à B ?) ;
- A == B (est-ce que A est égal à B ?) ;
- A != B (est-ce que A est différent de B ?).

Le résultat d'une comparaison est un bit, qui dit si la condition testée est vraie ou fausse. Dans la majorité des cas, ce bit vaut 1 si la comparaison est vérifiée, et 0 sinon. Une fois que l'instruction a fait son travail, il reste à stocker son résultat quelque part. Et pour ce faire, il existe plusieurs techniques :

- soit on utilise des registres d'état ;
- soit on utilise des registres à prédictats ;
- soit on ne mémorise pas le résultat.

Certains processeurs incorporent un **registre d'état**, qui stocke des bits qui ont chacun une signification prédéterminée lors de la conception du processeur. Le ou les bits du registre d'état modifiés par une instruction de test dépendent de l'instruction utilisée : par exemple, on peut utiliser un bit qui indiquera si l'entier testé est égal à un autre, un autre bit qui indiquera si le premier entier testé est supérieur à l'autre, etc. Ce registre peut aussi contenir d'autres bits suivant le processeur, comme le bit de débordement qui prévient quand le résultat d'une instruction est trop grand pour tenir dans un registre, le bit null qui précise que le résultat d'une instruction est nul (vaut zéro), le bit de retenue qui est utile pour les additions, le bit de signe qui permet de dire si le résultat d'une instruction est un nombre négatif ou positif. Le registre d'état a un avantage : certaines instructions arithmétiques peuvent modifier les bits du registre d'état, ce qui leur permet parfois de remplacer une instruction de test. Par exemple, prenons le cas d'un processeur où la soustraction modifie le bit null du registre d'état : on peut tester si deux nombres sont égaux en soustrayant leur contenu, le bit null étant mis à zéro si c'est le cas.

D'autres processeurs utilisent des **registres à prédictats**, des registres de 1 bit qui peuvent stocker n'importe quel résultat de comparaison. Une comparaison peut enregistrer son résultat dans n'importe quel registre à prédictats : elle a juste à préciser lequel avec son nom de registre. Cette méthode est plus souple que l'utilisation d'un registre d'état dont les bits ont une utilité fixée une fois pour toutes. Les registres à prédictats sont utiles pour accumuler les résultats de plusieurs comparaisons et les utiliser par la suite. Par exemple, une instruction de branchement pourra vérifier la valeur de plusieurs de ces registres pour prendre une décision. C'est impossible avec les autres approches, dont les branchements ne peuvent utiliser qu'un seul résultat de comparaison pour prendre leur décision.

Enfin, sur d'autres processeurs, il n'y a pas de registres pour stocker les résultats de comparaisons : les comparaisons sont fusionnées avec les branchements en une seule instruction, rendant le stockage du résultat inutile.

Branchements et comparaisons

Pour exécuter une suite d'instructions dans le bon ordre, le processeur doit connaître l'adresse de la prochaine instruction à exécuter : c'est le rôle du registre d'adresse d'instruction, aussi appelé **program counter**. Ce registre est régulièrement mis à jour de manière à passer d'une instruction à la suivante quand c'est nécessaire. Pour rendre un ordinateur plus intelligent, on peut souhaiter qu'il n'exécute une suite d'instructions que si une certaine condition est remplie, ou lui permettre de répéter une suite d'instructions. Pour cela, les instructions de test sont combinées avec des

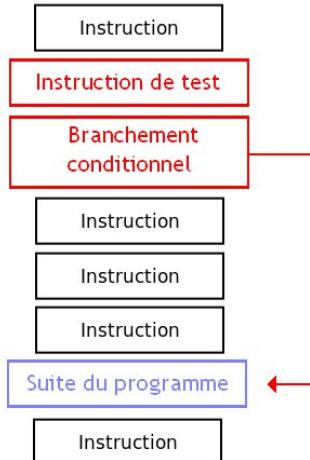
instructions de branchements. Ces branchements modifient la valeur stockée dans le registre d'adresse d'instruction, ce qui permet de sauter directement à une instruction autre que l'instruction suivante et poursuivre l'exécution à partir de celle-ci. Il existe deux types de branchements :

- les **branchements inconditionnels**, avec lesquels le processeur passe toujours à l'instruction vers laquelle le branchements va renvoyer ;
- les **branchements conditionnels**, où le branchements n'est exécuté que si certains bits du registre d'état sont à une certaine valeur.

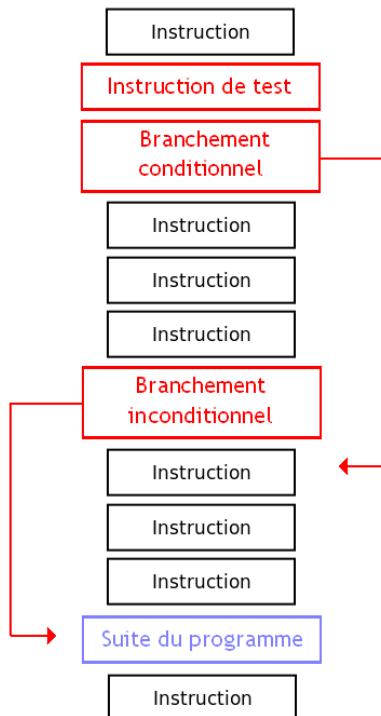
Sur la plupart des processeurs, les branchements conditionnels sont précédés d'une instruction de test ou de comparaison. Mais d'autres effectuent le test et le branchements en une seule instruction machine, pour se passer de registre d'état ou de registres à prédictifs. Sur d'autres processeurs, les instructions de test permettent de zapper l'instruction suivante si la condition testée est fausse : cela permet de simuler un branchements conditionnel à partir d'un branchements inconditionnel. Sur quelques rares processeurs, le *program counter* est un registre général qui peut être modifié par n'importe quel opération arithmétique : cela permet de remplacer les branchements par une simple écriture dans le *program counter*.

Généralement, les branchements servent à implémenter des fonctionnalités de langages programmation impératifs, appelées **structures de contrôles**. Celles-ci sont au nombre de plusieurs. Le IF permet d'exécuter une suite d'instructions si et seulement si une certaine condition est remplie. Le IF...ELSE sert à effectuer une suite d'instructions différente selon que la condition est respectée ou non : c'est un SI...ALORS contenant un second cas. Une boucle consiste à répéter une suite d'instructions machine tant qu'une condition est valide (ou fausse).

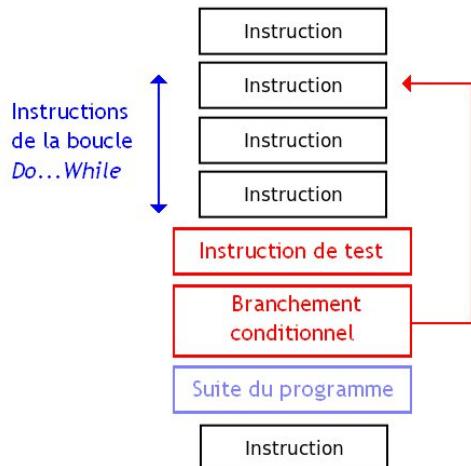
Implémenter un IF avec des branchements est assez simple : il suffit d'une comparaison pour tester la condition et d'un branchements. Si la condition testée est vraie, on poursuit l'exécution juste après le branchements. Sinon, on reprend directement à la suite du programme.



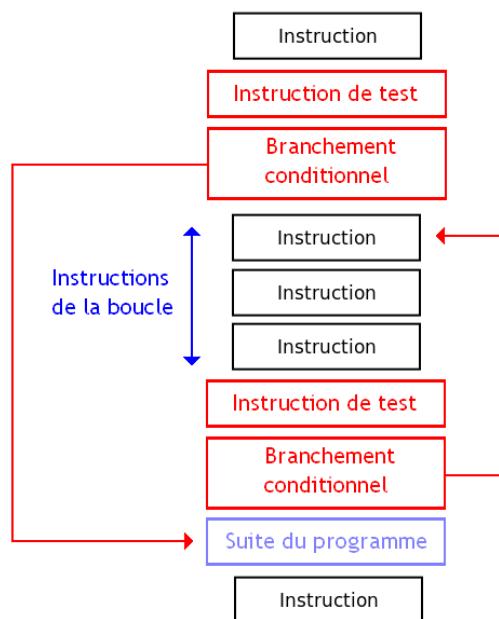
Le IF...ELSE sert à effectuer une suite d'instructions différente selon que la condition est respectée ou non : c'est un IF contenant un second cas.



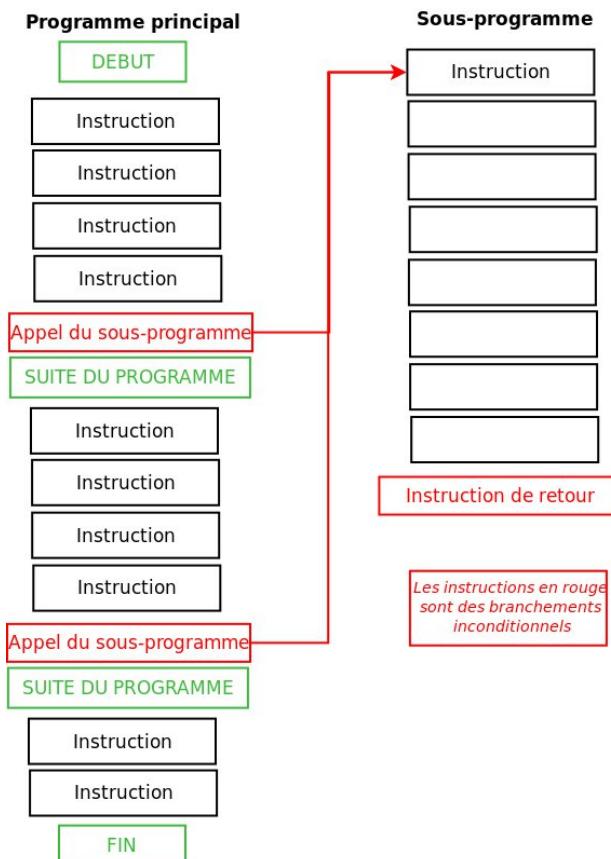
Les boucles sont une variante du IF dont le branchements renvoie le processeur sur une instruction précédente. Commençons par la boucle DO...WHILE : la suite d'instructions est exécutée au moins une fois, et est répétée tant qu'une certaine condition est vérifiée. Pour cela, la suite d'instructions à exécuter est placée avant les instructions de test et de branchements, le branchements permettant de répéter la suite d'instructions si la condition est remplie. Si jamais la condition testée est fausse, on passe tout simplement à la suite du programme.



Une boucle WHILE...DO est identique à une boucle DO...WHILE à un détail près : la suite d'instructions de la boucle n'a pas forcément besoin d'être exécutée au moins une fois. On peut donc adapter une boucle DO...WHILE pour en faire une boucle WHILE...DO : il suffit de tester si la boucle doit être exécutée au moins une fois avec un IF, et exécuter une boucle DO...WHILE équivalente si c'est le cas.



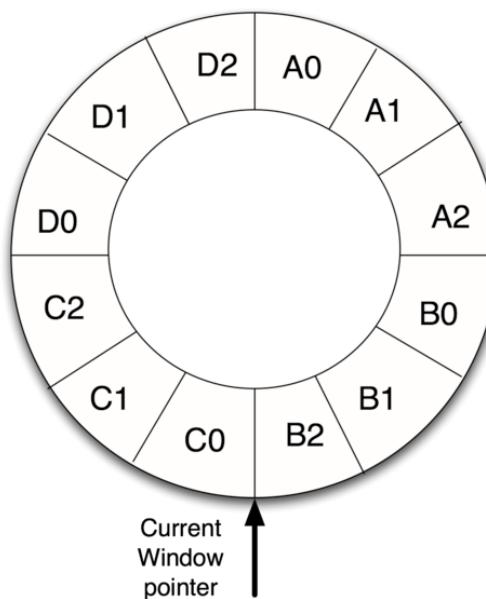
Et enfin, ces branchements sont utilisés pour fabriquer des fonctions. Pour comprendre ce que sont ces fonctions, il faut faire quelques rappels. Un programme contient souvent des suites d'instructions présentes en plusieurs exemplaires, qui servent souvent à effectuer une tâche bien précise : calculer un résultat bien précis, communiquer avec un périphérique, écrire un fichier sur le disque dur, ou autre chose encore. Il est possible de ne conserver qu'un seul exemplaire en mémoire, et l'utiliser au besoin. L'exemplaire en question est ce qu'on appelle une **fonction**, ou encore un sous-programme. C'est au programmeur de « sélectionner » ces suites d'instructions et d'en faire des fonctions. Pour exécuter une fonction, il suffit d'exécuter un branchement dont l'adresse de destination est celle de la fonction : on dit qu'on appelle la fonction. Toute fonction se termine par une instruction de retour, qui permet au processeur de revenir là où il en était avant d'appeler la fonction. Certains processeurs disposent d'une instruction de retour dédiée, alors que d'autres l'émulent à partir d'autres instructions. L'instruction de retour a besoin de connaître l'adresse de retour, l'adresse de la suite du programme. Celle-ci est sauvegardée soit par l'instruction d'appel de la fonction, soit par une instruction d'écriture spécialisée.



La pile d'appel de fonctions

Pour pouvoir exécuter plusieurs fonctions imbriquées les unes dans les autres, on doit sauvegarder plusieurs adresses de retour dans l'ordre. Pour cela, on utilise une **pile d'adresses de retour** : l'adresse de retour de la fonction en cours est toujours située au sommet de la pile. Certains processeurs émulent cette pile avec des registres généraux, ou sauvegardent une partie de la pile dans des registres cachés au lieu de tout mémoriser en RAM.

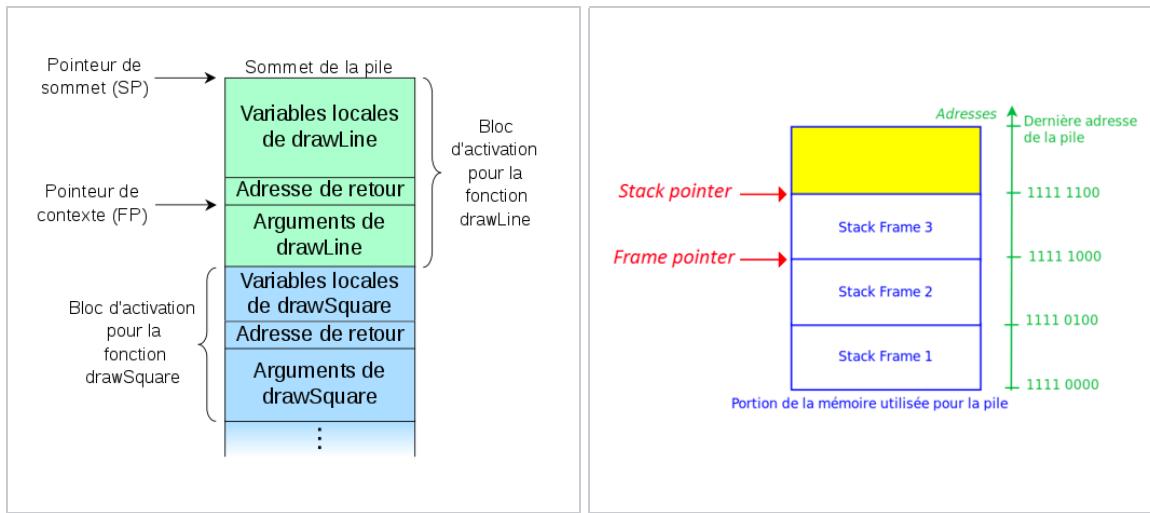
Lorsqu'une fonction s'exécute, elle utilise des registres et écrase souvent leur contenu. Pour éviter de perdre ces données écrasées, on doit en conserver une copie : on remet les registres dans leur état original une fois la fonction terminée. Pour cela, on utilise une **pile de registres**, qui est souvent fusionnée avec la pile d'adresse de retour. Plus un processeur a de registres architecturaux, plus cette sauvegarde prend du temps. Pour limiter le temps de sauvegarde des registres, certains processeurs utilisent le **fenêtrage de registres**, une technique qui permet d'intégrer cette pile de registre directement dans les registres du processeur. Cette technique duplique chaque registre architectural en plusieurs exemplaires qui portent le même nom. Chaque ensemble de registre architectural forme une fenêtre de registre, qui contient autant de registres qu'il y a de registres architecturaux. Lorsqu'une fonction s'exécute, elle se réserve une fenêtre inutilisée, et peut utiliser les registres de la fenêtre comme bon lui semble : une fonction manipule le registre architectural de la fenêtre qu'elle a réservé, mais pas les registres avec le même nom dans les autres fenêtres. Ainsi, pas besoin de sauvegarder les registres de cette fenêtre, vu qu'ils étaient vides de toute donnée. S'il ne reste pas de fenêtre inutilisée, on est obligé de sauvegarder les registres d'une fenêtre dans la pile.



Enfin, le processeur dispose souvent d'une troisième pile, souvent fusionnée avec les deux précédentes, qui contient des données utiles pour chaque fonction. Par exemple, certaines valeurs calculées hors de la fonction lui sont communiquées : ce sont des **arguments** ou paramètres. De même, la fonction peut calculer un ou plusieurs résultats, appelés **valeurs de retour**. On peut communiquer ces valeurs de deux manières, soit en en fournissant une copie, soit en fournissant leur adresse. Dans les deux cas, cette transmission peut se faire par copie soit sur la pile, soit dans les registres. Dans le cas d'un passage par les registres, ceux-ci ne sont pas sauvegardés lors de l'appel de la fonction (pour les arguments), et ne sont pas effacés lors du retour de la fonction (pour la valeur de retour). Généralement, le passage par la pile est très utilisé sur les processeurs CISC avec peu de registres, alors que les processeurs RISC privilégient le passage par les registres à cause de leur grand nombre de registres.

De plus, une fonction peut calculer des données temporaires, souvent appelées des **variables locales**. Une solution pour gérer ces variables consiste à réserver une portion de la mémoire statique pour chaque, dédiée au stockage de ces variables locales, mais cela gêne mal le cas où une fonction s'appelle elle-même (fonctions récursives). Une autre solution est de réserver un cadre de pile pour les variables locales. Le processeur dispose de modes d'adressages spécialisés pour adresser les variables automatiques d'un cadre de pile. Ces derniers ajoutent une constante au stack pointer.

Pour gérer les fonctions, certains processeurs possèdent deux ou plusieurs piles spécialisées pour les adresses de retour, les registres à sauvegarder, les paramètres et les variables locales. Mais d'autres processeurs ne possèdent qu'une seule pile à la fois pour les adresses de retour, les variables locales, les paramètres, et le reste. Pour cela, il faut utiliser une pile avec des cadres de pile de taille variable, dans lesquels on peut ranger plusieurs données. Pour localiser une donnée dans un cadre de pile pareil, on utilise sa position par rapport au début ou la fin du cadre de pile : on peut donc calculer l'adresse de la donnée en additionnant cette position avec le contenu du stack pointer. Pour gérer ces piles, on a besoin de sauvegarder deux choses : l'adresse à laquelle commence notre cadre de pile en mémoire, et de quoi connaître l'adresse de fin. Et il existe diverses façons de faire. Pour ce faire, on peut rajouter un registre, le frame pointer, qui sert à dire à quelle adresse commence le cadre de pile au sommet de celle-ci. D'autres processeurs arrivent à se passer de frame pointer, et se contentent de l'adresse de fin du cadre de pile (stack pointer), et de sa longueur. Cette solution est idéale si le cadre de pile a toujours la même taille.



Pile exécution contenant deux cadres de pile, un pour la fonction drawLine() et un autre pour la fonction drawSquare().

Frame pointer.

Opcodes et modes d'adressage

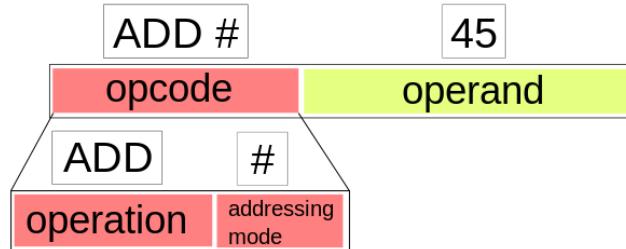
Les instructions sont stockées dans la mémoire sous la forme de suites de bits, tout comme les données. Cette suite de bits n'est pas organisée n'importe comment. Quelques bits de l'instruction indiquent quelle est l'opération à effectuer : est-ce une instruction d'addition, de soustraction, un branchement inconditionnel, un appel de fonction, une lecture en mémoire, etc. Cette portion de mémoire s'appelle l'**opcode**. Pour la même instruction, l'opcode peut être différent suivant le processeur, ce qui est source d'incompatibilités. Il existe certains processeurs qui utilisent une seule et unique instruction, et qui peuvent se passer d'opcodes.

Il arrive que certaines instructions soient composées d'un opcode, sans rien d'autre : elles ont alors une représentation en binaire qui est unique. Mais certaines instructions ajoutent une partie variable pour préciser la localisation des données à manipuler. Reste à savoir comment interpréter cette partie variable : après tout, c'est une simple suite de bits qui peut représenter une adresse, un nombre, un nom de registre, etc. Chaque manière d'interpréter la partie variable s'appelle un **mode d'adressage**. Pour résumer, ce mode d'adressage est une sorte de recette de cuisine capable de dire où se trouve la ou les données nécessaires pour exécuter une instruction. De plus, notre mode d'adressage peut aussi préciser où stocker le résultat de l'instruction. Ces modes d'adressage dépendent fortement de l'instruction qu'on veut faire exécuter et du processeur. Certaines instructions supportent certains modes d'adressage et pas d'autres, voir mixent plusieurs modes d'adresses : les instructions manipulant plusieurs données peuvent parfois utiliser un mode d'adressage différent pour chaque donnée. Dans de tels cas, tout se passe comme si l'instruction avait plusieurs parties variables, nommées **opérandes**, contenant chacune soit une adresse, soit une donnée, soit un registre.

Il existe deux méthodes pour préciser le mode d'adressage utilisé par l'instruction. Dans le premier cas, l'instruction ne gère qu'un mode d'adressage par opérande. Par exemple, toutes les instructions arithmétiques ne peuvent manipuler que des registres. Dans un cas pareil, pas besoin de préciser le mode d'adressage, qui est déduit automatiquement via l'opcode : on parle de **mode d'adressage implicite**. Dans le second cas, les instructions gèrent plusieurs modes d'adressage par opérande. Par exemple, une instruction d'addition peut additionner soit deux registres, soit un registre et une adresse, soit un registre et une constante. Dans un cas pareil, l'instruction doit préciser le mode d'adressage utilisé, au moyen de quelques bits intercalés entre l'opcode et les opérandes. On parle de **mode d'adressage explicite**. Sur certains processeurs, chaque instruction peut utiliser tous les modes d'adressage supportés par le processeur : on dit que le processeur est orthogonal.

machine instruction

ADD # 45



Une instruction va prendre un certain nombre de bits en mémoire : on dit aussi qu'elle a une certaine longueur. Cette longueur dépend de la longueur de l'opcode et de sa partie variable, qui n'ont pas forcément la même taille. Par exemple, un opérande contenant une adresse mémoire prendra plus de place qu'un opérande spécifiant un registre (il y a moins de registres que d'adresses). Sur certains processeurs, cette longueur d'une instruction est variable, toutes les instructions n'ayant pas la même taille. Cela permet de gagner un peu de mémoire : avoir des instructions qui font entre 2 et 3 octets est plus avantageux que de tout mettre sur 3 octets. Sur d'autres processeurs, cette longueur est fixe : cela gâche un peu de mémoire, mais permet au processeur de calculer plus facilement l'adresse de l'instruction suivante.

Pour comprendre un peu mieux ce qu'est un mode d'adressage, voyons quelques exemples de modes d'adresses assez communs et qui reviennent souvent.

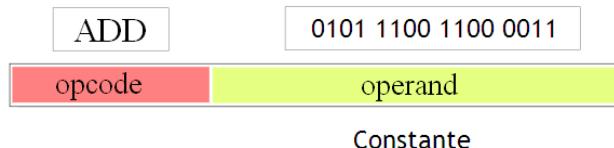
Adressage implicite

Avec l'adressage implicite, la partie variable n'existe pas ! Il peut y avoir plusieurs raisons à cela. Il se peut que l'instruction n'ait pas besoin de données : une instruction de mise en veille de l'ordinateur, par exemple. Ensuite, certaines instructions n'ont pas besoin qu'on leur donne la localisation des données d'entrée et « savent » où sont les données. Comme exemple, on pourrait citer une instruction qui met tous les bits du registre d'état à zéro. Certaines instructions manipulant la pile sont adressées de cette manière : on sait d'avance dans quels registres sont stockées l'adresse de la base ou du sommet de la pile.

Adressage immédiat

Avec l'adressage immédiat, la partie variable est une constante : un nombre entier, un caractère, un nombre flottant, etc. Avec ce mode d'adressage, la donnée est placée dans la partie variable et est chargée en même temps que l'instruction. Ces constantes sont souvent codées sur 8 ou 16 bits : aller au-delà serait inutile vu que la quasi-totalité des constantes manipulées par des opérations arithmétiques sont très petites et tiennent dans un ou deux octets.

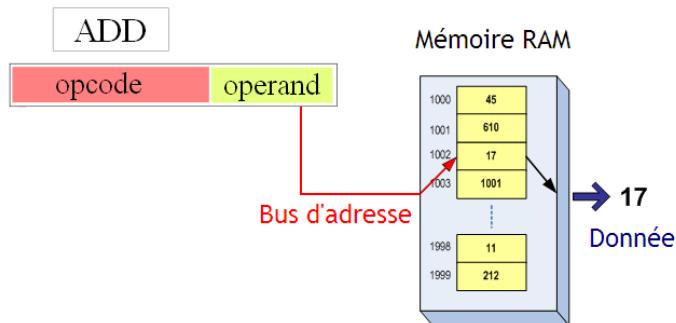
machine instruction



Adressage direct

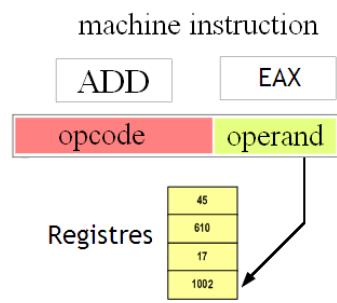
Passons maintenant à l'adressage absolu, aussi appelé adressage direct. Avec lui, la partie variable est l'adresse de la donnée à laquelle accéder. Cela permet de lire une donnée directement depuis la mémoire sans devoir la copier dans un registre.

machine instruction

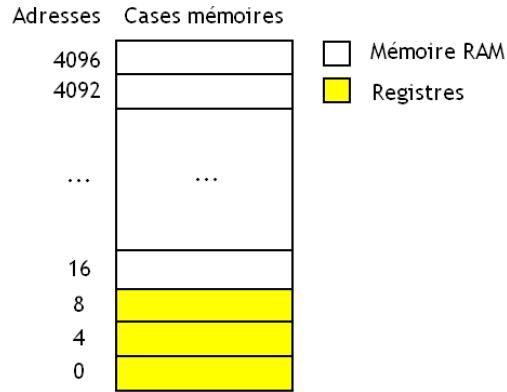


Adressage inhérent

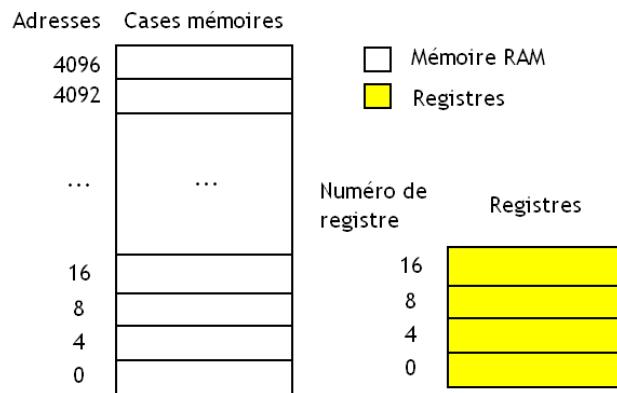
Avec le mode d'adressage inhérent, la partie variable va identifier un registre qui contient la donnée voulue.



Mais identifier un registre peut se faire de différentes façons. Certains processeurs détournent un petit nombre d'adresses mémoires pour les attribuer à des registres. Typiquement, les premières adresses mémoires sont redirigées vers les registres au lieu de servir à adresser une ROM ou une RAM. On est alors dans un cas particulier d'adressage direct.



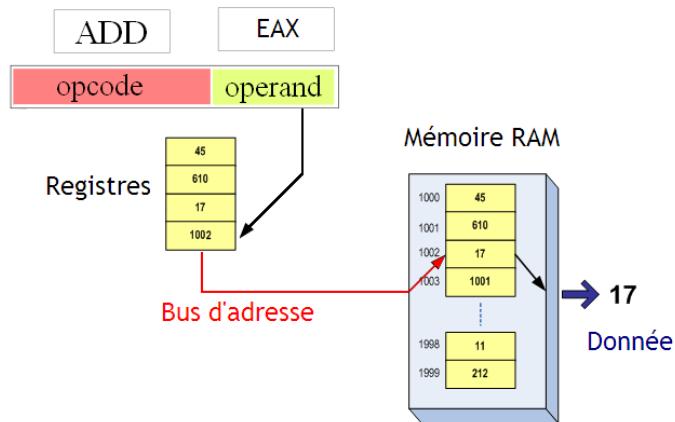
La seconde méthode demande d'attribuer un numéro spécialisé à chaque registre : le **nom de registre**. Celui-ci sera alors utilisé pour préciser à quel registre le processeur doit accéder. Cela demande d'ajouter un nouveau mode d'adressage, spécialisé pour les registres, afin d'utiliser ces noms de registre : le mode d'adressage inhérent.



Adressage indirect à registre

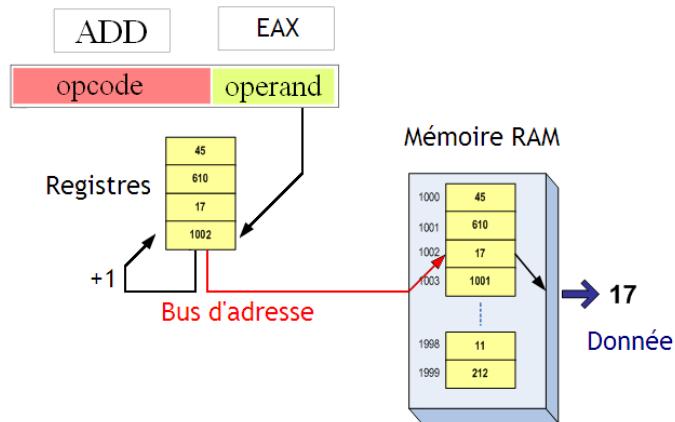
Dans certains cas, les registres généraux du processeur peuvent stocker des adresses mémoire. On peut alors décider d'accéder à l'adresse qui est stockée dans un registre : c'est le rôle du mode d'adressage indirect à registre. Ici, la partie variable identifie un registre contenant l'adresse de la donnée voulue. La différence avec le mode d'adressage inhérent vient de ce qu'on fait de ce nom de registre : avec le mode d'adressage inhérent, le registre indiqué dans l'instruction contiendra la donnée à manipuler, alors qu'avec le mode d'adressage indirect à registre, le registre contiendra l'adresse de la donnée. Le mode d'adressage indirect à registre permet d'implémenter de façon simple ce qu'on appelle les pointeurs.

machine instruction



Pour faciliter ces parcours de tableaux, on a inventé les modes d'adressages indirect avec auto-incrément (register indirect autoincrement) et indirect avec auto-décrément (register indirect autodecrement), des variantes du mode d'adressage indirect qui augmentent ou diminuent le contenu du registre d'une valeur fixe automatiquement. Cela permet de passer directement à l'élément suivant ou précédent dans un tableau.

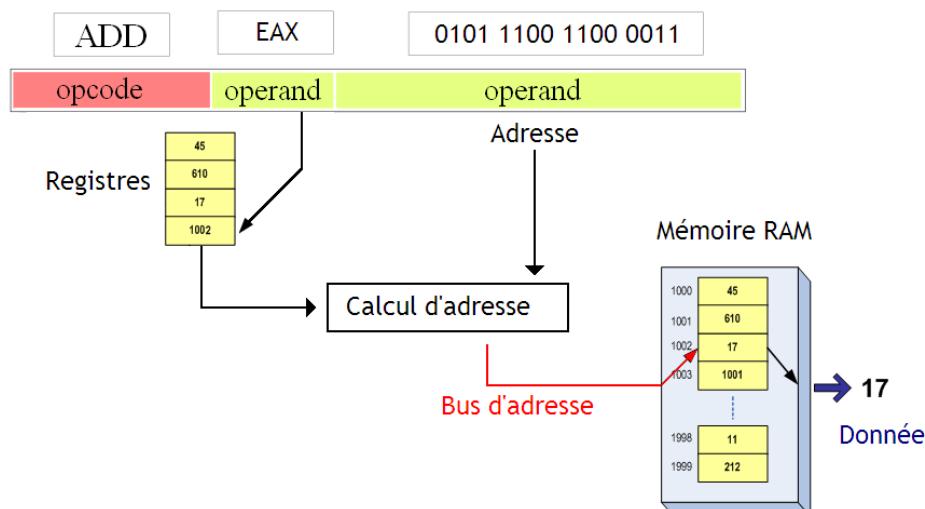
machine instruction



Adressage indexed absolute

D'autres modes d'adressage permettent de faciliter le calcul de l'adresse d'un élément du tableau. Pour éviter d'avoir à calculer les adresses à la main avec le mode d'adressage indirect à registre, on a inventé un mode d'adressage pour combler ce manque : le mode d'adressage indexed absolute. Celui-ci fournit l'adresse de base du tableau, et un registre qui contient l'indice. À partir de ces deux données, l'adresse de l'élément du tableau est calculée, envoyée sur le bus d'adresse, et l'élément est récupéré.

machine instruction

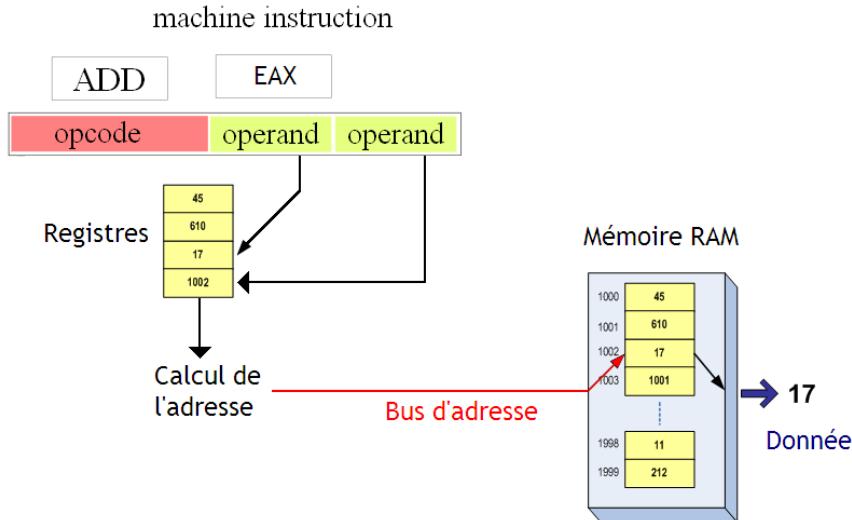


Ce mode d'adressage indexed absolute ne marche que pour des tableaux dont l'adresse est fixée une bonne fois pour toute. Ces tableaux sont assez rares : ils correspondent aux tableaux de taille fixe, déclarée dans la mémoire statique (souvenez-vous de la section précédente).

Adressage base + index

La majorité des tableaux sont des tableaux dont l'adresse n'est pas connue lors de la création du programme : ils sont déclarés sur la pile ou dans le tas, et leur adresse varie à chaque exécution du programme. On peut certes régler ce problème en utilisant du code automodifiant, mais ce serait vendre son âme au diable ! Pour contourner les limitations du mode d'adressage indexed absolute, on a inventé le mode d'adressage base + index.

Avec ce dernier, l'adresse du début du tableau n'est pas stockée dans l'instruction elle-même, mais dans un registre. Elle peut donc varier autant qu'on veut. Ce mode d'adressage spécifie deux registres dans sa partie variable : un registre qui contient l'adresse de départ du tableau en mémoire, le registre de base, et un qui contient l'indice, le registre d'index. Le processeur calcule alors l'adresse de l'élément voulu à partir du contenu de ces deux registres, et accède à notre élément. En clair : notre instruction ne fait pas que calculer l'adresse de l'élément : elle va aussi le lire ou l'écrire.

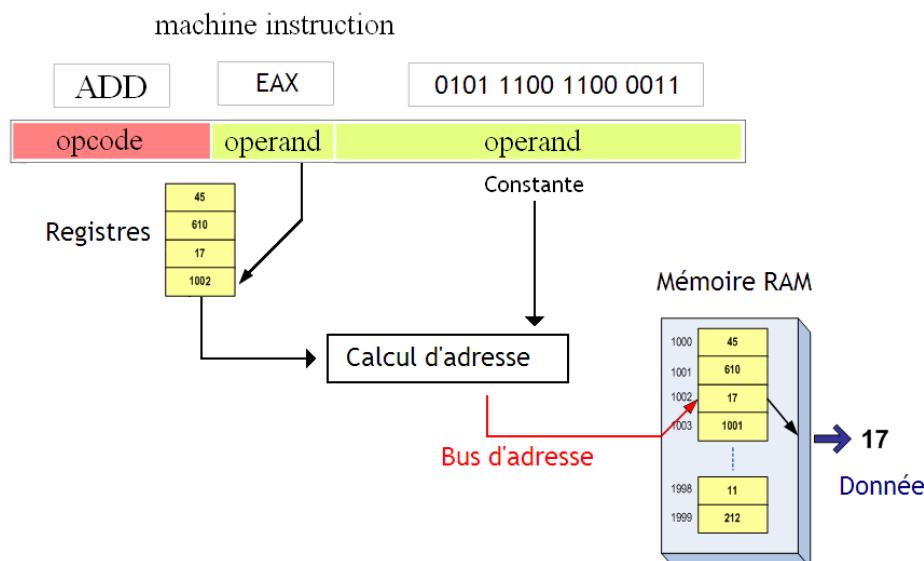


Ce mode d'adressage possède une variante qui permet de vérifier qu'on ne « déborde » pas du tableau, en calculant par erreur une adresse en dehors du tableau, à cause d'un indice erroné, par exemple. Accéder à l'élément 25 d'un tableau de seulement 5 éléments n'a pas de sens et est souvent signe d'une erreur. Pour cela, l'instruction peut prendre deux opérandes supplémentaires (qui peuvent être constants ou placés dans deux registres). L'instruction BOUND sur le jeu d'instruction x86 en est un exemple. Si cette variante n'est pas supportée, on doit faire ces vérifications à la main.

Adressage base + décalage

Outre les tableaux, les programmeurs utilisent souvent ce qu'on appelle des structures. Ces structures servent à créer des données plus complexes que celles que le processeur peut supporter. Mais le processeur ne peut pas manipuler ces structures : il est obligé de manipuler les données élémentaires qui la constituent une par une. Pour cela, il doit calculer leur adresse, ce qui n'est pas très compliqué. Une donnée a une place prédéterminée dans une structure : elle est donc à une distance fixe du début de celle-ci.

Calculer l'adresse d'un élément de notre structure se fait donc en ajoutant une constante à l'adresse de départ de la structure. Et c'est ce que fait le mode d'adressage base + décalage. Celui-ci spécifie un registre qui contient l'adresse du début de la structure, et une constante. Ce mode d'adressage va non seulement effectuer ce calcul, mais il va aussi aller lire (ou écrire) la donnée adressée.



Adressage base + index + décalage

Certains processeurs vont encore plus loin : ils sont capables de gérer des tableaux de structures ! Ce genre de prouesse est possible grâce au mode d'adressage base + index + décalage. Avec ce mode d'adressage, on peut calculer l'adresse d'une donnée placée dans un tableau de structure assez simplement : on calcule d'abord l'adresse du début de la structure avec le mode d'adressage base + index, et ensuite on ajoute une

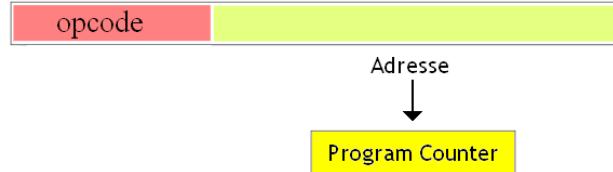
constante pour repérer la donnée dans la structure. Et le tout, en un seul mode d'adressage.

Modes d'adressage des branchements

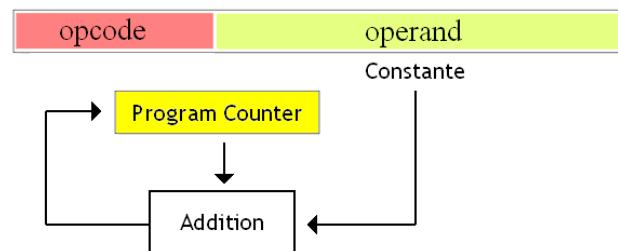
Les instructions de branchement peuvent avoir trois modes d'adresses :

- direct ;
- relatif ;
- indirect.

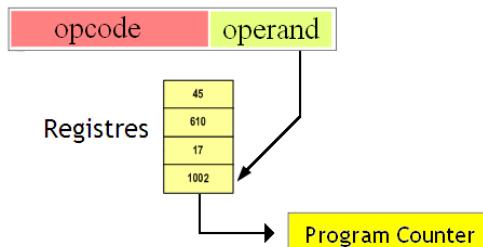
Avec un **branchement direct**, l'opérande est simplement l'adresse de l'instruction à laquelle on souhaite reprendre.



Avec un **branchement relatif**, l'opérande est un nombre qu'il faut ajouter au registre d'adresse d'instruction pour tomber sur l'adresse voulue. On appelle ce nombre un décalage (offset). De tels branchements sont appelés des branchements relatifs. Ces branchements permettent de localiser la destination d'un branchement par rapport à l'instruction en cours : cela permet de dire « le branchement est 50 instructions plus loin ».



Il existe un dernier mode d'adressage pour les branchements : l'adresse vers laquelle on veut brancher est stockée dans un registre. On appelle de tels branchements des **branchements indirects**. Avec de tels branchements, l'adresse vers laquelle on souhaite brancher peut varier au cours de l'exécution du programme. Ces branchements sont souvent camouflés dans des fonctionnalités un peu plus complexes des langages de programmation (pointeurs sur fonction, chargement dynamique de bibliothèque, structure de contrôle switch, et ainsi de suite).

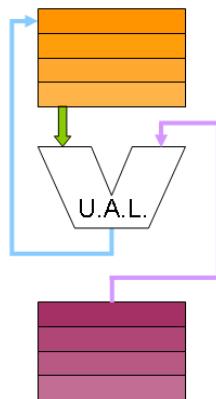


Jeux d'instructions

Les instructions d'un processeur dépendent fortement du processeur utilisé. La liste de toutes les instructions qu'un processeur peut exécuter s'appelle son **jeu d'instructions**. Celui-ci définit les instructions supportées, ainsi que la manière dont elles sont encodées en mémoire. Il existe différents jeux d'instructions, que l'on peut classer suivant divers critères.

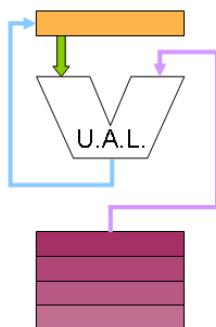
Adressage des opérandes

La première classification que nous allons voir est celle basée sur l'adressage des opérandes, et les transferts de données entre mémoire, unité de calcul, et registres. Les **processeurs à registres** peuvent stocker temporairement des données dans des registres généraux ou spécialisés. Pour échanger des données entre la mémoire et les registres, on peut utiliser une instruction à tout faire : MOV. Sur d'autres, on utilise des instructions séparées pour copier une donnée de la mémoire vers un registre (LOAD), copier le contenu d'un registre dans un autre, copier le contenu d'un registre dans la mémoire RAM (STORE), etc. Sur les **architectures load-store**, seules deux instructions peuvent accéder à la mémoire RAM : LOAD pour les lectures et STORE pour les écritures.

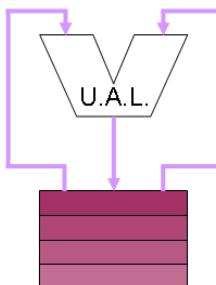


Ces architectures peuvent se classer en deux sous-catégories : les architectures à deux et trois adresses. Sur les **architectures à deux adresses**, le résultat d'une instruction est stocké à l'endroit indiqué par la référence du premier opérande : cette donnée sera remplacée par le résultat de l'instruction. Avec cette organisation, les instructions ne précisent que deux opérandes. Mais la gestion des instructions est moins souple, vu qu'un opérande est écrasé. Sur les **architectures à trois adresses**, on peut préciser le registre de destination du résultat. Ce genre d'architectures permet une meilleure utilisation des registres, mais les instructions deviennent plus longues que sur les architectures à deux adresses.

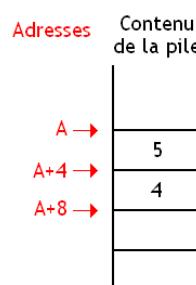
Certains processeurs n'utilisent qu'un seul registre pour les données : l'**accumulateur**. Toute instruction arithmétique va lire un opérande depuis cet accumulateur, et y écrire son résultat. Si l'instruction a besoin de plusieurs opérandes, les opérandes qui ne sont pas dans l'accumulateur sont lus depuis la mémoire ou dans des registres séparés de l'accumulateur.



Les toutes premières machines n'avaient pas de registres pour les données et ne faisaient que manipuler la mémoire RAM ou ROM : on parle d'**architectures mémoire-mémoire**.



On peut notamment citer les **machines à pile**, où les données que le processeur manipule sont regroupées dans une pile. Précisément, elles sont regroupées dans ce qu'on appelle des cadres de pile, qui regroupent plusieurs mots mémoire contigus (placés les uns à la suite des autres). Ces cadres de pile ne peuvent contenir que des données simples, comme des nombres entiers, des flottants, des caractères. Sur certaines machines à pile, tout ou partie de la pile est stockée directement dans des registres internes au processeur, pour gagner en performance.

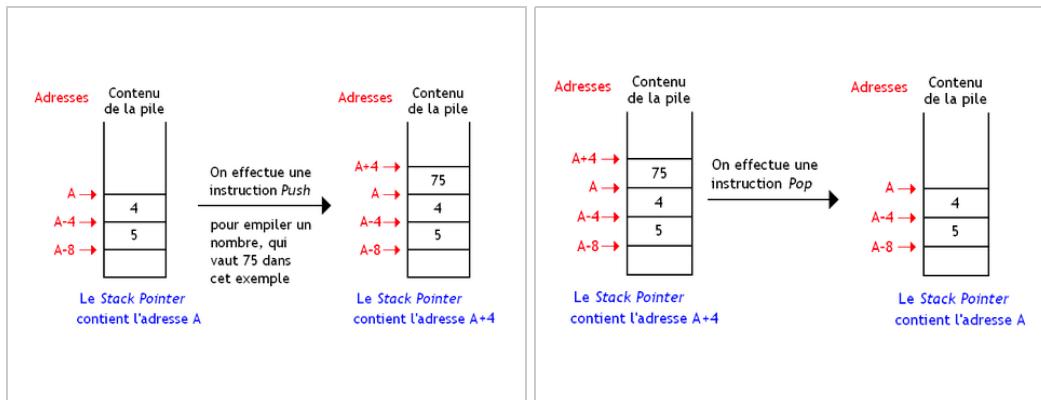


Ces cadres de pile sont créés un par un, et sont placés consécutivement dans la mémoire. On peut comparer cette pile à une pile d'assiettes : on peut ajouter une assiette au sommet de la pile ou enlever celle qui est au sommet, mais pas toucher aux autres. Sur la pile d'un ordinateur, c'est la même chose : on peut accéder au cadre de pile au sommet de la pile, le rajouter ou l'enlever, mais pas toucher aux autres cadres de pile. Si vous regardez bien, vous remarquerez que la donnée au sommet de la pile est la dernière donnée à avoir été ajoutée (empilée) sur la pile. Ce sera aussi la prochaine donnée à être dépiler (si on n'empile pas de données au dessus). Conséquence : les données sont dépiler dans l'ordre inverse d'empilement. Ces machines ont besoin d'un registre pour stocker l'adresse du sommet de la pile : je vous présente donc le stack pointer, ou pointeur de pile.

Le nombre de manipulations possibles sur cette pile se résume donc à trois manipulations de base :

- détruire le cadre de pile au sommet de la pile : on dépile ;
- créer un cadre de pile au sommet de la pile : on empile ;
- récupérer les données stockées du cadre de pile au sommet de la pile.

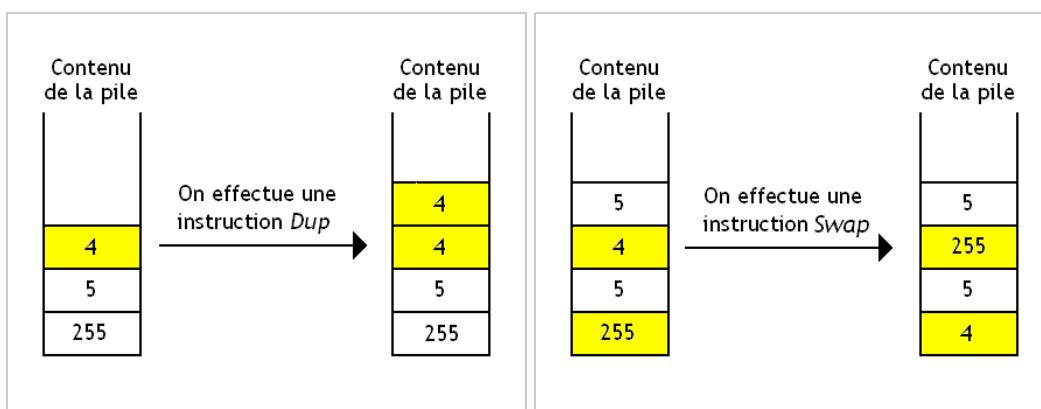
Sur une machine à pile, les seules données manipulables par une instruction sont celles placées au sommet de la pile : il faut empiler les opérandes un par un avant d'exécuter l'instruction. L'instruction dépile automatiquement les opérandes qu'elle utilise et empile son résultat. Pour empiler une donnée, le processeur fourni une instruction PUSH qui prend l'adresse de la donnée à empiler, charge la donnée, et met à jour le stack pointer. L'instruction POP dépile la donnée au sommet de la pile, la stocke à l'adresse indiquée dans l'instruction, et met à jour le stack pointer.



Instruction Push.

Instruction Pop.

Vu qu'une instruction dépile ses opérandes, on ne peut pas les réutiliser. Ceci dit, certaines instructions ont été inventées pour limiter la casse : on peut notamment citer l'instruction DUP, qui copie le sommet de la pile en deux exemplaires. On peut aussi citer l'instruction SWAP, qui échange deux données dans la pile.



Instruction Dup.

Instruction Swap.

Les machines à pile que je viens de décrire ne peuvent manipuler que des données sur la pile : ces machines à pile sont ce qu'on appelle des **machines à zéro adresse**. Avec une telle architecture, les instructions sont très petites, car il n'y a pas besoin de bits pour indiquer la localisation des données dans la mémoire, sauf pour POP et PUSH. Toutefois, certaines machines à pile plus évoluées autorisent certaines instructions à préciser l'adresse mémoire d'un (voire plusieurs, dans certains cas) de leurs opérandes. Ces machines sont appelées des **machines à pile à une adresse**.

RISC vs CISC

La seconde classification des jeux d'instructions que nous allons aborder se base sur le nombre d'instructions et classe nos processeurs en deux catégories :

- les RISC (reduced instruction set computer), au jeu d'instruction simple ;
- et les CISC (complex instruction set computer), qui ont un jeu d'instruction étoffé.

Propriété	CISC	RISC
Nombre d'instructions	Élevé, parfois plus d'une centaine.	Faible, moins d'une centaine.
Type d'instructions	Beaucoup d'instructions complexes, qui prennent plus d'un cycle d'horloge : fonctions trigonométriques, gestion de texte, calculs cryptographiques, compression de données, etc.	Pas d'instruction complexe : toutes les instructions prennent un cycle (hors accès mémoire).
Types de données	Supportent des types de données complexes : texte via certaines instructions, graphes, listes chaînées, etc.	Types supportés limités aux entiers (adresses comprises) et flottants.
Modes d'adressage	Gère un grand nombre de modes d'adressages : indirect avec auto-incrément/décrément, inhérent, indexed absolute, etc.	Peu de modes d'adressage : les modes d'adressages complexes ne sont pas pris en charge.
Nombre d'accès mémoire par instruction	Possibilité d'effectuer plusieurs accès mémoire par instruction, avec certains modes d'adressage.	Pas plus d'un accès mémoire par instruction.
Load-store	Non.	Oui.
Spécialisation des registres	Présence de registres spécialisés : registres flottants, registres d'adresses, registres d'index, registres pour les compteurs de boucle, etc.	Presque pas de registres spécialisés, et usage de registres généraux.
Nombre de registres	Peu de registres : rarement plus de 16 registres entiers.	Beaucoup de registres, souvent plus de 32.
Longueur des instructions	Instructions de taille variable, pour améliorer la densité de code.	Instructions de taille fixe pour simplifier le processeur.

Jeux d'instructions spécialisés

En parallèle de ces architectures CISC et RISC, d'autres classes de jeux d'instructions sont apparus. Ceux-ci visent des buts distincts, qui changent suivant le jeu d'instruction :

- soit ils cherchent à gagner en performance, en exécutant plusieurs instructions à la fois ;
- soit ils sont adaptés à certaines catégories de programmes ou de langages de programmation ;
- soit ils cherchent à diminuer la taille des programmes et à économiser de la mémoire ;
- soit ils tentent d'améliorer la sécurité des programmes et les rendent résistants aux attaques.

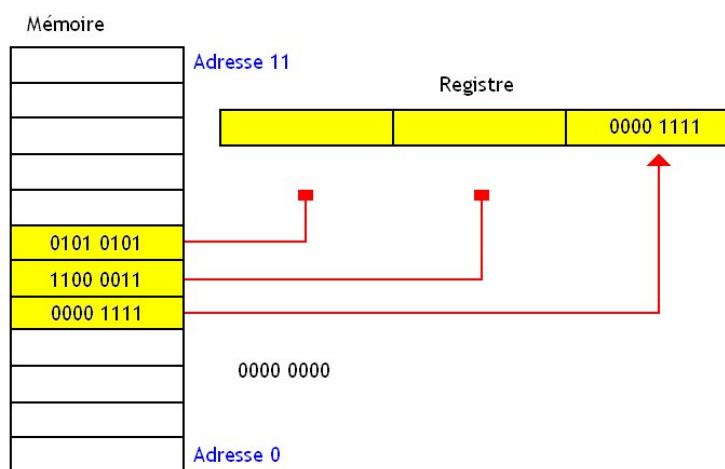
Certaines architectures sont conçues pour pouvoir exécuter plusieurs instructions en même temps, lors du même cycle d'horloge : elles visent le traitement de plusieurs instructions en parallèle, d'où leur nom d'**architectures parallèles**. Ces architectures visent la performance, et sont relativement généralistes, à quelques exceptions près. On peut, par exemple, citer les architectures very long instruction word, les architectures dataflow, les processeurs EDGE, et bien d'autres. D'autres instructions visent à exécuter une même instruction sur plusieurs données différentes : ce sont les instructions SIMD, vectorielles, et autres architectures utilisées sur les cartes graphiques actuelles. Nous verrons ces architectures plus tard dans ce tutoriel, dans les derniers chapitres.

Certains jeux d'instructions sont dédiés à des types de programmes bien spécifiques, et sont peu adaptés pour des programmes généralistes. On peut notamment citer les digital signal processors, conçus pour les applications de traitement d'image et de son (de traitement de signal), que nous aborderons plus tard dans ce cours. Certains processeurs sont carrément conçus pour un langage en particulier : par exemple, il existe des processeurs qui intègrent une machine virtuelle JAVA directement dans leurs circuits ! On appelle ces processeurs, conçus pour des besoins particuliers, des **processeurs dédiés**.

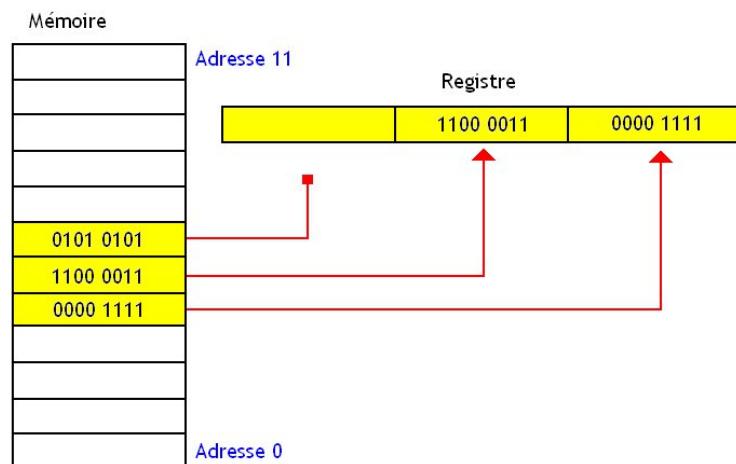
Certains chercheurs ont inventé des jeux d'instruction pour diminuer la taille des programmes. Certains processeurs disposent de deux jeux d'instructions : un compact, et un avec une faible densité de code. Il est possible de passer d'un jeu d'instructions à l'autre en plein milieu de l'exécution du programme, via une instruction spécialisée. D'autres processeurs sont capables d'exécuter des binaires compressés (la décompression a lieu lors du chargement des instructions dans le cache).

Alignement mémoire

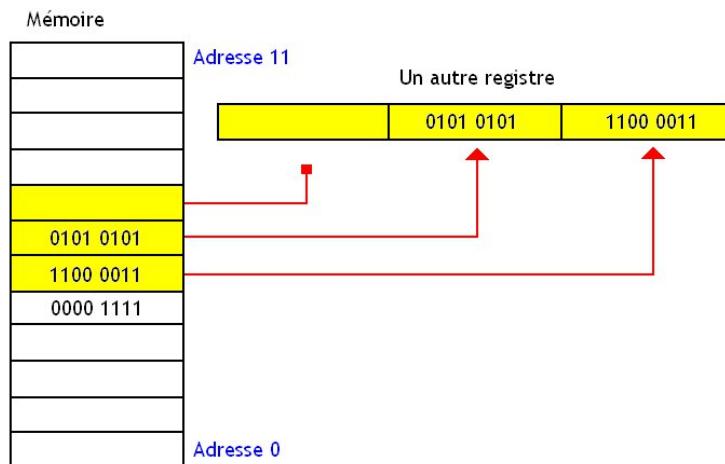
Il arrive que le bus de données ait une largeur de plusieurs mots mémoire : le processeur peut charger 2, 4 ou 8 mots mémoire d'un seul coup (parfois plus). Une donnée qui a la même taille que le bus de données est appelée un mot mémoire. Quand on veut accéder à une donnée sur un bus plus grand que celle-ci, le processeur ignore les mots mémoire en trop.



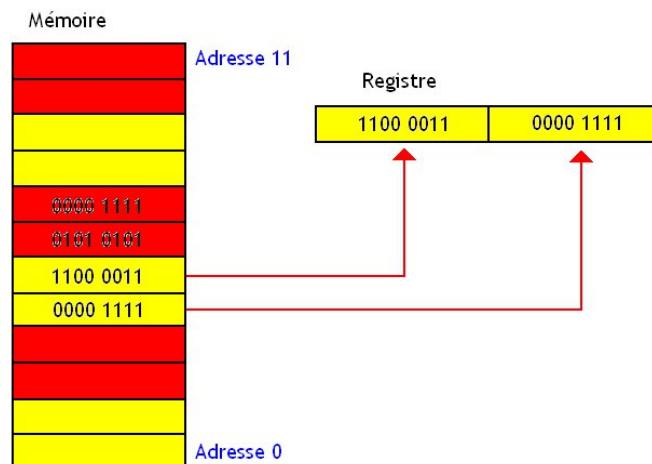
Sur certains processeurs, il existe des restrictions sur la place de chaque mot en mémoire, restrictions résumées sous le nom d'**alignement mémoire**. Sans alignement, on peut lire ou écrire un mot, peu importe son adresse. Pour donner un exemple, je peux parfaitement lire une donnée de 16 bits localisée à l'adresse 4, puis lire une autre donnée de 16 bits localisée à l'adresse 5 sans aucun problème.



Un peu plus tard...

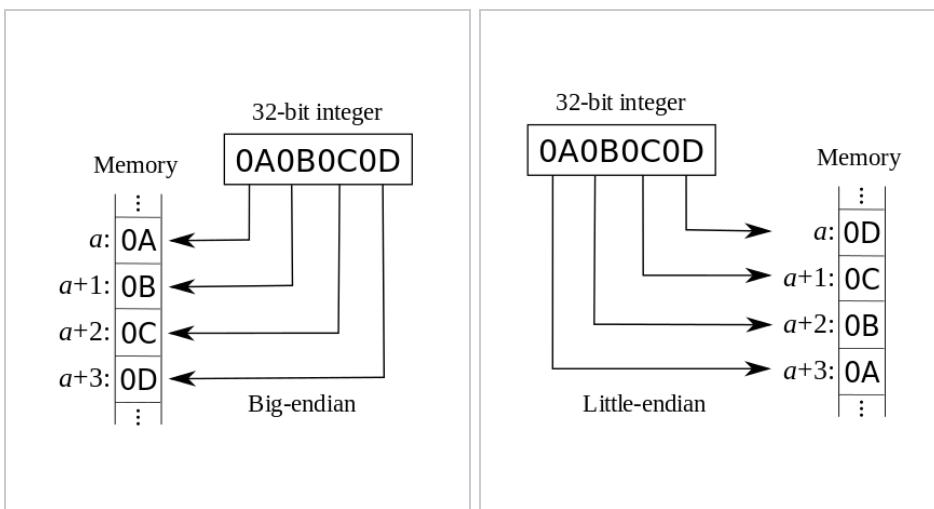


Mais d'autres processeurs imposent des restrictions dans la façon de gérer ces mots : ils imposent un alignement mémoire. Tout se passe comme si chaque mot mémoire de la mémoire avait la taille d'un mot : le processeur voit la mémoire comme si ces mots mémoire avaient la taille d'un mot, alors que ce n'est pas forcément le cas. Avec alignement, l'unité de lecture/écriture est le mot, pas le mot mémoire. Par contre, la capacité de la mémoire reste inchangée, ce qui fait que le nombre d'adresses utilisables diminue : il n'y a plus besoin que d'une adresse par mot mémoire et non par byte. L'adresse d'un mot est égale à l'adresse de son byte de poids faible, mais les autres ne sont pas adressables. Par exemple, si on prend un mot de 8 octets, on est certain qu'une adresse sur 8 disparaîtra.



Boutisme

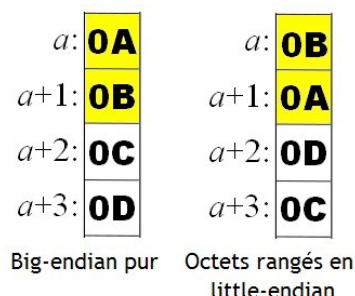
On peut introduire cet extrait par une analogie avec les langues humaines : certaines s'écrivent de gauche à droite et d'autres de droite à gauche. Dans un ordinateur, c'est pareil avec les octets des mots mémoire : on peut les écrire soit de gauche à droite, soit de droite à gauche. Quand on veut parler de cet ordre d'écriture, on parle de **boutisme** (endianness). Sur les **processeurs gros-boutistes**, l'octet de poids fort de notre donnée est stocké dans la case mémoire ayant l'adresse la plus faible. Sur les **processeurs petit-boutistes**, c'est l'inverse : l'octet de poids faible de notre donnée est stocké dans la case mémoire ayant l'adresse la plus faible. Certains processeurs sont un peu plus souples : ils laissent le choix du boutisme. Sur ces processeurs, on peut configurer le boutisme en modifiant un bit dans un registre du processeur : il faut mettre ce bit à 1 pour du petit-boutiste, et à 0 pour du gros-boutiste, par exemple. Ces processeurs sont dits **bi-boutistes**.



Gros-boutisme.

Petit-boutisme.

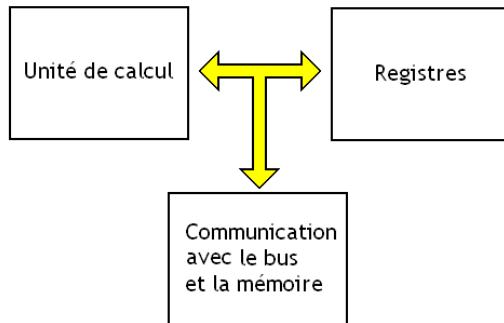
Certains processeurs sont toutefois un peu plus sadiques : ils utilisent des mots mémoire de plusieurs octets. Dans ce cas, il faut aussi prendre en compte le boutisme des octets dans le mot mémoire, qui peut être gros-boutiste ou petit-boutiste. Si l'ordre des mots mémoire et celui des octets dans le mot mémoire est identique, alors on retrouve du gros- ou petit-boutiste normal. Mais les choses changent si jamais l'ordre des mots mémoire et celui des octets dans le mot mémoire sont différents. Dans ces conditions, on doit préciser un ordre d'inversion des mots mémoire (byte-swap), qui précise si les octets doivent être inversés dans un mot mémoire processeur, en plus de préciser si l'ordre des mots mémoire est petit- ou gros-boutiste. Par exemple, comparons l'ordre des octets entre un nombre codé en gros-boutiste pur, et un nombre gros-boutiste dont les octets sont rangés dans un mot mémoire en petit-boutiste. Le nombre en question est **0x 0A 0B 0C 0D**, en hexadécimal, le premier mot mémoire étant indiqué en jaune, le second en blanc.



Les composants d'un processeur

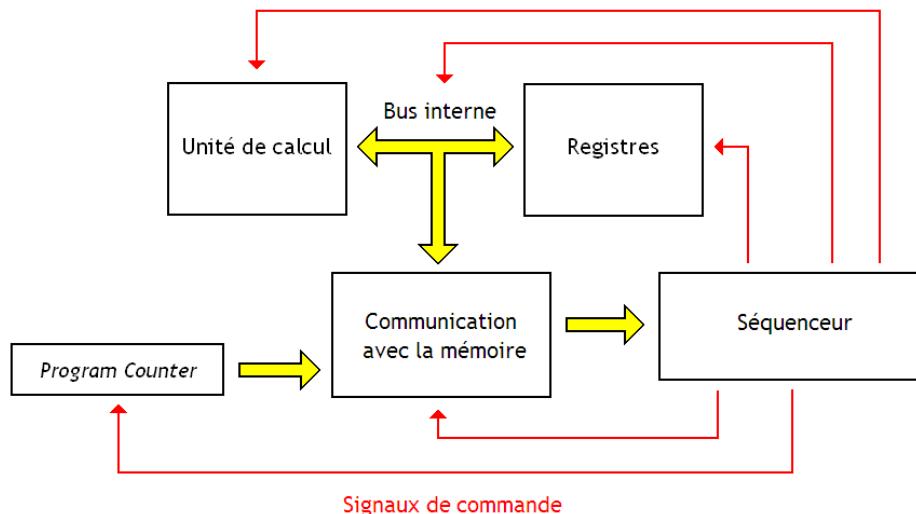
Dans le chapitre précédent, on a vu le processeur comme une boîte noire contenant des registres, qui exécutait des instructions les unes après les autres, et qui pouvait accéder à la mémoire. Dans ce chapitre, nous allons ouvrir cette boîte noire : nous allons aborder la **micro-architecture** du processeur.

Pour effectuer ces calculs, le processeur contient un circuit spécialisé : l'**unité de calcul**. De plus, le processeur contient des **registres**, ainsi qu'un circuit chargé des **communications avec la mémoire**. Les registres, l'unité de calcul, et les circuits de communication avec la mémoire sont reliés entre eux par un ensemble de fils afin de pouvoir échanger des informations : par exemple, le contenu des registres doit pouvoir être envoyé en entrée de l'unité de calcul, pour additionner leur contenu par exemple. Ce groupe de fils forme ce qu'on appelle le **bus interne du processeur**. L'ensemble formé par ces composants s'appelle le **chemin de données**. On l'appelle ainsi parce que c'est dans ce chemin de données que les données vont circuler et être traitées dans le processeur.



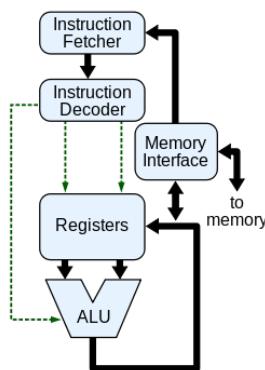
Si le chemin de données s'occupe de tout ce qui a trait aux données, il est complété par un circuit qui s'occupe de tout ce qui a trait aux instructions elles-mêmes. Ce circuit, l'**unité de contrôle** va notamment charger l'instruction dans le processeur, depuis la mémoire RAM. Il va ensuite configurer le chemin de données pour effectuer l'instruction. Il faut bien contrôler le mouvement des informations dans le chemin de données pour que les calculs se passent sans encombre. Pour cela, l'unité de contrôle contient un circuit : le **séquenceur**. Ce séquenceur envoie des signaux au chemin de données pour le configurer et le commander.

Il est évident que pour exécuter une suite d'instructions dans le bon ordre, le processeur doit savoir quelle est la prochaine instruction à exécuter : il doit donc contenir une mémoire qui stocke cette information. C'est le rôle du registre d'adresse d'instruction, aussi appelé **program counter**. Cette adresse ne sort pas de nulle part : on peut la déduire de l'adresse de l'instruction en cours d'exécution par divers moyens plus ou moins simples. Généralement, on profite du fait que le programmeur/compilateur place les instructions les unes à la suite des autres en mémoire, dans l'ordre où elles doivent être exécutées. Ainsi, on peut calculer l'adresse de la prochaine instruction en ajoutant la longueur de l'instruction chargée au program counter. Mais sur d'autres processeurs, chaque instruction précise l'adresse de la suivante. Ces processeurs n'ont pas besoin de calculer une adresse qui leur est fournie sur un plateau d'argent.



Pour exécuter une instruction, le processeur va effectuer trois étapes :

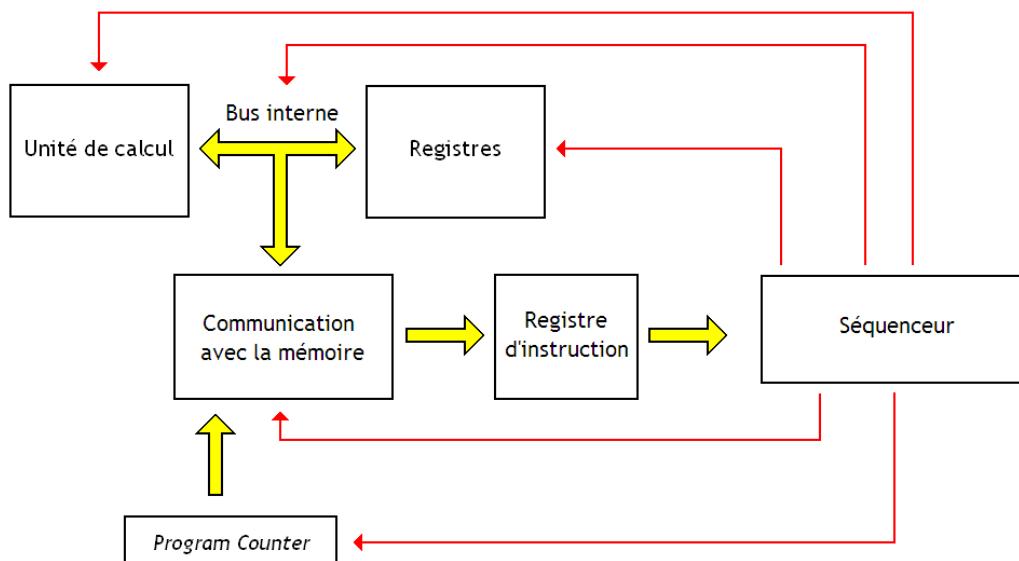
- l'unité de chargement va charger l'instruction depuis la mémoire : c'est l'étape de **chargement** (ou fetch) ;
- le séquenceur va ensuite « étudier » la suite de bits de l'instruction et en déduire comment configurer les circuits du processeur pour exécuter l'instruction voulue : c'est l'étape de **décodage** ;
- enfin, le séquenceur configure le chemin de données pour exécuter l'instruction : c'est l'étape d'**exécution**.



L'étape de chargement

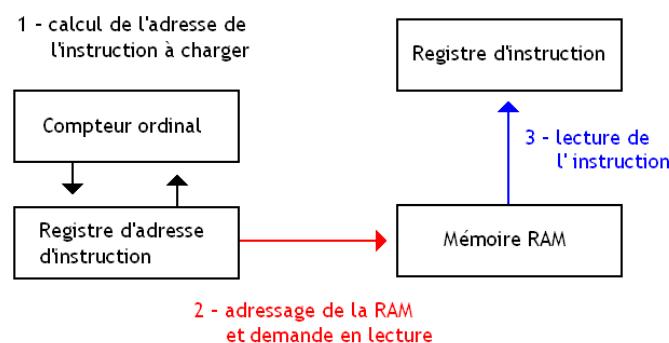
Au démarrage d'un programme, le program counter est initialisé à l'adresse de sa première instruction. Cette adresse n'est pas toujours l'adresse 0 : les premières adresses peuvent être réservées à des trucs assez importants, comme la pile ou le vecteur d'interruptions. Le program counter est mis à jour à la fin de chaque instruction pour le faire pointer sur l'adresse suivante. Dans la grosse majorité des cas, cette adresse est calculée par l'unité de calcul ou par un circuit dédié : le **compteur ordinal**.

Il faut noter que l'instruction chargée est parfois stockée dans un registre, pour mettre celle-ci à disposition du séquenceur. Ce registre est appelé le **registre d'instruction**. Cela arrive sur les processeurs ne disposant que d'une seule mémoire. Sans cela, pas d'accès mémoire aux données : le bus mémoire serait occupé en permanence par l'instruction chargée, et ne pourrait pas servir à charger ou écrire de données. Le bus de données serait donc inaccessible, rendant toute lecture ou écriture impossible. Les processeurs basés sur une architecture Harvard arrivent cependant à se passer de ce registre, vu que le bus des instructions est séparé du bus dédié aux données.



L'étape de chargement (ou fetch) est toujours décomposée en trois étapes :

- l'envoi du contenu du program counter sur le bus d'adresse ;
- la lecture de l'instruction sur le bus de données ;
- la mise à jour du *program counter* (parfois faite en parallèle de la seconde étape).



Mise à jour du program counter sans branchement

Sur certains processeurs assez rares, chaque instruction précise l'adresse de la suivante, qui est incorporée dans l'instruction. Mais sur la majorité

des processeurs, on profite du fait que les instructions sont placées dans l'ordre d'exécution dans la RAM. En faisant ainsi, on peut calculer l'adresse de la prochaine instruction en ajoutant la longueur de l'instruction au contenu du '*program counter*'. L'adresse de l'instruction en cours est connue dans le program counter, reste à connaître la longueur cette instruction. Lorsque les instructions ont toutes la même taille, ce calcul est simple : un simple additionneur suffit. Mais certains processeurs ont des instructions de taille variable, ce qui complique le calcul. Il y a plusieurs solutions à ce problème.

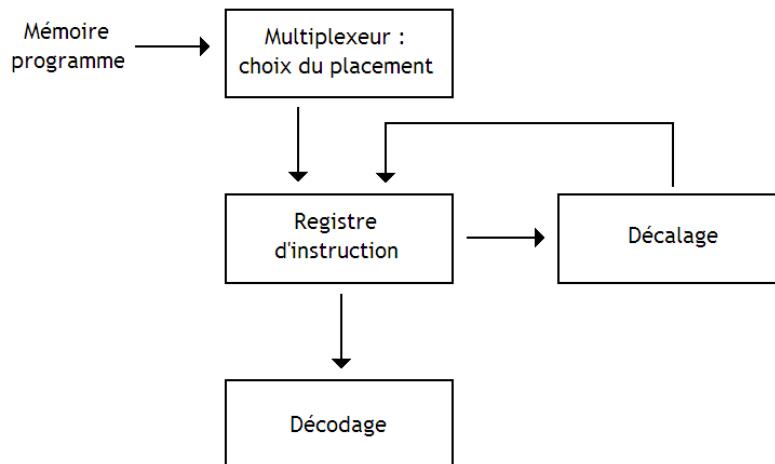
La plus simple consiste à indiquer la longueur de l'instruction dans une partie de l'opcode ou de la représentation en binaire de l'instruction.

Une autre solution consiste à charger l'instruction byte par byte, par morceaux de taille fixe. Ceux-ci sont alors accumulés les uns à la suite des autres dans le registre d'instruction, jusqu'à ce que le séquenceur détecte l'obtention d'une instruction complète. Le seul défaut de cette approche, c'est qu'il faut détecter quand une instruction complète a été chargée : un nouveau circuit est requis. Une solution similaire permet de se passer d'un registre d'instruction, en transformant le séquenceur en circuit séquentiel. Les bytes sont chargés à chaque cycle, faisant passer le séquenceur d'un état interne à un autre à chaque mot mémoire. Tant qu'il n'a pas reçu tous les mots mémoire de l'instruction, chaque mot mémoire non terminal le fera passer d'un état interne d'attente à un autre. S'il tombe sur le dernier mot mémoire d'une instruction, il rentre dans un état de décodage.

Et enfin, il existe une dernière solution, qui est celle qui est utilisée dans les processeurs haute performance de nos PC : charger un bloc de mots mémoire qu'on découpe en instructions, en déduisant leurs longueurs au fur et à mesure. Généralement, la taille de ce bloc est conçue pour être de la même longueur que l'instruction la plus longue du processeur : on est sûr de charger obligatoirement au moins une instruction complète, et peut-être même plusieurs. Dit autrement, les instructions doivent être alignées en mémoire (relisez le chapitre sur l'alignement des données en mémoire si besoin). Cette solution pose quelques problèmes : il se peut qu'on n'ait pas une instruction complète lorsque l'on arrive à la fin du bloc, mais seulement un morceau. On peut se retrouver avec des instructions à cheval entre deux blocs.

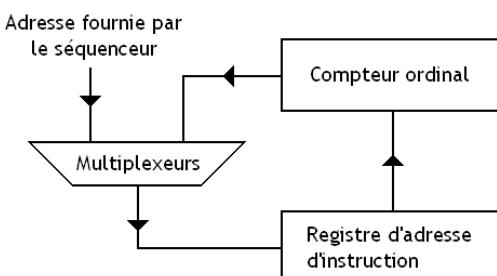


Dans ce cas, on doit décaler le morceau de bloc pour le mettre au bon endroit (au début du registre d'instruction), et charger le prochain bloc juste à côté. On a donc besoin d'un circuit qui décale tous les bits du registre d'instruction, couplé à un circuit qui décide où placer dans le registre d'instruction ce qui a été chargé, avec quelque circuits autour pour configurer les deux circuits précédents.

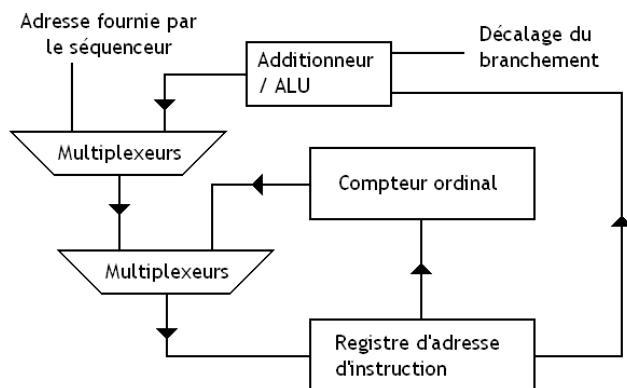


Unité de chargement avec branchements

Lors d'un branchement, l'adresse de destination du branchement est copiée dans le *program counter*. Il faut alors court-circuiter l'adresse calculée par le compteur ordinal si le branchement s'exécute. Pour les branchements directs, l'adresse de destination est fournie par le séquenceur. Pour les branchements indirects, il suffit de relier le *program counter* au bus interne au processeur, et de lire le registre voulu.



Pour prendre en compte les branchements relatifs, on a encore deux solutions : réutiliser l'ALU pour calculer l'adresse, ou rajouter un circuit qui fait le calcul.



Le séquenceur

Une fois que l'instruction a été chargé puis décodée, elle va s'exécuter. Pour s'exécuter, celle-ci va configurer le chemin de donnée. Seulement, une instruction, même simple, va souvent demander d'effectuer plusieurs étapes sur le chemin de donnée. Par exemple, le mode d'adressage base + index demande de calculer l'adresse avant la lecture, ce qui fait deux étapes : le calcul de l'adresse et l'accès mémoire. Prenons maintenant le cas d'une instruction d'addition dont les opérandes peuvent être des registres, des données placées en mémoire ou des constantes. Si celle-ci ne manipule que des registres, l'opération peut se faire en une seule sous-étape. Mais si un opérande est à aller chercher dans la mémoire, on doit rajouter une sous-étape pour le lire en mémoire. Et on peut aller plus loin en utilisant le mode d'adressage base + index, qui rajoute une étape de calcul d'adresse.

Bref, on voit bien que l'exécution d'une instruction s'effectue en plusieurs étapes distinctes, qui vont soit faire un calcul, soit échanger des données entre registres, soit communiquer avec la RAM. Chaque étape s'appelle une **micro-opération**, ou encore pinstruction. Toute instruction machine est équivalente à une suite de micro-opérations exécutée dans un ordre précis. Dit autrement, chaque instruction machine est traduite en suite de micro-opérations à chaque fois qu'on l'exécute. Certaines pinstructions font un cycle d'horloge, alors que d'autres peuvent prendre plusieurs cycles. Un accès mémoire en RAM peut prendre 200 cycles d'horloge et ne représenter qu'une seule pinstruction, par exemple. Même chose pour certaines opérations de calcul, comme des divisions ou multiplication, qui correspondent à une seule pinstruction mais prennent plusieurs cycles.

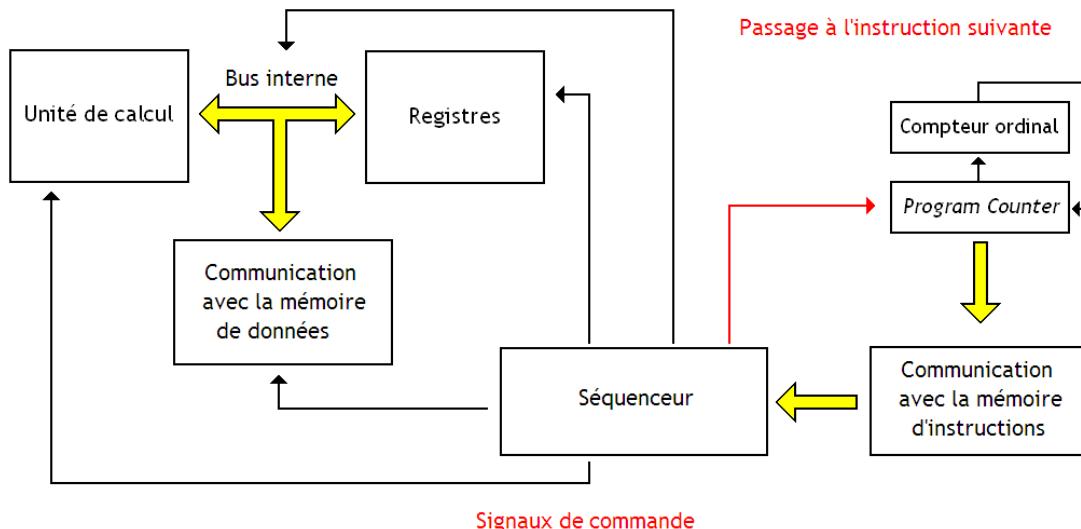
Comme on l'a vu plus haut, le chemin de données est rempli de composants à configurer d'une certaine manière pour exécuter une micro-instruction. Mais cela demande de déduire, pour l'instruction à exécuter, quelles sont les micro-opérations à exécuter et dans quel ordre. C'est le rôle de l'**unité de décodage d'instruction**, une portion du séquenceur qui « décode » l'instruction. Ce séquenceur va traduire une instruction en suite de micro-opération et configurer le chemin de données pour effectuer chaque micro-opération. Pour configurer le chemin de données, il va envoyer ce qu'il faut sur l'entrée de sélection de l'unité de calcul, les entrées du banc de registres, ou les circuits du bus du processeur. On appelle ces bits des **signaux de commande**. Lorsque nos unités de calcul (ou d'autres circuits du processeur) reçoivent un de ces signaux de commande, elles sont conçues pour effectuer une action précise et déterminée.

Séquenceurs câblés

Il existe des processeurs dans lesquels chaque instruction est décodée par un circuit combinatoire ou séquentiel : on parle de **séquenceur câblé**. Plus le nombre d'instructions à câbler est important, plus le nombre de portes utilisées pour fabriquer le séquenceur augmente. Autant dire que les processeurs CISC n'utilisent pas trop ce genre de séquenceurs et préfèrent utiliser des séquenceurs différents. Par contre, les séquenceurs câblés sont souvent utilisés sur les processeurs RISC, qui ont peu d'instructions, pour lesquels la complexité du séquenceur et le nombre de portes est assez faible et supportable.

Sur certains processeurs assez rares, toute instruction s'exécute en une seule micro-opération, chargement depuis la mémoire inclus. Dit autrement, le processeur effectue chaque instruction en un seul cycle d'horloge : un accès mémoire prendra autant de temps qu'une addition, ou qu'une multiplication, etc. Cela demande de caler la durée d'un cycle d'horloge sur l'instruction la plus lente, diminuant fortement les performances. Dans ces conditions, le séquenceur se résume à un simple circuit combinatoire.

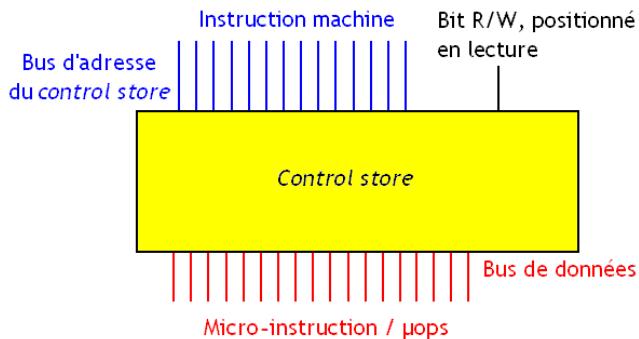
Mais sur la majorité des processeurs, les instructions s'exécutent en plusieurs micro-opérations. Pour enchaîner les micro-opérations, le séquenceur doit savoir à quelle micro-opération il en est, et mémoriser cette information dans une mémoire interne. En conséquence, ces séquenceurs câblés sont des circuits séquentiels. Ces séquenceurs doivent éviter de changer d'instruction à chaque cycle : le processeur doit autoriser ou interdire les modifications du program counter tant qu'il est en train de traiter une instruction.



Séquenceur microcodé

Créer des séquenceurs câblés est très compliqué quand le processeur doit gérer un grand nombre d'instructions machine. Pour limiter la complexité du séquenceur, on peut ruser afin de remplacer le séquenceur par quelque chose de moins complexe. Au lieu de déterminer à l'exécution la suite de micro-opérations à exécuter, on va la pré-calculer dans une mémoire ROM (pour chaque instruction) : cela donne des **séquenceurs microcodés**. Cela a un avantage : remplir une mémoire ROM est beaucoup plus simple à faire que créer un circuit combinatoire.

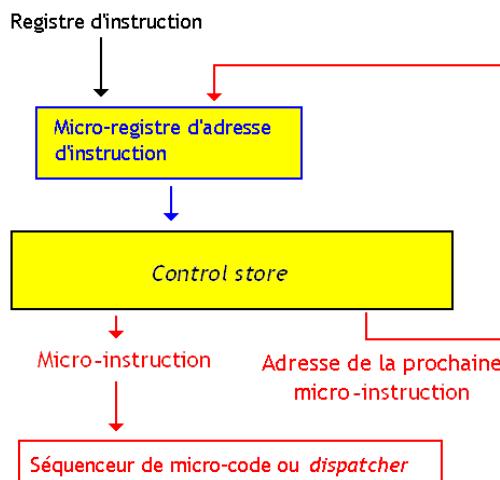
Un séquenceur microcodé contient donc une mémoire ROM : le **control store**. Celui-ci stocke, pour chaque instruction microcodée, la suite de micro-opérations équivalente. Le contenu du control store s'appelle le **microcode**. Parfois, le **control store** est une EEPROM : on peut changer son contenu pour corriger des bugs ou ajouter des instructions. Pour retrouver la suite de micro-opérations correspondante, le séquenceur considère l'opcode de l'instruction microcodée comme une adresse : le **control store** est conçu pour que cette adresse pointe directement sur la suite de micro-opérations correspondante. La micro-opération est parfois recopiée dans un registre, le registre de micro-opération, qui est aux micro-opérations ce que le registre d'instruction est aux instructions machine : il sert à stocker une micro-opération pour que le séquenceur puisse décoder celle-ci.



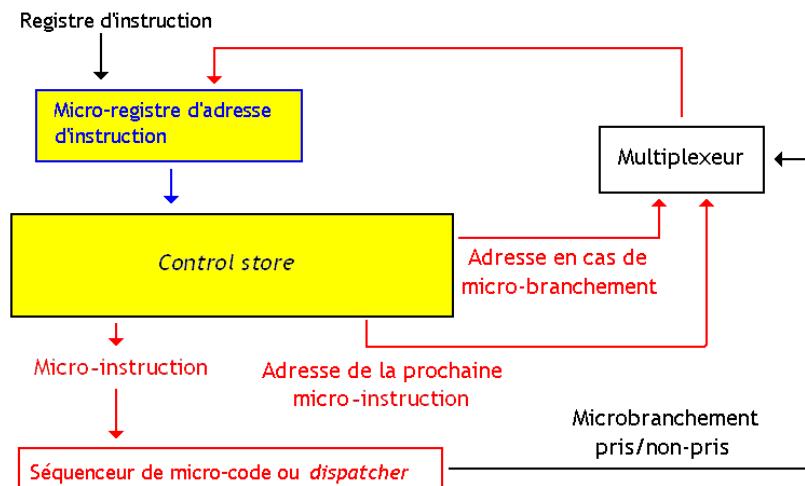
Il existe plusieurs sous-types de séquenceurs microcodés, qui se distinguent par la façon dont sont stockées les micro-opérations. Avec le **microcode horizontal**, chaque instruction du microcode encode directement les signaux de commande à envoyer aux unités de calcul. Avec un **microcode vertical**, il faut traduire les micro-opérations en signaux de commande à l'aide d'un séquenceur câblé. Un séquenceur microcodé utilisant un microcode vertical est divisé en deux parties : un microséquenceur, et une unité de décodage câblée pour décoder les micro-opérations en signaux de commandes. Son avantage est le faible nombre de bits utilisé pour chaque micro-opération : il n'est pas rare que les instructions d'un microcode horizontal fassent plus d'une centaine de bits !

Dans tous les cas, le processeur doit trouver un moyen de dérouler les micro-instructions les unes après les autres, ce qui est la même chose qu'avec des instructions machines. Le micro-code est donc couplé à un circuit qui se charge de l'exécution des micro-opérations les unes après les autres, dans l'ordre demandé. Ce circuit est l'équivalent du circuit de chargement, mais pour les micro-opérations. Un séquenceur microcodé peut même gérer des micro-instructions de branchement, qui précisent la prochaine micro-instruction à exécuter : on peut faire des boucles de micro-opérations, par exemple.

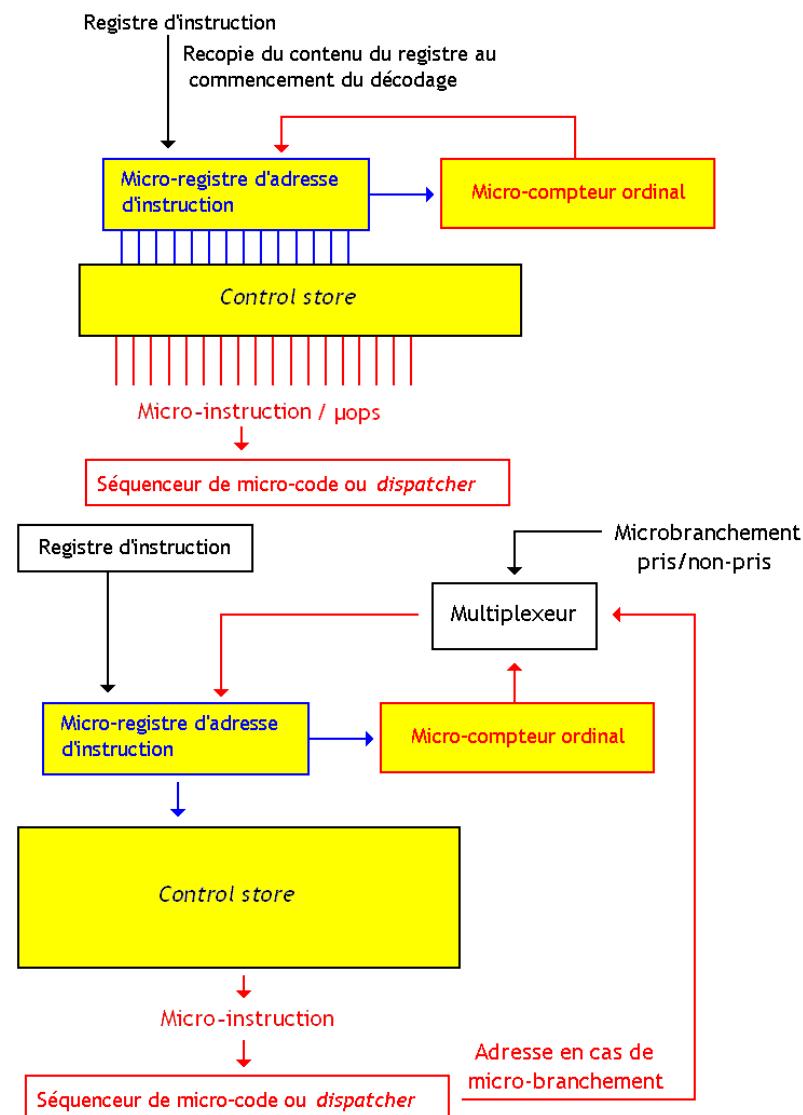
Dans le cas le plus rare, chaque micro-instruction va stocker l'adresse de la micro-instruction suivante.



Pour gérer les micro-branchements, il faut rajouter une seconde adresse : celle de la destination d'un éventuel branchement. Cette adresse est rajoutée pour toutes les instructions, vu que toutes les micro-opérations ont la même taille. Elle n'est simplement pas prise en compte si la micro-opération n'est pas un micro-branchement.



Mais dans la majorité des cas, le séquenceur contient un équivalent du *program counter* pour le microcode : le **registre d'adresse de micro-opération**, couplé à un **microcompteur ordinal**. On peut modifier ce microcompteur ordinal de façon à permettre les branchements entre micro-opérations, d'une manière identique à celle vue pour l'unité de chargement.

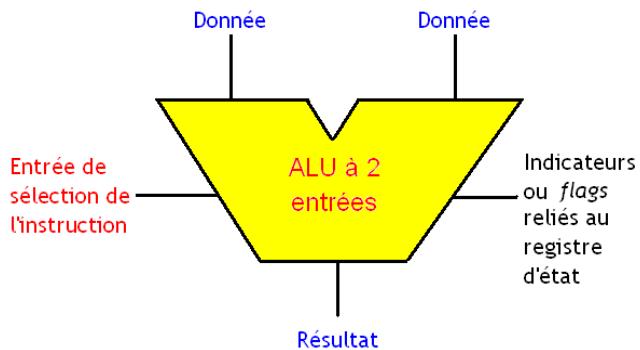


Séquenceurs hybrides

Les séquenceurs hybrides sont un compromis entre séquenceurs câblés et microcodés : une partie des instructions est décodée par une partie câblée, et l'autre par une partie microcodée. Cela évite de câbler une partie du séquenceur qui prendrait beaucoup de portes pour décoder des instructions complexes, généralement rarement utilisées, tout en gardant un décodage rapide pour les instructions simples, souvent utilisées.

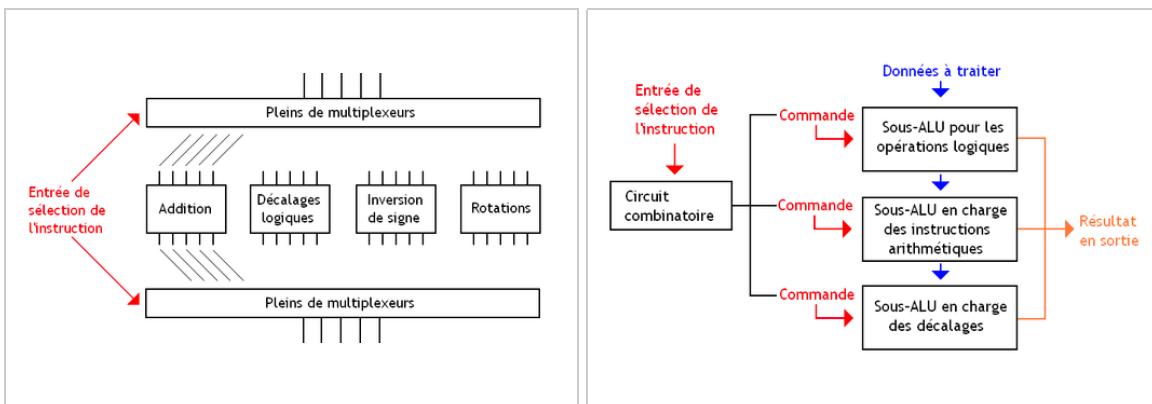
Les unités de calcul

Le processeur contient des circuits capables de faire des calculs arithmétiques, des opérations logiques, et des comparaisons. Ceux-ci sont regroupés dans une unité de calcul, souvent appelée **unité arithmétique et logique**. Certains (dont moi) préfèrent l'appellation anglaise *arithmetic and logic unit*, ou ALU. Les instructions de comparaisons ou de calcul peuvent mettre à jour le registre d'état : le registre d'état est obligatoirement relié à certaines sorties de l'unité de calcul. De plus, il faudra bien spécifier l'instruction à effectuer à notre unité de calcul : il faut bien prévenir notre unité de calcul qu'on veut faire une addition et pas une multiplication. Pour cela, notre unité de calcul possède une entrée permettant de la configurer convenablement : **l'entrée de sélection de l'instruction**. Sur cette entrée, on va mettre un numéro qui précise l'instruction à effectuer. Pour chaque unité de calcul, il existe donc une sorte de correspondance entre ce numéro, et l'instruction à exécuter.



L'intérieur d'une unité de calcul

Ces unités de calcul contiennent souvent un circuit différent pour chaque opération possible. L'entrée de sélection sert uniquement à sélectionner le bon circuit. Pour effectuer cette sélection, certaines unités de calcul utilisent des multiplexeurs commandés par l'entrée de sélection. D'autres envoient les opérandes à tous les circuits en même temps, et activent ou désactivent chaque sous-circuit suivant les besoins. Chaque circuit possède ainsi une entrée de commande, dont la valeur est déduite par un circuit combinatoire à partir de l'entrée de sélection d'instruction de l'ALU (généralement un décodeur).



Unité de calcul conçue avec des sous-ALU reliées par des multiplexeurs.

ALU composée de sous-ALU configurables.

Sur certains processeurs assez anciens, l'ALU est elle-même découpée en plusieurs ALU qui traitent chacune un petit nombre de bits. Par exemple, l'ALU des processeurs AMD Am2900 est une ALU de 16 bits composée de plusieurs sous-ALU de 4 bits. Cette technique qui consiste à créer des unités de calcul plus grosses à partir d'unités de calcul plus élémentaires s'appelle en jargon technique du **bit slicing**.

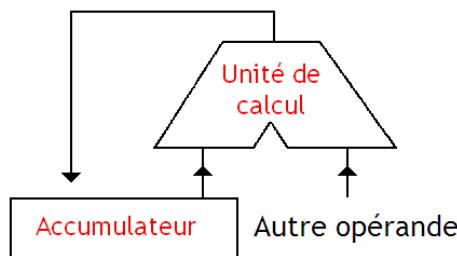
Les unités de calcul spécialisées

Il n'est pas rare qu'un processeur possède plusieurs unités de calcul. Un processeur peut aussi disposer d'unités de calcul spécialisées, séparées de l'unité de calcul principale, pour les décalages, les divisions, etc. Cette séparation est parfois nécessaire : certaines opérations sont assez compliquées à insérer dans une unité de calcul normale, et les garder à part peut simplifier la conception du processeur. Il faut signaler que les processeurs modernes possèdent plusieurs unités de calcul, toutes reliées aux registres. Cela permet d'exécuter plusieurs calculs en même temps dans des unités de calcul différentes. Cela permet d'augmenter les performances du processeur relativement facilement. Diverses technologies, abordées dans la suite du cours permettent de profiter au mieux de ces unités de calcul : pipeline, exécution dans le désordre, exécution superscalaire, jeux d'instructions VLIW, etc.

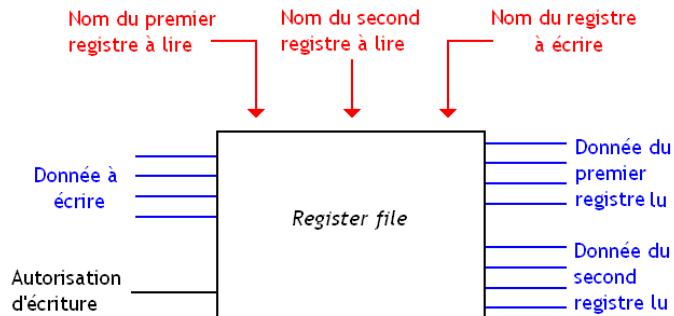
Généralement, les processeurs utilisent une unité de calcul spécialisée pour les nombres flottants : la **floating-point unit**, aussi appelée FPU. Autrefois, ces FPU n'étaient pas incorporés dans le processeur, mais étaient regroupés dans un processeur séparé du processeur principal de la machine. Un emplacement dans la carte mère était réservé à un de ces processeurs spécialisés. On appelait ces processeurs spécialisés dans les calculs flottants des coprocesseurs arithmétiques. Ces coprocesseurs étaient très chers et relativement peu utilisés. Aussi, seules certaines applications assez rares étaient capables d'en tirer profit : des logiciels de conception assistée par ordinateur, par exemple.

Registres

De nombreux registres n'ont pas de nom de registre ou d'adresse et sont sélectionnés implicitement par certaines instructions : *program counter*, registre d'état, etc. Ces registres sont reliés au chemin de données directement. On peut citer le cas du registre accumulateur, présent sur les architectures à accumulateur, qui est un simple registre relié à l'unité de calcul de cette façon.



Les registres qui sont numérotés avec un nom de registre sont rassemblés dans une RAM, dont chaque byte est un registre. Celle-ci porte le nom de **banc de registres** (register file). Le banc de registres est souvent une mémoire multiport, avec au moins un port d'écriture et deux ports de lecture. Cela sert pour les instructions dites dyadiques, qui ont besoin de lire deux données et d'enregistrer leur résultat. On peut avoir plus de ports, pour certaines instructions spéciales. Certains registres non adressables, sans nom, peuvent être placés dans un banc de registres, mais c'est rarement utilisé. Dans ce cas, leur nom de registre est fourni par le séquenceur, et non lu dans l'instruction elle-même.



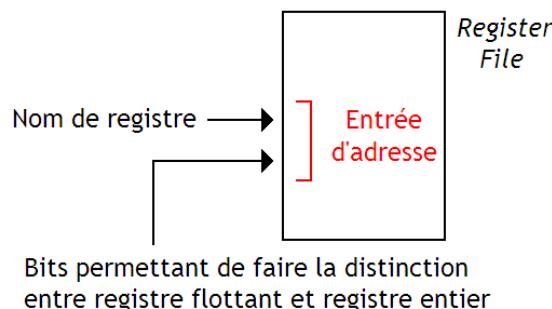
Bancs de registres séparés

Plus un banc de registres a de ports, plus il utilisera de circuits, consommera de courant et chauffera. Les concepteurs de processeurs sont donc obligés de faire des compromis entre le nombre de ports du banc de registres (et donc la performance du processeur), et la chaleur dégagée par le processeur.

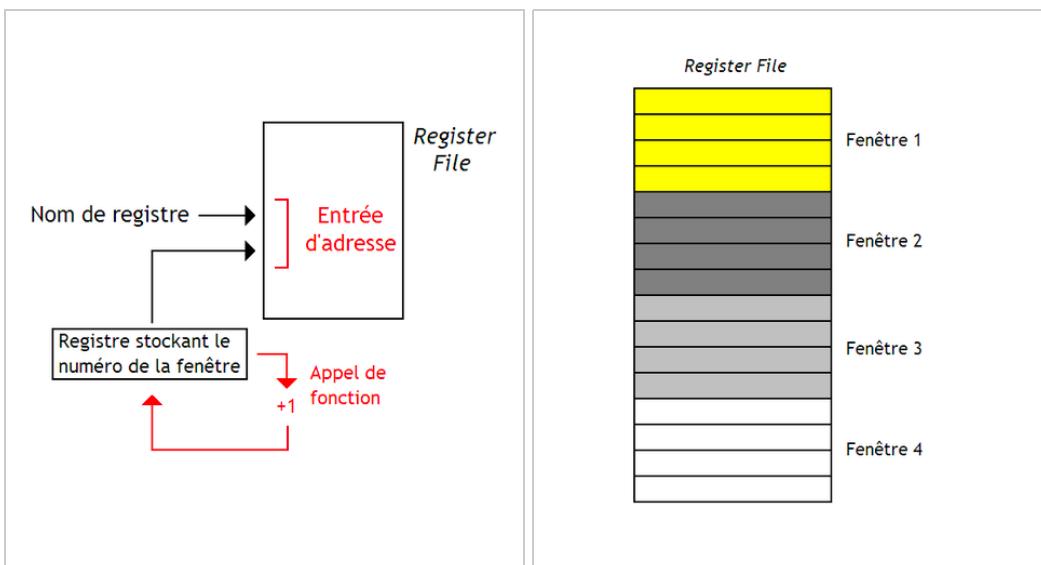
Au-delà d'un certain nombre de registres, il devient difficile d'utiliser un seul gros banc de registres. Il faut alors scinder le banc de registres en plusieurs bancs de registres séparés. C'est assez souvent utilisé sur les architectures dont les registres sont spécialisés, et ne peuvent contenir qu'un type bien défini de donnée (entiers, nombres flottants, adresses) : on trouve alors un banc de registre pour chaque type de registre. Mais rien n'empêche d'utiliser plusieurs bancs de registres sur un processeur qui utilise des registres généraux : il faut juste prévoir de quoi échanger des données entre les bancs de registres. Dans la plupart des cas, cette séparation est invisible du point de vue du langage machine. Sur d'autres processeurs, les transferts de données entre bancs de registres se font via une instruction spéciale, souvent appelée COPY.

Bancs de registres unifiés

Certains processeurs utilisent un seul gros banc de registres dit unifié, même avec des registres spécialisés. Par exemple, c'est le cas des Pentium Pro, Pentium II, Pentium III, ou des Pentium M : ces processeurs ont des registres séparés pour les flottants et les entiers, mais ils sont regroupés dans un seul banc de registres. Avec cette organisation, un registre flottant et un registre entier peuvent avoir le même nom de registre en langage machine, mais l'adresse envoyée au banc de registres ne doit pas être le même : le séquenceur doit ajouter des bits au nom de registre pour former l'adresse finale.



L'utilisation d'un banc de registres unifié permet d'implémenter facilement le fenétrage de registres. Il suffit pour cela de regrouper tous les registres des différentes fenêtres dans un seul banc de registres. Il suffit de faire comme vu au-dessus : rajouter des bits au nom de registre pour faire la différence entre les fenêtres. Cela implique de se souvenir dans quelle fenêtre de registre on est actuellement, cette information étant mémorisée dans un registre qui stocke le numéro de la fenêtre courante. Pour changer de fenêtre, il suffit de modifier le contenu de ce registre lors d'un appel ou retour de fonction avec un petit circuit combinatoire. Bien sûr, il faut aussi prendre en compte le cas où ce registre déborde, ce qui demande d'ajouter des circuits pour gérer la situation.

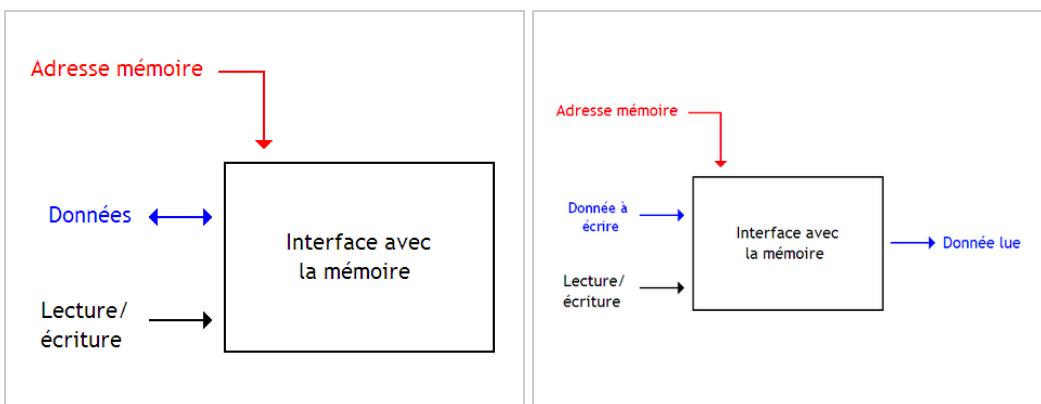


Désambiguisation des fenêtres de registres.

Fenétrage de registres au niveau du banc de registres.

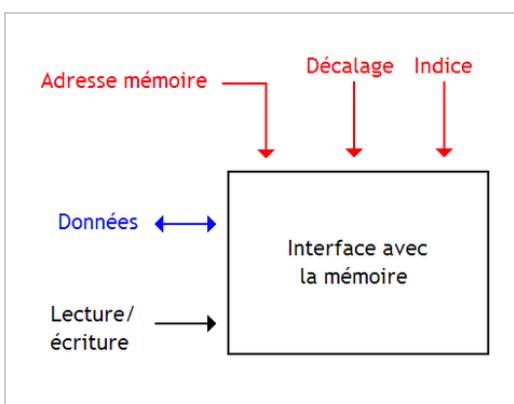
Communication avec la mémoire

L'unité de communication avec la mémoire possède une entrée pour spécifier l'adresse à lire ou écrire, une entrée pour indiquer si l'accès est une lecture ou une écriture, et au moins une entrée/sortie connectée au bus de données. Pour simplifier la conception du processeur, le bus mémoire est parfois relié à des **registres d'interfaçage mémoire**, intercalés entre le bus mémoire et le chemin de données. Au lieu d'aller lire ou écrire directement sur le bus, on va lire ou écrire dans ces registres spécialisés. Généralement, on trouve deux registres d'interfaçage mémoire : un registre relié au bus d'adresse, et autre relié au bus de données. Certaines unités de communication avec la mémoire peuvent gérer certains modes d'adressage elle-mêmes, et faire des calculs d'adresse à partir d'un indice ou d'un décalage. Pour cela, l'unité de communication avec la mémoire contient une unité de calcul interne, l'**Address generation unit**, ou AGU.



Unité de communication avec la mémoire, de type simple port.

Unité de communication avec la mémoire, de type multiport.



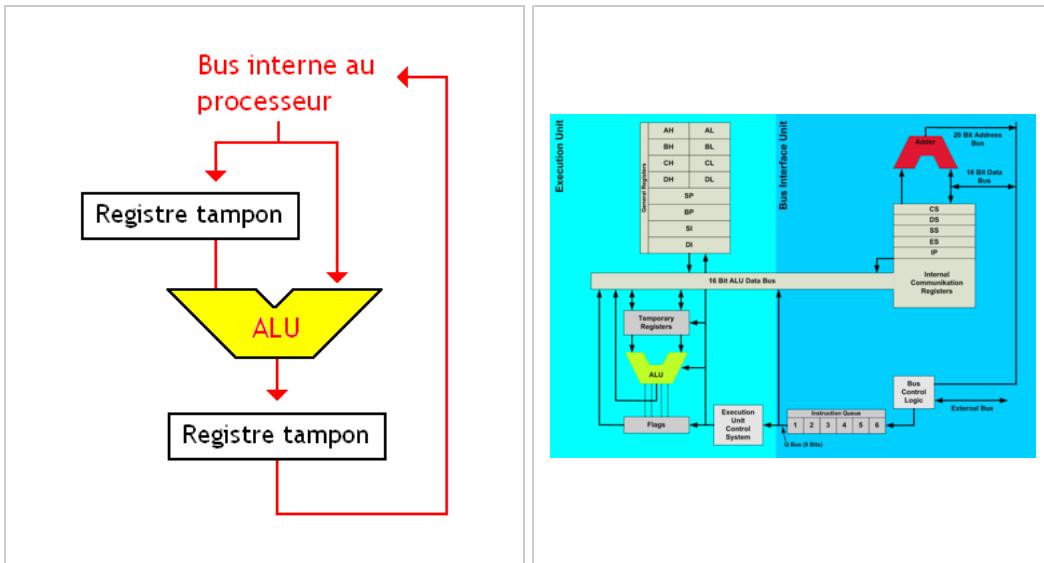
Unité de calcul d'adresse mémoire.

Le bus interne

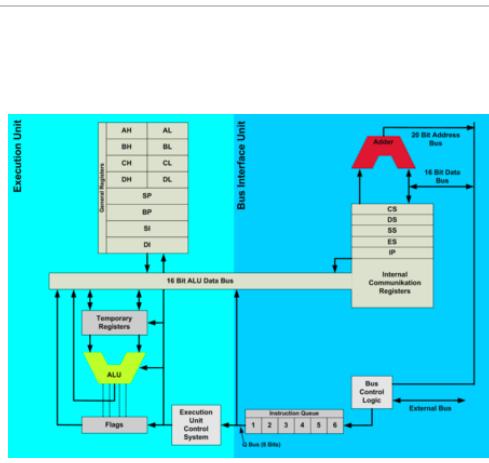
Pour échanger des informations entre les composants du chemin de données, on utilise un ou plusieurs bus internes. Toute micro-instruction

configure ce bus, configuration qui est commandée par le séquenceur. Chaque composant du chemin de données est relié au bus via des transistors, qui servent d'interrupteurs. Pour connecter le banc de registres ou l'unité de calcul à ce bus, il suffit de fermer les bons interrupteurs et d'ouvrir les autres.

Dans le cas le plus simple, le processeur utilise un seul bus interne. Sur ces processeurs, l'exécution d'une instruction dyadique (deux opérandes) peut prendre plusieurs étapes. Si les deux opérandes sont dans les registres, il faut relier l'ALU à ces deux registres, ce qui est impossible avec un seul bus. Pour résoudre ce problème, on doit utiliser un registre spécial pour mettre en attente un opérande pendant qu'on charge l'autre. De même, il est préférable d'utiliser un registre temporaire pour le résultat, pour les mêmes raisons. Le déroulement d'une addition est donc simple : il faut recopier la première opérande dans le registre temporaire, connecter le registre contenant la deuxième opérande sur l'unité de calcul, lancer l'addition, et recopier le résultat du registre temporaire dans le banc de registres. Sur les architectures à accumulateur, ce registre temporaire n'est autre que l'accumulateur lui-même. Les exemples qui vont suivre montrent cependant que d'autres registres temporaires peuvent être ajoutés, pour les autres opérandes.



Chemin de données à un seul bus, principe général.



Exemple d'une architecture sans accumulateur, avec plusieurs registres tampons en amont de l'ALU. Il s'agit de l'architecture de l'Intel 80186.

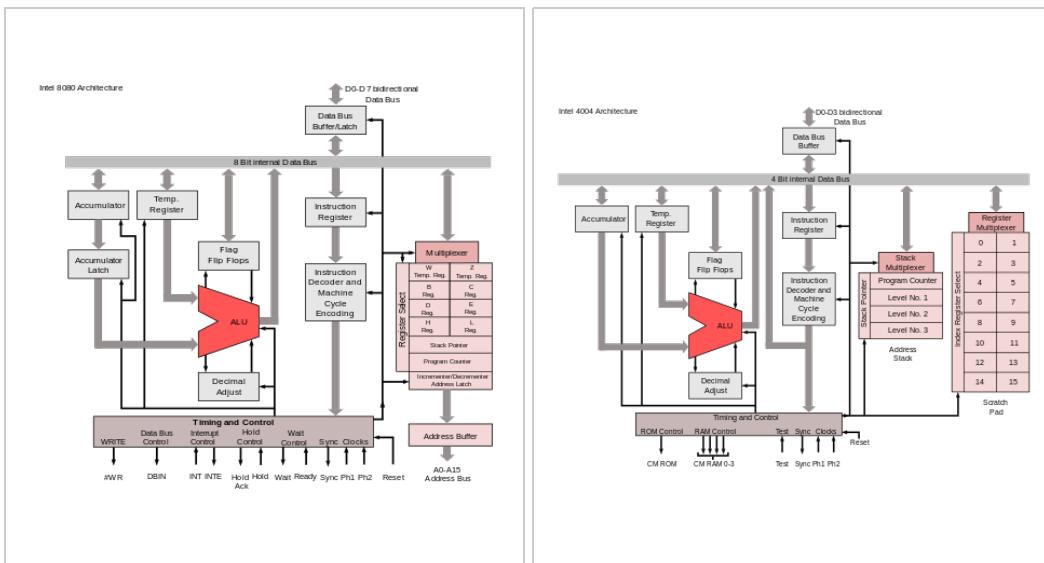
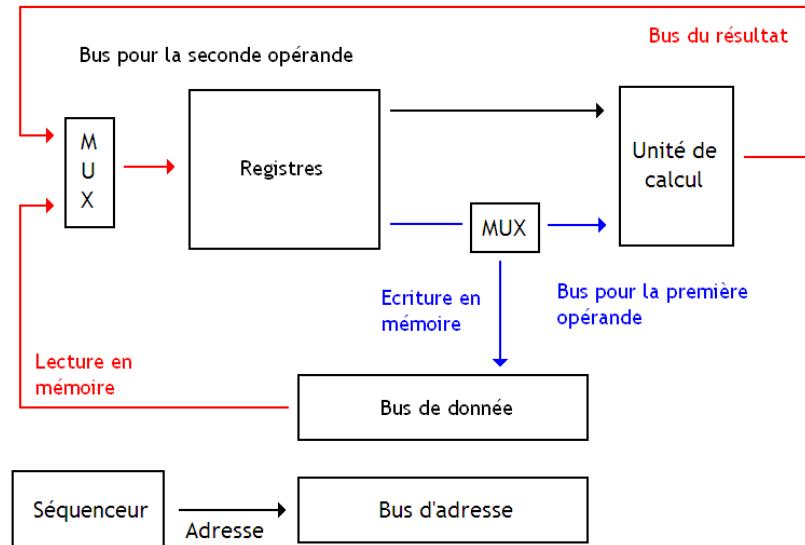


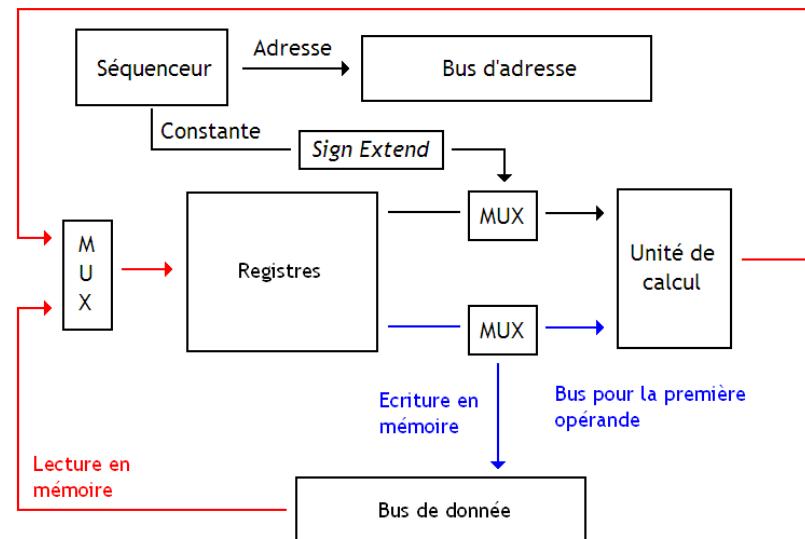
Illustration d'une véritable architecture à accumulateur. L'Intel 8080 possédait une architecture à accumulateur, complétée avec plusieurs registres de données.

Autre illustration avec une autre architecture à accumulateur : l'Intel 4004. On voit que celui-ci contenait aussi une pile.

Certains processeurs s'arrangent pour relier les composants du chemin de données en utilisant plusieurs bus, histoire de simplifier la conception du processeur ou d'améliorer ses performances. Par exemple, on peut utiliser trois bus reliés comme indiqué dans le schéma qui suit. Avec cette organisation, le processeur peut gérer les modes d'adressage absolus, et à registre, pas plus.



Pour gérer l'adressage immédiat, on doit placer la constante inscrite dans l'instruction sur l'entrée de notre ALU. Cette constante est fournie par le séquenceur. Sur certains processeurs, la constante peut être négative et doit alors être convertie en un nombre de même taille que l'entrée de l'ALU. Pour effectuer cette extension de signe, on peut planter un circuit spécialisé.



La gestion du mode d'adressage indirect à registre demande de relier un bus interne du processeur sur le bus d'adresse, ou sur le registre d'interfaçage mémoire adéquat. Si on veut effectuer une écriture, il suffit d'envoyer la donnée à écrire sur le bus de données via un autre bus interne au processeur. Bref, je suppose que vous voyez le principe : on peut toujours adapter l'organisation des bus internes de notre processeur pour gérer de nouveaux modes d'adressage, ou diminuer le nombre de micro-instructions nécessaires pour exécuter une instruction.

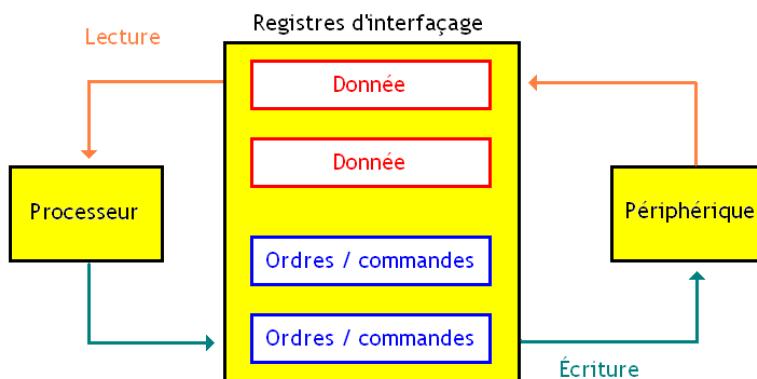
Communication avec les entrées-sorties

Dans ce chapitre, on va voir comment nos périphériques vont faire pour communiquer efficacement avec notre processeur ou notre mémoire. On sait déjà que nos entrées-sorties (et donc nos périphériques) sont reliées au reste de l'ordinateur par un ou plusieurs bus. Pour communiquer avec un périphérique, le processeur a juste besoin de configurer ces bus avec les bonnes valeurs. Mais communiquer avec un périphérique n'est pas aussi simple que ça, comme ce chapitre va vous le montrer.

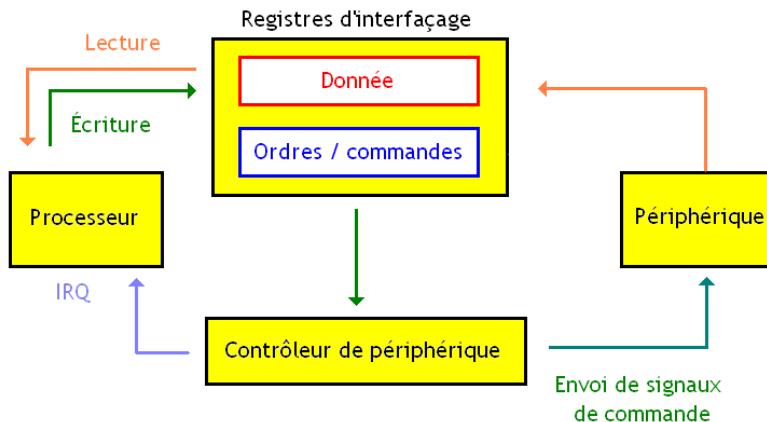
Interfaçage entrées-sorties

Dans la façon la plus simple de procéder, le processeur se connecte au bus et envoie sur le bus l'adresse et les données et commandes à envoyer à l'entrée-sortie ou au périphérique. Ensuite, le processeur va devoir attendre et rester connecté au bus tant que le périphérique n'a pas traité sa demande correctement, que ce soit une lecture, ou une écriture. Mais les périphériques sont tellement lents que le processeur passe son temps à attendre le périphérique.

Pour résoudre ce problème, il suffit d'intercaler des **registres d'interfaçage** entre le processeur et les entrées-sorties. Une fois que le processeur a écrit les informations à transmettre dans ces registres, il peut faire autre chose dans son coin : le registre se charge de maintenir/mémoriser les informations à transmettre. Le processeur doit vérifier ces registres d'interfaçage régulièrement pour voir si un périphérique lui a envoyé quelque chose, mais il peut prendre quelques cycles pour faire son travail en attendant que le périphérique traite sa commande. Ces registres peuvent contenir des données tout ce qu'il y a de plus normales ou des « commandes », des valeurs numériques auxquelles le périphérique répond en effectuant un ensemble d'actions préprogrammées.



Les commandes sont traitées par un **contrôleur de périphérique**, qui va lire les commandes envoyées par le processeur, les interpréter, et piloter le périphérique de façon à faire ce qui est demandé. Le boulot du contrôleur de périphérique est de générer des signaux de commande qui déclenchent une action effectuée par le périphérique. L'analogie avec le séquenceur d'un processeur est possible. Les contrôleurs de périphérique vont du simple circuit de quelques centaines de transistors à un microcontrôleur très puissant. Certains contrôleurs de périphérique peuvent permettre au processeur de communiquer avec plusieurs périphériques en même temps. C'est notamment le cas pour tout ce qui est des contrôleurs PCI, USB et autres. Si le contrôleur de périphérique peut très bien être séparé du périphérique qu'il commande, certains périphériques intègrent en leur sein ce contrôleur : les disques durs IDE, par exemple. Certains de ces contrôleurs intègrent un **registre d'état**, lisible par le processeur, qui contient des informations sur l'état du périphérique. Ils servent à signaler des erreurs de configuration ou des pannes touchant un périphérique.



Lorsqu'un ordinateur utilise un système d'exploitation, celui-ci ne connaît pas toujours le fonctionnement d'un périphérique ou de son contrôleur : il faut installer un programme qui va s'exécuter quand on souhaite communiquer avec le périphérique, et qui s'occupera de tout ce qui est nécessaire pour le transfert des données, l'adressage du périphérique, etc. Ce petit programme est appelé un driver ou **pilote de périphérique**. La « programmation » d'un contrôleur de périphérique est très simple : il suffit de savoir quoi mettre dans les registres pour paramétriser le contrôleur.

Interruptions

La vérification régulière des registres d'interfaçage prend du temps que le processeur pourrait utiliser pour autre chose. Pour réduire à néant ce temps perdu, certains processeurs supportent les **interruptions**. Il s'agit de fonctionnalités du processeur, qui interrompent temporairement l'exécution d'un programme pour réagir à un événement extérieur (matériel, erreur fatale d'exécution d'un programme...). L'interruption va effectuer un petit traitement (ici, communiquer avec un périphérique), réalisé par un programme nommé **routine d'interruption**. Avec ces interruptions, le processeur n'a pas à vérifier périodiquement si le contrôleur de périphérique a fini son travail : il suffit que le contrôleur de périphérique prévienne le processeur avec une interruption. Lorsqu'un processeur exécute une interruption, celui-ci :

- arrête l'exécution du programme en cours et sauvegarde l'état du processeur (registres et program counter) ;
- exécute la routine d'interruption ;

- restaure l'état du programme sauvegardé afin de reprendre l'exécution de son programme là où il en était.

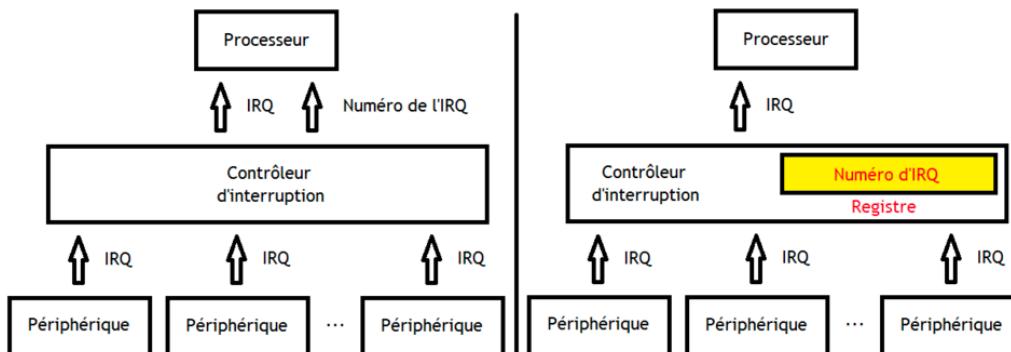
L'appel d'une routine d'interruption est similaire à un appel de fonction, ce qui fait qu'il faut sauvegarder les registres du processeur, vu que la fonction ou la routine peuvent écraser des données dans les registres. Cette sauvegarde n'est pas toujours faite automatiquement par notre processeur : c'est parfois le programmeur qui doit coder lui-même la sauvegarde de ces registres dans la routine d'interruption elle-même. Certains processeurs fournissent des registres spécialement dédiés aux interruptions, qui ne sont accessibles que par les routines d'interruptions : cela évite d'avoir à sauvegarder les registres généraux. D'autres utilisent le fenêtrage de registres, avec une fenêtre pour les interruption et une autre pour les programmes. Pour savoir s'il est dans une interruption, le processeur utilise une bascule.

Devant la multiplicité des périphériques, on se doute bien qu'il n'existe pas d'interruption à tout faire : il va de soi qu'un programme envoyant un ordre au disque dur sera différent d'un programme agissant sur une carte graphique. On a donc besoin de plusieurs routines d'interruption : au moins une par périphérique. Certains ordinateurs utilisent une partie de leur mémoire pour stocker les adresses de chaque routine : cette portion de mémoire s'appelle le vecteur d'interruption. Lorsqu'une interruption a lieu, le processeur va automatiquement aller chercher son adresse dans ce vecteur d'interruption. Une autre solution est simplement de déléguer cette gestion du choix de l'interruption au système d'exploitation : l'OS devra alors traiter l'interruption tout seul. Dans ce cas, le processeur contient un registre qui stockera des bits qui permettront à l'OS de déterminer la cause de l'interruption : est-ce le disque dur qui fait des siennes, une erreur de calcul dans l'ALU, une touche appuyée sur le clavier, etc.

Quand deux interruptions de déclenchent en même temps, on ne peut exécuter qu'une seule interruption. Le truc, c'est que certaines interruptions seront prioritaires sur les autres : par exemple, l'interruption qui gère l'horloge système est plus prioritaire qu'une interruption en provenance de périphériques lents comme le disque dur ou une clé USB. Quand deux interruptions souhaitent s'exécuter en même temps, on choisit d'exécuter celle qui est la plus prioritaire. L'autre interruption n'est pas exécutée, et est mise en attente : on parle de masquage d'interruption. Le masquage d'interruption permet de bloquer des interruptions temporairement, et de les exécuter ultérieurement, une fois le masquage d'interruption levé. Il faut noter qu'il est parfois possible de masquer à l'avance certaines interruptions : on peut ainsi désactiver temporairement l'exécution des interruptions, quelle qu'en soit la raison. Certaines interruptions ne sont pas masquables. Il s'agit généralement d'erreurs matérielles importantes, comme une erreur de protection mémoire, une division par zéro au niveau du processeur, une surchauffe du processeur, etc. Ces interruptions sont systématiquement exécutées en priorité, et ne sont jamais masquées.

Il y a trois méthodes pour déclencher une interruption :

- Les **interruptions logicielles** sont déclenchées par un programme en cours d'exécution, via une instruction d'interruption. Un programmeur peut donc décider d'utiliser des interruptions à un certain moment de ce programme, pour des raisons particulières. Ces interruptions logicielles sont surtout utilisées par les pilotes de périphériques ou les systèmes d'exploitation.
- Une **exception matérielle** est aussi une interruption, mais qui a pour raison un événement interne au processeur, par exemple une erreur d'adressage, une division par zéro... Pour pouvoir exécuter des exceptions matérielles, notre processeur doit pouvoir déclencher une interruption lorsqu'une erreur particulière survient dans le traitement d'une instruction. Il faut donc que ce processeur intègre des circuits dédiés à cette tâche. Lorsqu'une exception matérielle survient, il faut trouver un moyen de corriger l'erreur qui a été la cause de l'exception matérielle : la routine exécutée va donc servir à corriger celle-ci. Bien sûr, une exception matérielle peut avoir plusieurs causes. On a donc plusieurs routines.
- Les **IRQ** sont des interruptions déclenchées par un périphérique. Dans une implémentation simple des IRQ, chaque périphérique envoie ses interruptions au processeur via une entrée IRQ : la mise à 1 de cette entrée déclenche une interruption bien précise au cycle suivant. Pour économiser des entrées, on a inventé le contrôleur d'interruptions, un circuit sur lequel on connecte tous les fils d'IRQ. Ce contrôleur va gérer les priorités et les masques. Ce contrôleur envoie un signal d'interruption IRQ global au processeur et un numéro qui précise quel périphérique a envoyé l'interruption, qui permet de savoir quelle routine exécuter. Parfois, ce numéro n'est pas envoyé au processeur directement, mais stocké dans un registre, accessible via le bus de données.



Direct memory access

Avec nos interruptions, seul le processeur gère l'adressingage de la mémoire. Impossible par exemple, de permettre à un périphérique d'adresser la mémoire RAM ou un autre périphérique. Il doit donc forcément passer par le processeur, et le monopoliser durant un temps assez long, au lieu de laisser notre processeur exécuter son programme tranquille. Pour éviter cela, on a inventé le bus mastering. Grâce au bus mastering, le périphérique adresse la mémoire directement. Il est capable d'écrire ou lire des données directement sur les différents bus. Ainsi, un périphérique peut accéder à la mémoire, ou communiquer avec d'autres périphériques directement, sans passer par le processeur. Le **direct memory access** est une technologie de bus mastering qui permet aux périphériques d'accéder à la RAM sans passer par le processeur. Elle peut même servir à transférer des données de la mémoire vers la mémoire, pour effectuer des copies de très grosses données.

Avec la technologie DMA, l'échange de données entre le périphérique et la mémoire est intégralement géré par un circuit spécial, intégré au périphérique et relié au bus mémoire : le **contrôleur DMA**. Ce contrôleur DMA est capable de transférer un gros bloc de mémoire entre un périphérique et la mémoire. Il contient des registres dans lesquels le processeur pour initialiser un transfert de données. Ces registres contiennent, au minimum un registre pour l'adresse du segment de la mémoire, un autre pour la longueur du segment de mémoire. Le travail du contrôleur est assez simple. Celui-ci doit se contenter de placer les bonnes valeurs sur les bus, pour effectuer le transfert. Il va donc initialiser le bus d'adresse à l'adresse du début du bloc de mémoire. Puis, à chaque fois qu'une donnée est lue ou écrite sur le périphérique, il va augmenter l'adresse de ce qu'il faut pour sélectionner le bloc de mémoire suivant. Le transfert peut aller dans les deux sens : du périphérique vers la RAM, ou de la RAM vers le périphérique. Le sens du transfert, ainsi que les informations sur le bloc de mémoire à transférer, sont précisés dans des registres interne au contrôleur DMA. On trouve aussi parfois un ou plusieurs registres de contrôle. Ces registres de contrôle peuvent contenir beaucoup de choses : avec quel périphérique doit-on échanger des données, les données sont-elles copiées du périphérique vers la RAM ou l'inverse, et bien d'autres choses encore. Lorsqu'un périphérique souhaite accéder à la mémoire ou qu'un programme veut envoyer des données à un périphérique, il déclenche l'exécution d'une interruption et configure le contrôleur DMA avec les données nécessaires pour démarrer le transfert de donnée.

Il existe trois façons de transférer des données entre le périphérique et la mémoire : le mode block, le mode cycle stealing, et le mode transparent.

- Dans le **mode block**, le contrôleur mémoire se réserve le bus mémoire, et effectue le transfert en une seule fois, sans interruption. Cela a un désavantage : le processeur ne peut pas accéder à la mémoire durant toute la durée du transfert entre le périphérique et la mémoire. Alors certes, ça va plus vite que si on devait utiliser le processeur comme intermédiaire, mais bloquer ainsi le processeur durant le transfert peut diminuer les performances. Dans ce mode, la durée du transfert est la plus faible possible. Il est très utilisé pour charger un programme du

- disque dur dans la mémoire, par exemple. Eh oui, quand vous démarrez un programme, c'est souvent un contrôleur DMA qui s'en charge !
- Dans le **mode cycle stealing**, on est un peu moins strict : cette fois-ci, le contrôleur ne bloque pas le processeur durant toute la durée du transfert. En cycle stealing, le contrôleur va simplement transférer un mot mémoire (un octet) à la fois, avant de rendre la main au processeur. Puis, le contrôleur récupérera l'accès au bus après un certain temps. En gros, le contrôleur transfère un mot mémoire, fait une pause d'une durée fixe, puis recommence, et ainsi de suite jusqu'à la fin du transfert.
 - Et enfin, on trouve le **mode transparent**, dans lequel le contrôleur DMA accède au bus mémoire uniquement quand le processeur ne l'utilise pas.

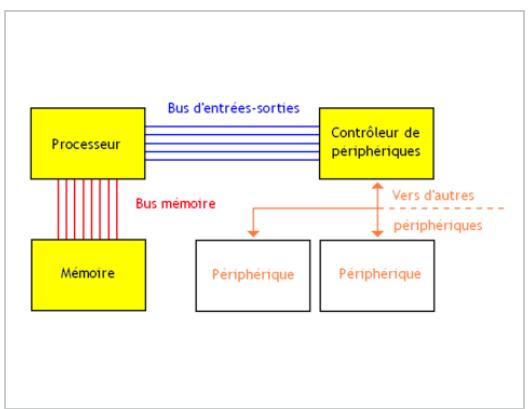
Adressage des périphériques

Pour accéder aux registres du contrôleur de périphérique, il existe trois méthodes :

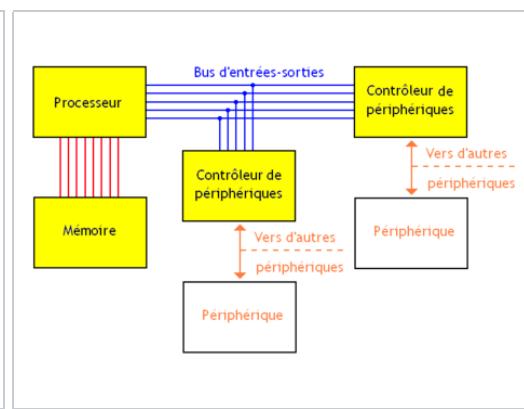
- la connexion directe ;
- l'espace d'adressage séparé ;
- les entrées-sorties mappées en mémoire.

Connexion directe

Dans le cas le plus simple, le contrôleur est directement relié au processeur par un bus d'entrées-sorties. Si le contrôleur est unique, il n'a pas d'adresse qui permettrait de l'identifier : le bus d'entrées-sorties se réduit donc à un bus de données couplé à un bus de commande. Les problèmes commencent quand il faut câbler plusieurs contrôleurs de périphérique. Pour éviter tout problème, les contrôleurs de périphériques se partagent le même bus d'entrées-sorties. Avec cette solution, chaque contrôleur de périphérique se voit attribuer une adresse, utilisée pour l'identifier et le sélectionner. Cette adresse a deux buts : adresser le contrôleur de périphérique, et préciser dans quel registre du contrôleur il faut aller lire ou écrire. D'ordinaire, certains bits de l'adresse indiquent quel contrôleur de périphérique est le destinataire, les autres indiquant le registre de destination. Cette adresse peut être fixée une bonne fois pour toute dès la conception du périphérique, ou se configurer via un registre ou une EEPROM.



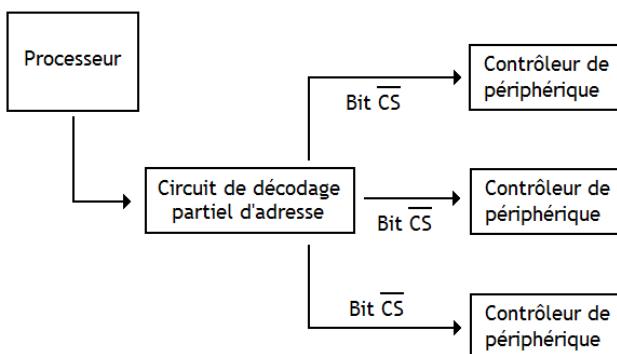
Bus entre processeur et contrôleur de périphérique.



Bus d'entrées-sorties multiplexé.

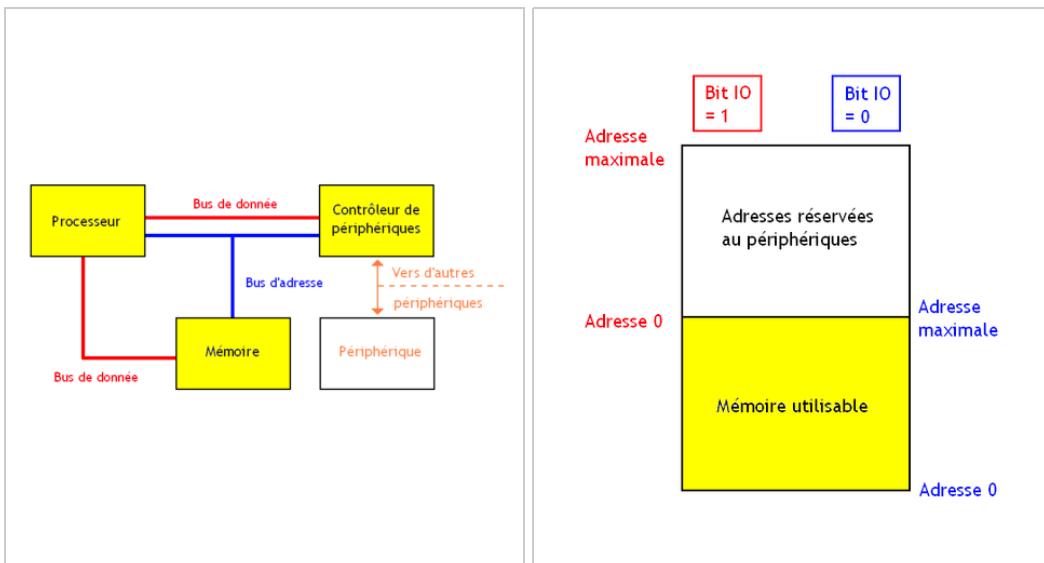
L'adressage des périphériques peut se faire de deux manières différentes. Première solution : chaque composant branché sur le bus vérifie si l'adresse envoyée par le processeur est bien la sienne : si c'est le cas, il va se connecter sur le bus (les autres composants restants déconnectés). En conséquence, chaque contrôleur contient un comparateur pour cette vérification d'adresse, dont la sortie commande les circuits trois états qui relient le contrôleur au bus.

Seconde solution : utiliser un circuit qui détermine, à partir de l'adresse, quel est le composant adressé. Seul ce composant sera aconnecter activé/connecté au bus, tandis que les autres seront désactivés/déconnectés du bus. Pour cela, chaque contrôleur possède une entrée CS, qui désactive le contrôleur de périphérique : si ce bit est à zéro, le contrôleur de périphérique ne prend absolument pas en compte ce qu'on trouve sur ses entrées, et ses sorties sont déconnectées. Ce bit est identique au bit CS : des mémoires RAM et ROM. Pour éviter les conflits sur le bus, un seul contrôleur de périphérique doit avoir son bit CS à 1. Pour cela, il faut ajouter un circuit qui prend en entrée l'adresse et qui commande les bits CS : ce circuit est un circuit de décodage partiel d'adresse.



Espace d'adressage séparé

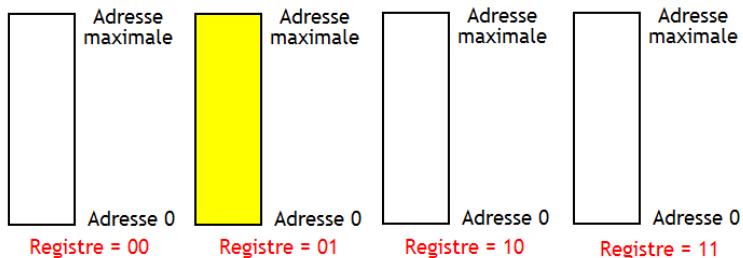
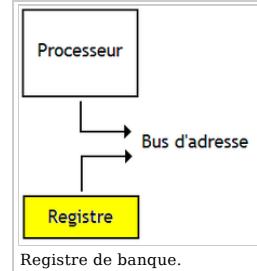
Pour économiser des fils et des broches sur le processeur, il est possible de mutualiser le bus d'adresses entre le bus d'entrées-sorties et le bus mémoire. En faisant cela, un mot mémoire et un registre du contrôleur de périphérique peuvent avoir la même adresse. On doit donc indiquer la destination de l'adresse, périphérique ou mémoire, via un **bit IO** sur le bus d'adresse. Pour faire la sélection entre périphérique et mémoire, le bit IO est envoyé au circuit de décodage partiel d'adresse, qui commande les bits CS des périphériques et de la mémoire. Si le bit IO vaut zéro, le bit CS de la mémoire est mis à zéro, déconnectant celle-ci du bus. Pour positionner le bit IO à la bonne valeur, le processeur utilise des instructions différentes pour communiquer avec le contrôleur de périphérique et la mémoire. Suivant l'instruction utilisée, le bit IO sera positionné à la bonne valeur.



Espace d'adressage séparé.

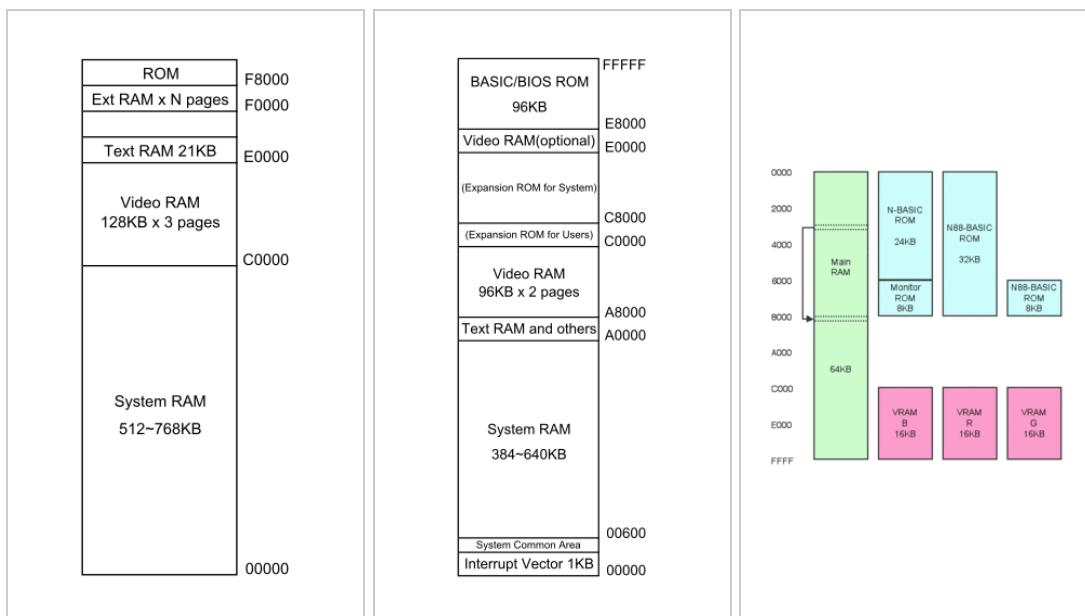
Bit IO.

Certains processeurs améliorent ce principe, en utilisant plusieurs bits IO : cela donne la technique du bank switching. Cette technique consiste à utiliser un bus d'adresse réel plus grand que celui du processeur, les bits manquants étant reliés à un registre configurable par le processeur : le **registre de banque**. L'espace mémoire du processeur est présent en plusieurs exemplaires, sélectionnés non pas par un bit IO, mais par la valeur du registre de banque. On peut changer de banque en changeant le contenu de ce registre : le processeur dispose souvent d'instructions spécialisées qui en sont capables. Chaque exemplaire de l'ensemble des adresses du processeur s'appelle une banque. Le processeur sera limité par son bus d'adresse pour adresser les données dans une banque, pas pour l'ensemble de la mémoire. Cette technique s'appelle la **commutation de banque**. En répartissant les données utiles dans différentes banques, le processeur peut donc adresser beaucoup plus de mémoire. Par exemple, supposons que j'ai besoin d'adresser une mémoire ROM de 4 kibioctets, une RAM de 8 kibioctets, et divers périphériques. Le processeur a un bus d'adresse de 12 bits, ce qui limite à 4 kibioctets. Dans ce cas, je peux réserver 4 banques : une pour la ROM, une pour les périphériques, et deux banques qui contiennent chacune la moitié de la RAM.



Entrées-sorties mappées en mémoire

Partager le bus d'adresse complexifie pas mal la conception d'un processeur, vu qu'il faut ajouter des instructions spécialisées. Pour éviter ce genre de désagrément, on a trouvé une autre solution : mapper les entrées-sorties en mémoire. Avec cette technique, périphériques et mémoire sont connectées au processeur par le même bus, qui est intégralement mutualisé. Certaines adresses mémoire sont redirigées automatiquement vers les périphériques, ce qui fait que le périphérique se retrouve inclus dans l'ensemble des adresses utilisées pour manipuler la mémoire. Cette technique peut aussi s'appliquer pour les mémoires rom, ou dans les systèmes qui utilisent plusieurs mémoires.

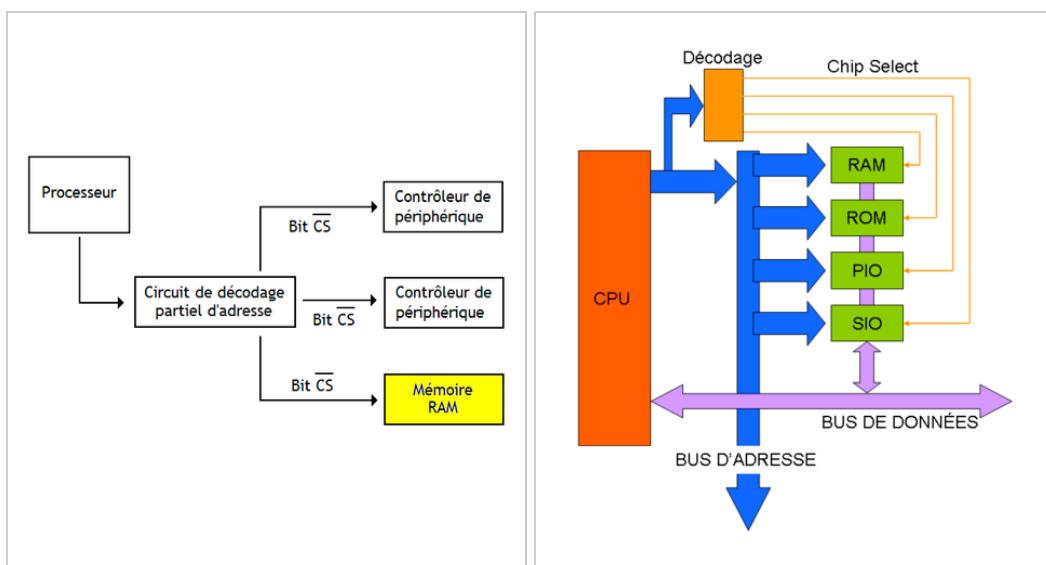


Adressage mémoire (carte mématoire) du N5200mk2.

Adressage mémoire (carte mématoire) du PC-9801VM.

Adressage mémoire (carte mématoire) du PC-8801. On voit que cette technique peut se combiner avec la commutation de banques. Chaque banque utilise des RAM/ROM mappées.

Certains mots mémoire renverront donc vers un périphérique, cette redirection s'effectuant par décodage partiel d'adresse : le circuit de décodage partiel d'adresse va ainsi placer le bit CS de la mémoire à 1 pour les adresses invalidées, l'empêchant de répondre à ces adresses. Évidemment, les adresses utilisées pour les périphériques ne sont plus disponibles pour la mémoire RAM. C'est ce qui fait que certaines personnes installent 4 gigaoctets de mémoire sur leur ordinateur et se retrouvent avec « seulement » 3,5 à 3,8 gigaoctets de mémoire, les périphériques prenant le reste. Ce « bug » apparaît sur les processeurs x86 quand on utilise un système d'exploitation 32 bits. On remarque ainsi le défaut inhérent à cette technique : on ne peut plus adresser autant de mémoire qu'avant. Et mine de rien, quand on a une carte graphique avec 512 mégaoctets de mémoire intégrée, une carte son, une carte réseau PCI, des ports USB, un port parallèle, un port série, des bus PCI Express ou AGP, et un BIOS à stocker dans une EEPROM/Flash, ça part assez vite.



Décodage d'adresse avec entrées-sorties mappées en mémoire. Exemple détaillé.

Bus et cartes-mères

Dans un ordinateur, les composants sont placés sur un circuit imprimé (la carte mère), un circuit sur lequel on vient connecter les différents composants d'un ordinateur, et qui relie ceux-ci via divers bus. Mais ces composants ne communiquent pas que par un seul bus : il existe un bus pour communiquer avec le disque dur, un bus pour la carte graphique, un pour le processeur, un pour la mémoire, etc. De ce fait, un PC contient un nombre impressionnant de bus, jugez plutôt :

- le SMBUS, qui est utilisé pour communiquer avec les ventilateurs, les sondes de température et les sondes de tension présentes un peu partout dans notre ordinateur ;
- les bus USB ;
- le bus PCI, utilisé pour les cartes son et qui servait autrefois à communiquer avec les cartes graphiques ;
- le bus AGP, autrefois utilisé pour les cartes graphiques ;
- le bus PCI Express, utilisé pour communiquer avec des cartes graphiques ou des cartes son ;
- le bus P-ATA, relié au disque dur ;
- le bus S-ATA et ses variantes eSATA, eSATAp, ATAoE, utilisés pour communiquer avec le disque dur ;
- le bus Low Pin Count, qui permet d'accéder au clavier, aux souris, au lecteur de disquette, et aux ports parallèle et série ;
- le bus ISA et son cousin le bus EISA, autrefois utilisés pour des cartes d'extension ;
- l'Intel QuickPath Interconnect et l'HyperTransport, qui relient les processeurs récents au reste de l'ordinateur ;
- le FireWire (1394) ;
- le bus SCSI et ses variantes (SCSI Parallel, Serial Attached SCSI, iSCSI), qui permettent de communiquer avec des disques durs ;
- le bus MIDI, une véritable antiquité oubliée de tous, qui servait pour les cartes son ;
- le fameux RS-232 utilisé dans les ports série ;
- enfin, le bus IEEE-1284 utilisé pour le port parallèle.

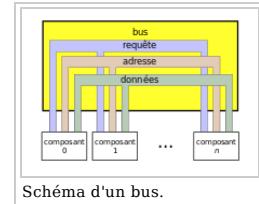


Schéma d'un bus.

Propriétés d'un bus

Ces bus sont très différents les uns des autres, et ont des caractéristiques très différentes qui sont listées dans le tableau ci-dessous.

Caractéristique	Définition
Largeur	Nombre maximal de bits transmis simultanément sur le bus.
Débit binaire	Nombre de bits que le bus peut transmettre par seconde.
Latence	Temps d'attente que met une donnée à être transférée sur le bus.
Caractère synchrone ou asynchrone	Certains bus transmettent un signal d'horloge, d'autres non.
Protocole	Le protocole d'un bus définit comment les données sont envoyées sur le bus.

Bus série et parallèle

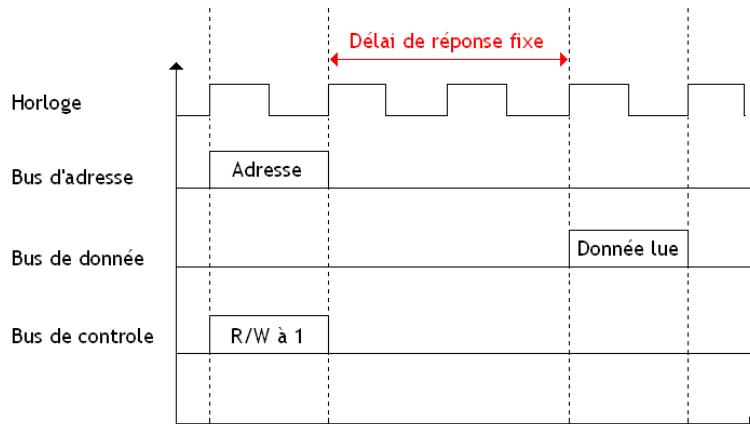
La plupart des bus peuvent échanger plusieurs bits en même temps et sont appelés **bus parallèles**. Mais il existe des bus qui ne peuvent échanger qu'un bit à la fois : ce sont des **bus série**. On pourrait croire qu'un bus série ne contient qu'un seul fil pour transmettre les données, mais il existe des contremes. Généralement, c'est le signe que le bus n'utilise pas un codage NRZ, mais une autre forme de codage un peu plus complexe. Par exemple, le bus USB utilise deux fils D+ et D- pour transmettre un bit. Pour faire simple, lorsque le fil D+ est à sa tension maximale, l'autre est à zéro (et réciproquement).

Passons maintenant aux bus parallèles. Pour information, si le contenu d'un bus de largeur de n bits est mis à jour m fois par secondes, alors son débit binaire est de $n \times m$. Mais contrairement à ce qu'on pourrait croire, les bus parallèles ne sont pas plus rapides que les bus série. En effet, le temps d'attente entre deux transmissions est plus grand sur les bus parallèles. Un bus série n'a pas ce genre de problèmes, ce qui surcompense le fait qu'un bus série ne peut envoyer qu'un bit à la fois.

Il existe plusieurs raisons à cette infériorité des temps de latence des bus série. Premièrement, les fils d'un bus ne sont pas identiques électriquement : leur longueur et leur résistance changent très légèrement d'un fil à l'autre. En conséquence, un bit va se propager à des vitesses différentes suivant le fil. On est obligé de se caler sur le fil le plus lent pour éviter des problèmes à la réception. En second lieu, il y a le phénomène de crosstalk. Lorsque la tension à l'intérieur du fil varie (quand le fil passe de 0 à 1 ou inversement), le fil va émettre des ondes électromagnétiques qui perturbent les fils d'à côté. Il faut attendre que la perturbation électromagnétique se soit atténuée pour lire les bits, ce qui limite le nombre de changements d'état du bus par seconde.

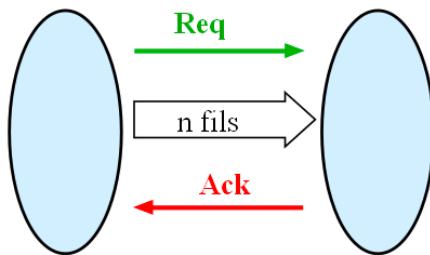
Bus synchrones et asynchrones

Certains bus sont synchronisés sur un signal d'horloge : ce sont les **bus synchrones**. Avec ces bus, le temps de transmission d'une donnée est fixé une fois pour toute. Le composant sait combien de cycles d'horloge durent une lecture ou une écriture. Sur certains bus, le contenu du bus n'est pas mis à jour à chaque front montant, ou à chaque front descendant, mais aux deux : fronts montant et descendant. De tels bus sont appelés des bus double data rate. Cela permet de transférer deux données sur le bus (une à chaque front) en un seul cycle d'horloge : le débit binaire est doublé sans toucher à la fréquence du bus.

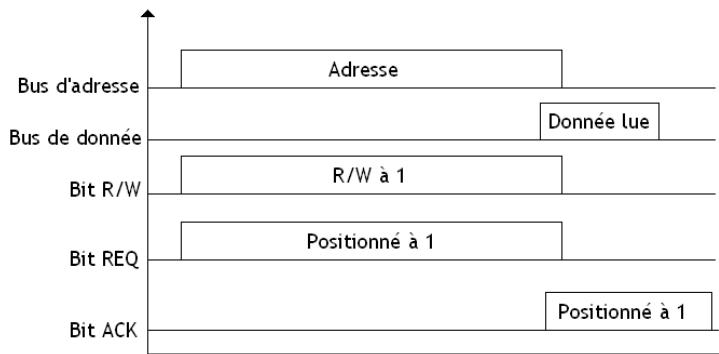


À haute fréquence, le signal d'horloge met un certain temps pour se propager à travers le fil d'horloge, ce qui induit un léger décalage entre les composants. Plus on augmente la longueur des fils, plus ces décalages deviendront ennuyeux. Plus on augmente la fréquence, plus la période

diminue comparée au temps de propagation de l'horloge dans le fil. Ces phénomènes font qu'il est difficile d'atteindre des fréquences de plusieurs gigahertz sur les bus actuels. Pour ces raisons, certains bus se passent complètement de signal d'horloge, et ont un protocole conçu pour : ce sont les **bus asynchrones**. Ces bus sont donc très adaptés pour transmettre des informations sur de longues distances (plusieurs centimètres ou plus). Pour cela, ces bus permettent à deux composants de se synchroniser grâce à des fils spécialisés du bus de commande, qui transmettent des bits particuliers. Généralement, ce protocole utilise deux fils supplémentaires : REQ et ACK.



Lorsqu'un composant veut envoyer une information sur le bus à un autre composant, celui-ci place le fil REQ à 1, afin de dire au récepteur : « attention, j'ai besoin que tu me fasses quelque chose ». Les autres composants vont alors réagir et lire le contenu du bus. Le composant à qui la donnée ou l'ordre est destiné va alors réagir et va faire ce qu'on lui a demandé (les autres composants se rendorment et se déconnectent du bus). Une fois qu'il a terminé, celui-ci va alors positionner le fil ACK à 1 histoire de dire : j'ai terminé, je libère le bus !



Bus dédiés et multiplexés

Sur certains bus, on peut connecter un nombre assez important de composants : ce sont les **bus multiplexés**. Ces bus sont à opposer aux **bus dédiés**, qui se contentent de connecter deux composants entre eux. Le composant qui envoie une donnée sur le bus est appelé un émetteur. Celui qui se contente de recevoir une donnée sur le bus est appelé récepteur.

Les bus dédiés sont classés en trois types, suivant les récepteurs et les émetteurs.

Simplex	Half-duplex	Full-duplex
Les informations ne vont que dans un sens : un composant est l'émetteur et l'autre reste à tout jamais récepteur.	Il est possible d'être émetteur ou récepteur, suivant la situation. Par contre, impossible d'être à la fois émetteur et récepteur.	Ce bus permet d'être à la fois récepteur et émetteur. On peut créer un bus full duplex en regroupant deux bus simplex ensemble, un bus pour l'émission et un pour la réception. Mais certains bus n'utilisent pas cette technique et se contentent d'un seul bus bidirectionnel.

Sur les bus multiplexés, il arrive que plusieurs composants tentent d'envoyer ou de recevoir une donnée sur le bus en même temps : c'est un **conflit d'accès au bus**. Cela pose problème si un composant cherche à envoyer un 1 et l'autre un 0 : tout ce que l'on reçoit à l'autre bout du fil est un espèce de mélange incohérent des deux données envoyées sur le bus par les deux composants. Pour résoudre ce problème, il faut répartir l'accès au bus pour n'avoir qu'un émetteur à la fois. On doit choisir un émetteur parmi les candidats. Ce choix sera effectué différemment suivant le protocole du bus et son organisation, mais ce choix n'est pas gratuit : certains composants devront attendre leur tour pour avoir accès au bus.

Les concepteurs de bus ont inventé des méthodes pour gérer ces conflits d'accès, et choisir le plus efficacement possible l'émetteur : on parle d'**arbitrage du bus**. Dans l'**arbitrage centralisé**, un circuit spécialisé s'occupe de l'arbitrage du bus. Dans l'**arbitrage distribué**, chaque composant se débrouille de concert avec tous les autres pour éviter les conflits d'accès au bus : chaque composant décide seul d'émettre ou pas, suivant l'état du bus.

L'**arbitrage par multiplexage temporel** peut se résumer en une phrase : chacun son tour ! Chaque composant a accès au bus à tour de rôle, durant un temps fixe. Cet arbitrage est mal adapté quand certains composants effectuent beaucoup de transactions sur le bus et d'autres très peu. Une autre méthode est celle de l'**arbitrage par requête**, qui se résume à un simple « premier arrivé, premier servi » ! L'idée est que tout composant peut réserver le bus si celui-ci est libre, mais doit attendre si le bus est déjà réservé. Pour savoir si le bus est réservé, il existe deux méthodes :

- soit chaque composant peut vérifier à tout moment si le bus est libre ou non (aucun composant n'écrit dessus) ;
- soit on rajoute un bit qui indique si le bus est libre ou occupé : le bit busy.

Certains protocoles permettent de libérer le bus de force pour laisser la place à un autre composant. Sur certains bus, certains composants sont prioritaires, et les circuits chargés de l'arbitrage libèrent le bus de force si un composant plus prioritaire veut utiliser le bus. D'autres préemptent le bus : ils donnent l'accès au bus à un composant durant un certain temps fixe, même si un composant qui n'utilise pas totalement le temps qui lui est attribué peut libérer le bus prématurément.

Interfaçage avec le bus

Une fois que l'on sait quel composant a accès au bus à un instant donné, il faut trouver un moyen pour que les composants non sélectionnés par l'arbitrage ne puissent pas écrire sur le bus. Une première solution consiste à relier les sorties des composants au bus via un multiplexeur : on est alors certain que seul un composant pourra émettre sur le bus à un moment donné. L'arbitrage du bus choisit quel composant peut émettre, et configure l'entrée de commande du multiplexeur en fonction.

Une autre solution consiste à déconnecter du bus les sorties qui n'envoient pas de données. Plus précisément, leurs sorties peuvent être mises dans un état de haute impédance, qui n'est ni un 0 ni un 1 : quand une sortie est en haute impédance, celle-ci n'a pas la moindre influence sur les

composants auxquels elle est reliée. Un composant dont les sorties sont en haute impédance ne peut pas influencer le bus et ne peut donc pas y écrire.

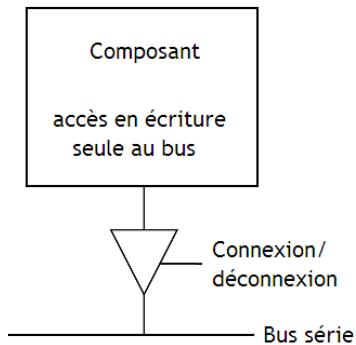
Pour mettre une sortie en état de haute impédance, on utilise des **circuits trois états**, qui possèdent une entrée de donnée, une entrée de commande, et une sortie : suivant ce qui est mis sur l'entrée de commande, la sortie est soit en état de haute impédance (déconnectée du bus), soit dans l'état normal (0 ou 1).

Commande	Entrée	Sortie
0	0	Haute impédance/Déconnexion
0	1	Haute impédance/Déconnexion
1	0	0
1	1	1

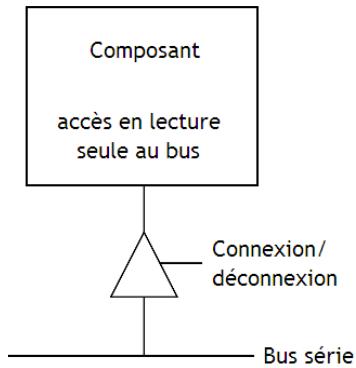
Pour simplifier, on peut voir ceux-ci comme des interrupteurs :

- si on envoie un 0 sur l'entrée de commande, ces circuits trois états se comportent comme un interrupteur ouvert ;
- si on envoie un 1 sur l'entrée de commande, ces circuits trois états se comportent comme un interrupteur fermé.

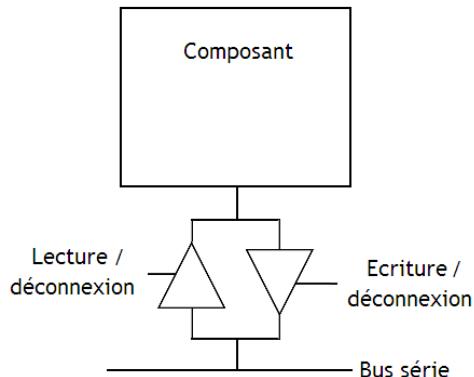
On peut utiliser ces circuits trois états pour permettre à un composant d'émettre ou de recevoir des données sur un bus. Par exemple, on peut utiliser ces composants pour autoriser les émissions sur le bus, le composant étant déconnecté (haute impédance) si jamais il n'a rien à émettre. Le composant a accès au bus en écriture seule. L'exemple typique est celui d'une mémoire ROM reliée à un bus de données.



Une autre possibilité est de permettre à un composant de recevoir des données sur le bus. Le composant peut alors surveiller le bus et regarder si des données lui sont transmises, ou se déconnecter du bus. Le composant a alors accès au bus en lecture seule.



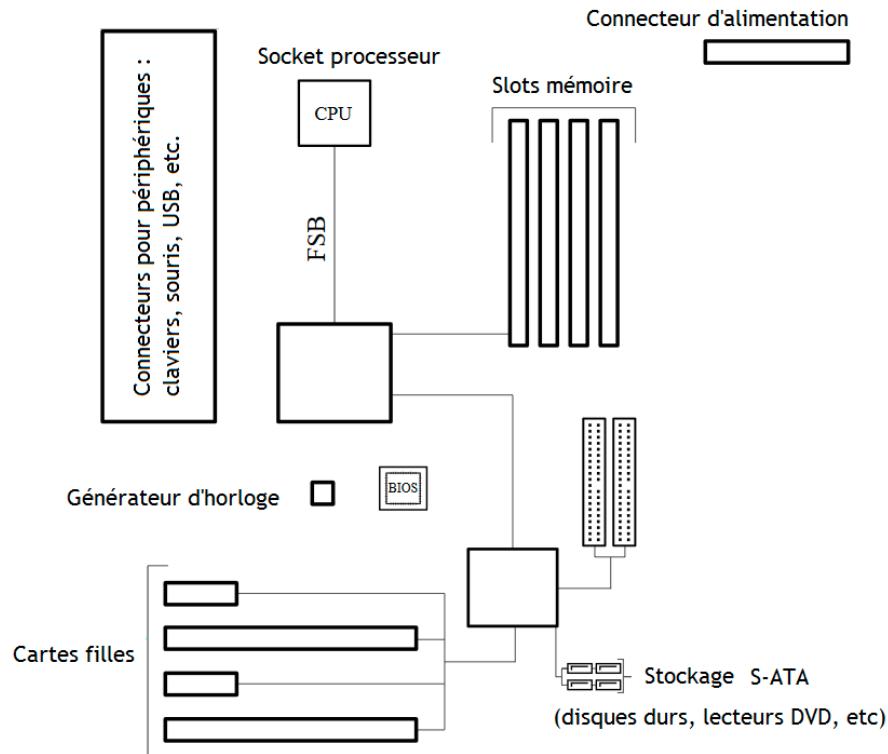
Évidemment, on peut autoriser lectures et écritures : le composant peut alors aussi bien émettre que recevoir des données sur le bus quand il s'y connecte. On doit alors utiliser deux circuits trois états, un pour l'émission/écriture et un autre pour la réception/lecture. Comme exemple, on pourrait citer les mémoires RAM, qui sont reliées au bus mémoire par des circuits de ce genre. Dans ce cas, les circuits trois états doivent être commandés par le bit CS (qui connecte ou déconnecte la mémoire du bus), mais aussi par le bit R/W qui décide du sens de transfert. Pour faire la traduction entre ces deux bits et les bits à placer sur l'entrée de commande des circuits trois états, on utilise un petit circuit combinatoire assez simple.



La carte mère

Dans un ordinateur, les composants sont placés sur un circuit imprimé (la **carte mère**), un circuit sur lequel on vient connecter les différents composants d'un ordinateur, et qui relie ceux-ci via divers bus.

- Le processeur vient s'enchâsser dans la carte mère sur un connecteur particulier : le **socket**. Celui-ci varie suivant la carte mère et le processeur, ce qui est source d'incompatibilités.
- Les barrettes de mémoire RAM s'enchâssent dans un autre type de connecteurs : les **slots mémoire**.
- Les mémoires de masse disposent de leurs propres connecteurs : connecteurs P-ATA pour les anciens disques durs, et S-ATA pour les récents.
- Les périphériques (clavier, souris, USB, Firewire, ...) sont connectés sur un ensemble de connecteurs dédiés, localisés à l'arrière du boîtier de l'unité centrale.
- Les autres périphériques sont placés dans l'unité centrale et sont connectés via des connecteurs spécialisés. Ces périphériques sont des cartes imprimées, d'où leur nom de **cartes filles**. On peut notamment citer les cartes réseau, les cartes son, ou les cartes vidéo.



Générateur d'horloge

Outre cet ensemble de composants, la carte mère contient aussi quelques composants spécialisés. Le premier est le **générateur d'horloge**, le composant qui génère le signal d'horloge envoyé au processeur, la mémoire RAM, et aux différents bus. Celui-ci est généralement basé sur un dispositif à quartz, similaire à celui présent dans nos montres électroniques. La fréquence de base fournie par le quartz est de 32768 Hz, soit 2^{15} cycles d'horloge par seconde. Celle-ci est multipliée par divers circuits pour obtenir l'ensemble des fréquences de l'ordinateur.

CMOS RAM

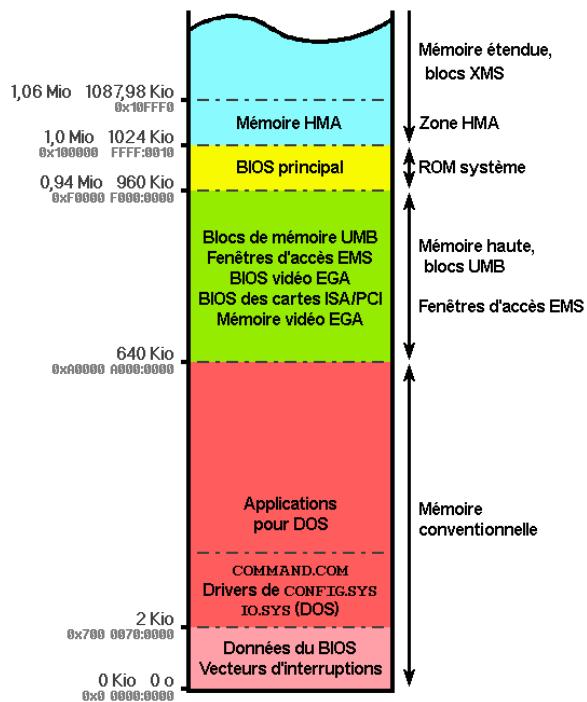
Dans un ordinateur, et dans tout composant électronique capable de conserver une heure ou une date, on trouve une mémoire capable de retenir l'heure et la date. Il s'agit souvent d'une mémoire fabriquée avec des transistors CMOS, d'où son nom de CMOS RAM. D'autres circuits additionnels permettent de mettre à jour l'heure, les minutes, les secondes ou la date présentes dans la CMOS SRAM (en cas de changement d'heure, par exemple). Ceux-ci sont reliés à l'horloge, afin de compter les secondes, minutes, heures qui s'écoulent. Cette mémoire est une mémoire SRAM, et non une mémoire EEPROM, comme on pourrait le penser. Vu qu'il s'agit d'une mémoire RAM, volatile, celle-ci doit être alimentée en permanence. Pour cela, il nous faut une source d'énergie qui remplace l'alimentation secteur lorsque l'ordinateur est débranché. Cette source d'énergie est quasiment toujours une pile au lithium installée sur la carte mère. L'enlever et la remettre réinitialise la **CMOS RAM**. Cette mémoire, la mémoire CMOS, est adressable, mais on y accède indirectement, comme si c'était un périphérique : le contenu de la CMOS RAM ne peut pas être écrit ou lu directement case mémoire par case mémoire par notre processeur, mais il peut envoyer des ordres de lecture ou des informations à écrire une par une sur une adresse bien précise. On y accède via les adresses 0x0007 0000 et 0x0007 0001 (ces adresses sont écrites en hexadécimal).

BIOS

Sur les PC avec un processeur x86, il existe un programme, lancé automatiquement lors du démarrage, qui se charge du démarrage avant de rendre la main au système d'exploitation. Ce programme s'appelle le **BIOS système**, communément appelé BIOS. Autrefois, le système d'exploitation déléguait la gestion des périphériques au BIOS. Ce programme est mémorisé dans la mémoire EEPROM, ce qui permet de mettre à jour le programme de démarrage : on appelle cela flasher le BIOS. En plus du BIOS système, les cartes graphiques actuelles contiennent toutes un **BIOS vidéo**, une mémoire ROM ou EEPROM qui contient des programmes capables d'afficher du texte et des graphismes monochromes ou 256 couleurs à l'écran. Lors du démarrage de l'ordinateur, ce sont ces routines qui sont utilisées pour gérer l'affichage avant que le système d'exploitation ne lance les pilotes graphiques. De même, des cartes d'extension peuvent avoir un BIOS. On peut notamment citer le cas des cartes réseaux, certaines permettant de démarrer un ordinateur sur le réseau. Ces BIOS sont ce qu'on appelle des **BIOS d'extension**. Si le contenu des BIOS d'extension dépend fortement du périphérique en question, ce n'est pas du tout le cas du BIOS système, dont le contenu est relativement bien standardisé.

Accès au BIOS

Le BIOS est mappé en mémoire (certaines adresses mémoire sont redirigées vers le BIOS). Il faut noter que le processeur démarre systématiquement en mode réel, un mode d'exécution spécifique aux processeurs x86, où le processeur n'a accès qu'à 1 mégaoctet de mémoire (les adresses font 20 bits maximum). Par la suite, le système d'exploitation basculera en mode protégé, un mode d'exécution où il peut utiliser des adresses mémoires de 32/64 bits. Dans les grandes lignes, le premier mégaoctet de mémoire est décomposé en deux portions de mémoire : les premiers 640 kibioctets sont ce qu'on appelle la **mémoire conventionnelle**, alors que les octets restants forment la **mémoire haute**. La mémoire conventionnelle sert à exécuter des programmes en mode réel sur certains systèmes d'exploitation (MS-DOS, avant sa version 5.0). Le BIOS peut aussi utiliser une portion de la mémoire conventionnelle pour mémoriser des informations diverses : cette portion de RAM s'appelle la **BIOS Data Area**. Celle-ci commence à l'adresse 0040:0000h et a une taille de 255 octets. Elle est initialisée lors du démarrage de l'ordinateur. Par contre, les octets de la mémoire haute servent pour communiquer avec les périphériques. On y trouve aussi le BIOS vidéo (s'il existe). Tout le reste de la mémoire, au-delà du premier mégaoctet, est ce qu'on appelle la **mémoire étendue**.



Démarrage de l'ordinateur

Au démarrage de l'ordinateur, le processeur est initialisé de manière à commencer l'exécution des instructions à partir de l'adresse 0xFFFF:0000h (l'adresse maximale en mémoire réelle moins 16 octets). C'est à cet endroit que se trouve le BIOS. Le BIOS s'exécute et initialise l'ordinateur avant de laisser la main au système d'exploitation. En cas d'erreur à cette étape, le BIOS émet une séquence de bips, la séquence dépendant de l'erreur et de la carte mère. Pour cela, le BIOS est relié à un buzzer placé sur la carte mère. Si vous entendez cette suite de bips, la lecture du manuel de la carte mère vous permettra de savoir quelle est l'erreur qui correspond.

Les premiers éléments vérifiés lors du **POST (Power On Self Test)** sont la stabilité de l'alimentation électrique de l'ordinateur, la stabilité de l'horloge, et l'intégrité des 640 premiers kibioctets de la mémoire. Ensuite, les périphériques sont détectés, testés et configurés pour garantir leur fonctionnement. Pour cela, le BIOS scanne la mémoire haute à la recherche des périphériques, tout en détectant la présence d'autres BIOS d'extension. Le BIOS est conçu pour lire la mémoire haute, par pas de 2 kibioctets. Par exemple, le BIOS regarde s'il y a un BIOS vidéo aux adresses mémoire 0x000C:0000h et 0x000E:0000h. Pour détecter la présence d'un BIOS d'extension, le BIOS système lit ces adresses et y recherche une signature, une valeur bien précise qui indique qu'une ROM est présente à cet endroit : la valeur en question vaut 0x55AA. Cette valeur est suivie par un octet qui indique la taille de la ROM, lui-même suivi par le code du BIOS d'extension. Si un BIOS d'extension est détecté, le BIOS système lui passe la main, grâce à un branchement vers l'adresse du code du BIOS d'extension. Ce BIOS peut alors faire ce qu'il veut, mais il finit par rendre la main au BIOS (avec un branchement) quand il a terminé son travail.

À ce stade du démarrage, une **interface graphique** s'affiche. La majorité des cartes mères permettent d'accéder à une interface pour configurer le BIOS, en appuyant sur F1 ou une autre touche lors du démarrage. Cette interface donne accès à plusieurs options modifiables, qui permettent de configurer le matériel. Ces paramètres sont stockés dans une mémoire flash ou EEPROM séparée du BIOS, généralement fusionnée avec la CMOS, qui est lue par le BIOS à l'allumage de l'ordinateur. Cette mémoire, la mémoire CMOS, est adressable via les adresses 0x0007:0000 et 0x0007:0001.

Par la suite, le BIOS va lancer le système d'exploitation. Dans tous les PC actuels, le système d'exploitation est placé sur le disque dur et doit être chargé dans la RAM. Pour cela, il doit utiliser des informations mémorisées dans le **Master Boot Record**, le premier secteur du disque dur. Un MBR est assez bien organisé et contient trois grandes parties aux usages différents.

- Les premiers octets du MBR sont remplis par le **bootloader**, un programme servant à charger le système d'exploitation. Il existe parfois une portion du code exécutable qui contient les messages d'erreur à afficher dans le cas où le lancement du système d'exploitation échouerait. Sur les BIOS récents, cette portion se termine par quelques octets de signature et d'anticopie.
- La **table des partitions** contient des informations sur les différentes partitions installées sur le disque dur : leur « nom », leur taille, et leur localisation sur le disque dur (adresse LBA). On ne peut créer que quatre lignes dans cette table, on se trouve donc limité à quatre partitions principales. Il faut dire que cette table ne fait que 64 octets (elle a été conçue comme cela).
- Les deux derniers octets du MBR doivent avoir une valeur bien précise pour que le BIOS autorise l'exécution. Cette valeur, appelée le **nombre magique**, vaut 0xAA55, ce qui correspond à 43 605 en décimal. Mais le résultat est encore plus joli en binaire : 1010101001010101.

Code exécutable	Table des partitions				Valeur magique
	Entrée de partition	Entrée de partition	Entrée de partition	Entrée de partition	55 AA
446 Bytes	16 Bytes	16 Bytes	16 Bytes	16 Bytes	1 Byte 1 Byte
	64 Bytes				2 Bytes

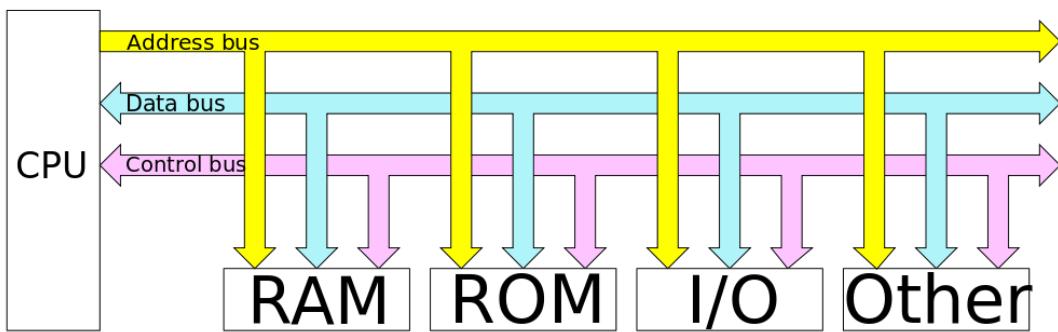
Chipset, backplane bus, et autres

L'organisation des cartes mères de nos ordinateurs a évolué au cours du temps pendant que de nombreux bus apparaissaient. On considère qu'il existe trois générations de cartes mères bien distinctes :

- une première génération avec un bus unique, la plus ancienne ;
- une seconde génération avec des bus segmentés ;
- une troisième génération avec une intégration du *chipset* nord dans le processeur.

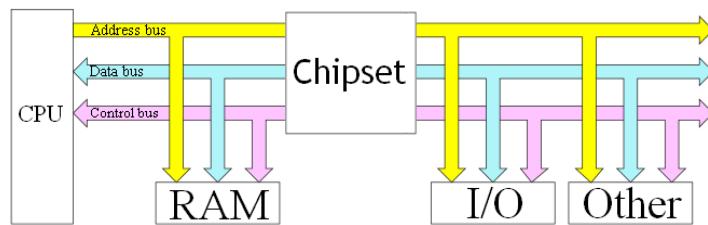
Première génération

Pour les bus de première génération, un seul et unique bus reliait tous les composants de l'ordinateur. Ce bus s'appelait le **bus système** ou backplane bus. Ces bus de première génération avaient le fâcheux désavantage de relier des composants allant à des vitesses très différentes : il arrivait fréquemment qu'un composant rapide doive attendre qu'un composant lent libère le bus. Le processeur était le composant le plus touché par ces temps d'attente.



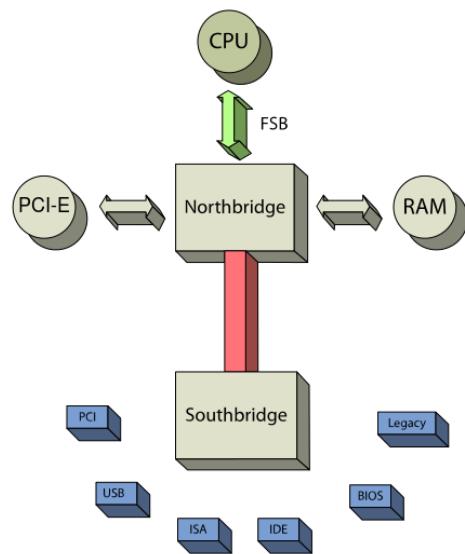
Seconde génération

Pour régler ce problème, on a divisé le bus système en deux : un bus pour les périphériques lents, et un autre pour les périphériques rapides. Deux composants lents peuvent ainsi communiquer entre eux sans avoir à utiliser le bus reliant les périphériques rapides, qui est alors utilisable à volonté par les périphériques rapides (et vice versa). Ces deux bus étaient reliés par un composant nommé le **chipset**, chargé de faire la liaison et de transmettre les données d'un bus à l'autre.



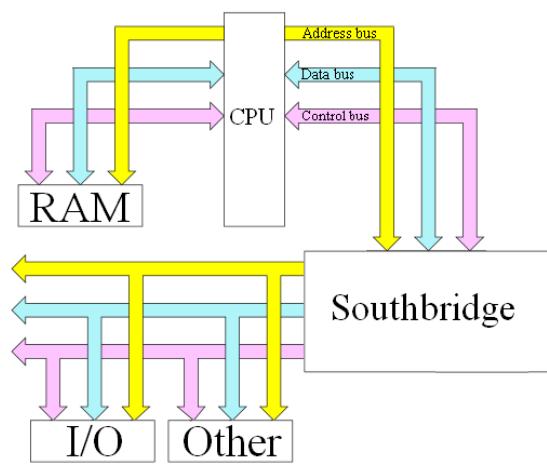
Sur certains ordinateurs, le chipset est divisé en deux :

- Le northbridge, pour le bus des composants rapides ;
- Le southbridge, pour le bus des périphériques lents.



Troisième génération

Sur les cartes-mères récentes, le northbridge est intégré au processeur.



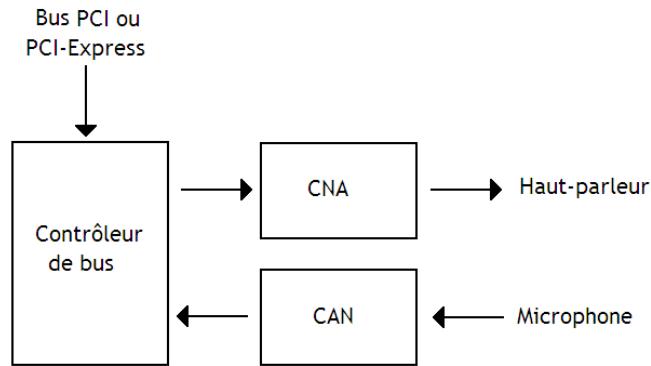
Les périphériques

Après avoir vu les composants internes à un ordinateur, il est temps de passer aux composants connectés sur l'unité centrale, les fameux **périphériques**. Dans ce chapitre, nous allons étudier les cartes son, les cartes graphiques, les claviers, l'écran, etc.

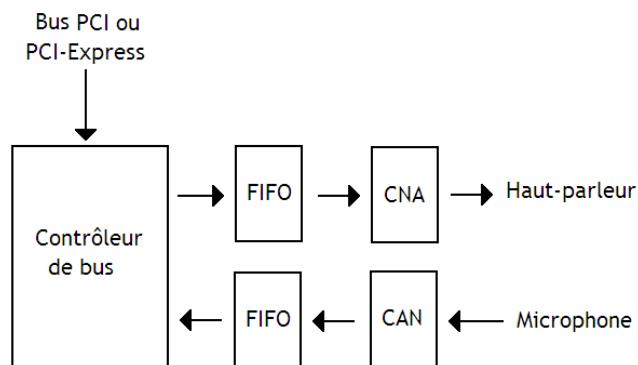
La carte son

Dans un ordinateur, toutes les données sont codées sous forme numérique, c'est à dire sous la forme de nombres entiers/ou avec un nombre de chiffres fixes après la virgule. Les ordinateurs actuels utilisent du binaire. Les informations sonores sont aussi stockées dans notre PC en binaire. Toutes vos applications qui émettent du son sur vos haut-parleurs stockent ainsi du son en binaire/numérique. Seulement, les haut-parleurs et microphone sont des composants dits analogiques, qui codent ou décoden l'information sonore sous la forme d'un tension électrique, qui peut prendre toutes les valeurs comprises dans un intervalle. Dans ces conditions, il faut bien faire l'interface entre les informations sonores numériques du PC, et le haut-parleur ou micro-phone analogique. Ce rôle est dévolu à un composant électronique qu'on appelle la **carte son**.

Une carte son est composée de plusieurs sous-composants. Le premier recueille la tension transmise par le micro-phone et la convertit en binaire. Ce composant est le **convertisseur analogique-numérique**, ou CAN. Celui-ci est relié à un composant qui mesure la tension envoyée par le microphone à intervalles régulier : l'**échantillonneur**. Le second composant transforme des informations binaires en tension à envoyer à un haut-parleur. Ce composant s'appelle le **convertisseur numérique-analogique**, ou CNA. Évidemment, on trouve aussi des circuits chargés de gérer les transferts entre la mémoire et la carte son, et éventuellement un processeur pour prendre en charge divers effets sonores.



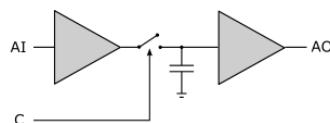
En aval du CAN, on trouve une petite mémoire dans laquelle sont stockées les informations qui viennent d'être numérisée. Cette mémoire sert à interfaçer le processeur avec le CNA : le processeur est forcément très rapide, mais il n'est pas disponible à chaque fois que la carte son a fini de traduire un son. Les informations numérisées sont donc accumulées dans une petite mémoire FIFO en attendant que le processeur puisse les traiter. La même chose a lieu pour le CNA. Le processeur étant assez pris, il n'est pas forcément capable d'être disponible tous les 10 millisecondes pour envoyer un son à la carte son. Dans ces conditions, le processeur prépare à l'avance une certaine quantité d'information, suffisamment pour tenir durant quelques millisecondes. Une fois qu'il dispose de suffisamment d'informations sonores, il va envoyer le tout en une fois à la carte son. Celle-ci stockera ces informations dans une petite mémoire FIFO qui les conservera dans l'ordre. Elle convertira ces informations au fur et à mesure, à un fréquence régulière.



Échantillonneur

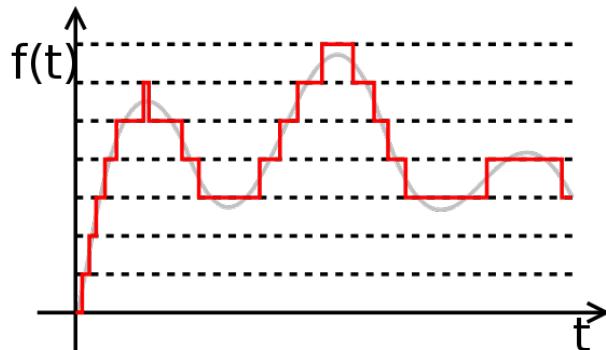
La tension transmise par le microphone varie de manière continue, ce qui rend sa transformation en numérique difficile. Pour éviter tout problème, la valeur de la tension est mesurée à intervalles réguliers, tout les 20 millisecondes par exemple. On parle alors d'**échantillonnage**. Le nombre de fois que notre tension est mesurée par seconde s'appelle la **fréquence d'échantillonnage**. Pour donner quelques exemples, le signal sonore d'un CD audio a été échantillonné à 44,1 kHz, c'est à dire 44100 fois par secondes. Plus cette fréquence est élevée, plus le son sera de qualité, proche du signal analogique mesuré. C'est ce qui explique qu'en augmenter la fréquence d'échantillonnage augmente la quantité de mémoire nécessaire pour stocker le son. Sur les cartes sons actuelles, il est possible de configurer la fréquence d'échantillonnage.

L'échantillonnage est réalisé par un circuit appelé l'**échantillonneur-bloqueur**. L'échantillonneur-bloqueur le plus simple ressemble au circuit du schéma ci-dessous. Les triangles de ce schéma sont ce qu'on appelle des amplificateurs opérationnels, mais on n'a pas vraiment à s'en préoccuper. Dans ce montage, ils servent juste à isoler le condensateur du reste du circuit, en ne laissant passer les tensions que dans un sens. L'entrée C est reliée à un signal d'horloge qui ouvre ou ferme l'interrupteur à fréquence régulière. La tension va remplir le condensateur quand l'interrupteur se ferme. Une fois le condensateur rempli, l'interrupteur est déconnecté isolant le condensateur de la tension d'entrée. Celui-ci mémorisera alors la tension d'entrée jusqu'au prochain échantillonnage.

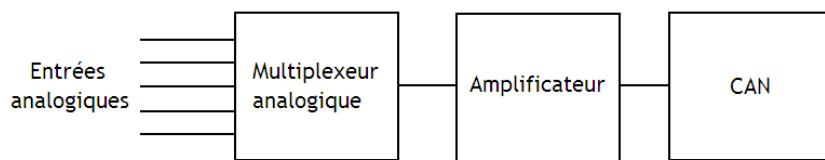


Convertisseur analogique-numérique

Le CAN convertit la tension du microphone en nombres codés en binaire. Un signal analogique ne peut pas être traduit en numérique sans pertes, l'infinie de valeurs d'un intervalle de tension ne pouvant être codé sur un nombre fini de bits. La tension envoyée va ainsi être arrondie à une tension qui peut être traduite en un entier sans problème. Cette perte de précision va donner lieu à de petites imprécisions qui forment ce qu'on appelle le **bruit de quantification**. Plus le nombre de bits utilisé pour encoder la valeur numérique est élevée, plus ce bruit est faible. Sur les cartes sons actuelles, ce nombre de bits porte un nom : c'est la **Résolution de la carte son**. Celui-ci varie entre 16 et 24 bits sur les cartes sons récentes.

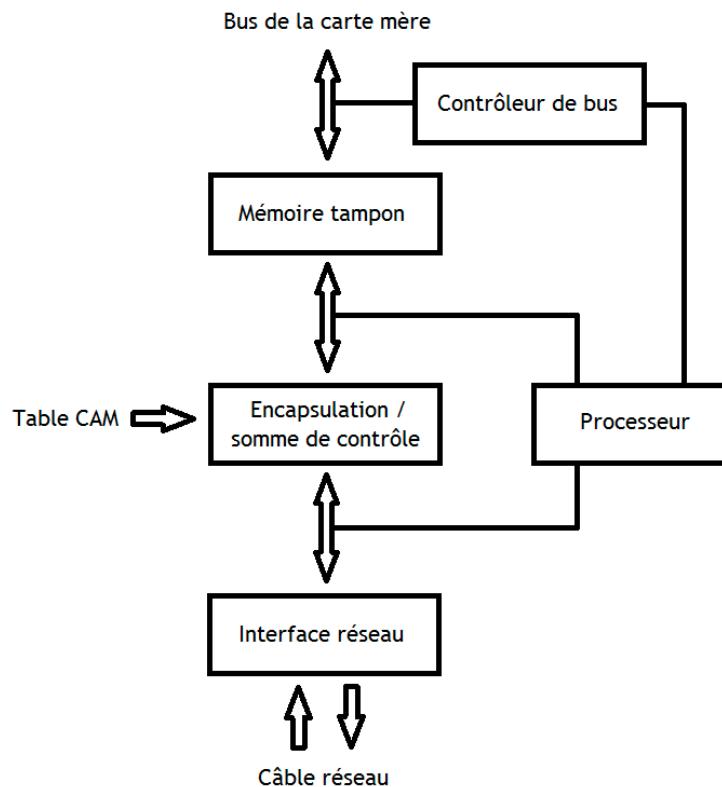


Une carte son peut supporter plusieurs entrées analogiques. Malheureusement, cela coûterait trop cher de mettre plusieurs CAN sur une carte son. A la place, les concepteurs de cartes sons mutualisent le CAN sur plusieurs entrées analogiques grâce à un **multiplexeur analogique**. Ce multiplexeur récupère les tensions des différentes entrées, et en choisira une qui est recopiée sur la sortie. Ce multiplexeur comporte une entrée de commande qui permet de choisir quelle entrée sera choisie pour être recopiée sur la sortie. Ce multiplexeur est ensuite suivi par un **amplificateur**, qui fait rentrer la tension fournie en entrée dans un intervalle de tension compatible avec le CAN.



La carte réseau

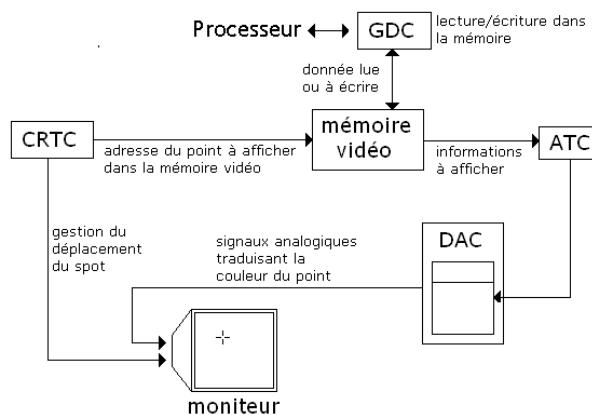
La **carte réseau** est le composant qui permet à notre ordinateur de communiquer sur un réseau (local ou internet). LA plupart permet d'envoyer ou de recevoir des informations sur un câble réseau ou une connexion WIFI. Elle communique avec le reste de l'ordinateur via le bus de la carte mère. Les données échangées sont mémorisées temporairement dans une mémoire tampon. Celle-ci permet de mettre en attente les données à envoyer tant que le réseau n'est pas disponible, ou d'accumuler les données reçues en attendant de les recevoir complètement. Ces données sont ensuite gérées par un circuit qui s'occupe de gérer les aspects réseau de la transmission (ajout/retrait des adresses MAC pour l'encapsulation, calcul de la somme de contrôle). La carte réseau contient ensuite un circuit qui transforme les données à transmettre en ondes WIFI ou en signaux électriques (pour les câbles réseau). Dans tous les cas, les transferts d'informations se font en série (le câble est l'équivalent d'un bus série). L'interface de transfert contient donc deux registres à décalage : un pour faire la conversion parallèle -> série, et un autre pour la conversion série -> parallèle.



La carte graphique

Les cartes graphiques sont des cartes qui s'occupent de communiquer avec l'écran, pour y afficher des images. Au tout début de l'informatique, ces opérations étaient prises en charge par le processeur : celui-ci calculait l'image à afficher à l'écran, et l'envoyait pixel par pixel à l'écran, ceux-ci étant affichés immédiatement après. Cela demandait de synchroniser l'envoi des pixels avec le rafraîchissement de l'écran.

Pour simplifier la vie des programmeurs, les fabricants de matériel ont inventé des cartes d'affichage, ou cartes vidéo. Avec celles-ci, le processeur calcule l'image à envoyer à l'écran, et la transmet à la carte d'affichage. Celle-ci prend alors en charge son affichage à l'écran, déchargeant le processeur de cette tâche. De telles cartes vidéo étaient alors très simples : elles contenaient un contrôleur de bus, une mémoire vidéo, un CRTC qui s'occupait de lire les pixels dans l'ordre pour les envoyer à l'écran (suivant la fréquence de rafraîchissement de l'écran et sa résolution), et de circuits annexes (comme le DAC, qui convertissait les pixels en informations analogiques).



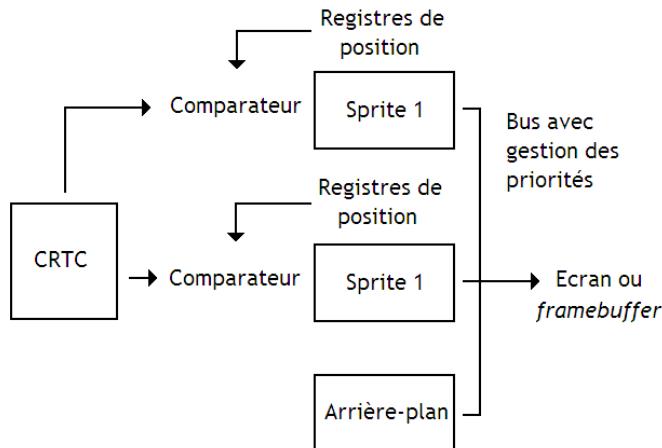
Les cartes accélératrices 2D

Avec l'arrivée des interfaces graphiques, les concepteurs de cartes vidéo ont permis à celles-ci d'accélérer le rendu 2D. La base d'un rendu en 2D est de superposer des images 2D précalculées les unes au-dessus des autres. Par exemple, on peut avoir une image pour l'arrière-plan (le décor), une image pour le monstre qui vous fonce dessus, une image pour le dessin de votre personnage, etc. Ces images sont superposées les unes au-dessus des autres, au bon endroit sur l'écran.

D'ordinaire, cela se traduit dans la mémoire vidéo par une copie des pixels de l'image à superposer sur l'image d'en-dessous. Les premières cartes graphiques 2D comprenaient des circuits pour faire ces copies directement dans la mémoire vidéo. Au lieu de laisser le processeur faire ces copies lui-même, en adressant la mémoire vidéo, il pouvait placer les sprites à rendre dans la mémoire vidéo et demander à la carte 2D de faire les copies elles-mêmes. Ceux-ci pouvaient aussi gérer les images transparentes (je ne détaille pas comment ici).

Avec d'autres cartes 2D, les sprites ne sont pas ajoutés sur l'arrière-plan : celui-ci n'est pas modifié. À la place, c'est la carte graphique qui décidera d'afficher les pixels de l'arrière-plan ou du sprite pendant l'envoi des pixels à l'écran, lors du balayage effectué par le CRTC. Pour cela, les sprites sont stockés dans des registres (voire des RAM pour l'arrière-plan). Pour chacune des RAM associée au sprite, on trouve trois registres permettant de mémoriser la position du sprite à l'écran : un pour sa coordonnée X, un autre pour sa coordonnée Y, et un autre pour sa profondeur (pour savoir celui qui est superposé au-dessus de tous les autres). Lorsque le CRTC demande à afficher le pixel à la position (X, Y), chacun des registres de position est alors comparé à la position envoyée par le CRTC. Si aucun sprite ne correspond, les mémoires des sprites sont

déconnectées du bus. Le pixel affiché est celui de l'arrière-plan. Dans le cas contraire, la RAM du sprite est connectée sur le bus, et son contenu est envoyé au framebuffer ou au RAMDAC. Si plusieurs sprites doivent s'afficher en même temps, le bus choisit celui dans la profondeur est la plus faible (celui superposé au-dessus de tous les autres).



Cette technique a autrefois été utilisée sur les anciennes bornes d'arcade, ainsi que sur certaines console de jeu bon assez anciennes. Mais de nos jours, elle est aussi présente dans les cartes graphiques actuelles dans un cadre particulièrement spécialisé : la prise en charge du curseur de la souris, ou le rendu de certaines polices d'écritures ! Les cartes graphiques contiennent un ou plusieurs sprites, qui représentent chacun un curseur de souris, et deux registres, qui stockent les coordonnées x et y du curseur. Ainsi, pas besoin de redessiner l'image à envoyer à l'écran à chaque fois que l'on bouge la souris : il suffit de modifier le contenu des deux registres, et la carte graphique place le curseur sur l'écran automatiquement. Pour en avoir la preuve, testez une nouvelle machine sur laquelle les drivers ne sont pas installés, et bougez le curseur : effet lag garantit !

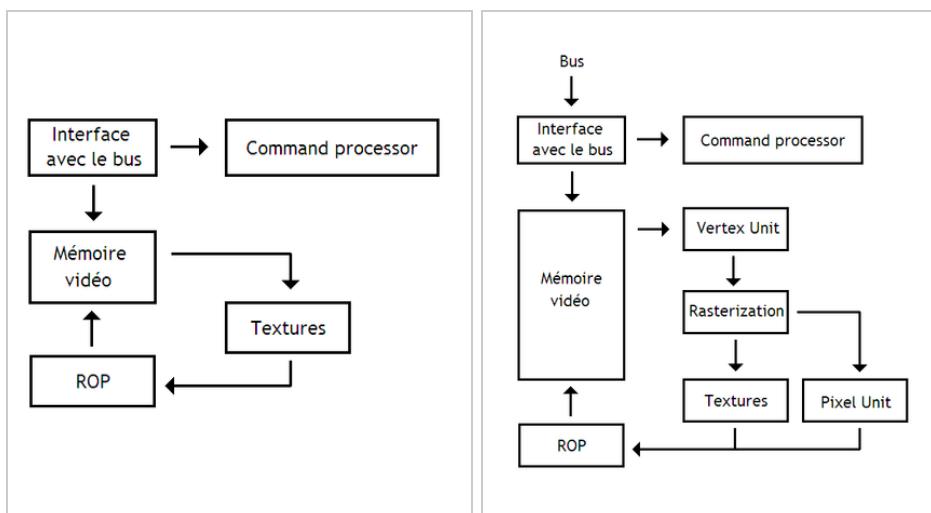
Les cartes accélératrices 3D

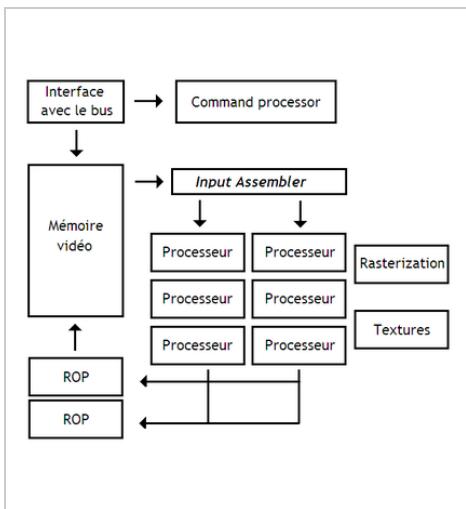
Le premier jeu à utiliser de la "vraie" 3D fut le jeu Quake, premier du nom (un excellent jeu, en passant). Et depuis sa sortie, presque tous les jeux vidéos un tant soit peu crédibles utilisent de la 3D. Face à la prolifération de ces jeux vidéos en 3D, les fabricants de cartes graphiques se sont adaptés et ont inventé des cartes capables d'accélérer les calculs effectués pour rendre une scène en 3D : les cartes accélératrices 3D.

Toutes premières cartes graphiques contenait simplement des circuits pour gérer les textures, ainsi qu'un framebuffer. Seules l'étape de texturing, quelques effets graphiques (brouillard) et l'étape d'enregistrement des pixels en mémoire étaient prises en charge par la carte graphique. L'étape de texturing est réalisée par un circuit spécialisé, l'unité de textures. L'enregistrement de l'image à afficher, ainsi que les calculs de profondeur de brouillard, et autres, sont réalisés par les Raster Operation Pipeline, ou ROP. Par la suite, ces cartes s'améliorent en ajoutant plusieurs circuits de gestion des textures, pour colorier plusieurs pixels à la fois : c'est ce qu'on appelle du multitexturing. Les cartes graphiques construites sur cette architecture sont très anciennes : ATI rage, 3DFX Voodoo, Nvidia TNT, etc.

Les cartes suivantes ajoutèrent une gestion des étapes de rasterization directement en matériel. Les cartes ATI rage 2, les Invention de chez Rendition, et d'autres cartes graphiques supportaient ces étapes en hardware. La première carte graphique capable de gérer la géométrie fut la GeForce 256, la toute première Geforce. Elle disposait pour cela d'un circuit pour manipuler les vertices et la géométrie. Pour plus d'efficacité, ces cartes graphiques possédaient parfois plusieurs unités de traitement des vertices et des pixels, ou plusieurs ROP. De nos jours, toutes les cartes 3D possèdent un tel circuit de traitement de la géométrie.

Par la suite, les circuits qui gèrent les calculs sur les pixels et les vertices sont devenus des processeurs programmables. Au tout début, seuls les traitements sur les vertices étaient programmables. C'était le cas sur les NVIDIA's GeForce 3, GeForce4 Ti, Microsoft's Xbox, et les ATI's Radeon 8500. Puis, les cartes suivantes ont permis de programmer les traitements sur les pixels : certains processeurs s'occupaient des vertices, et d'autres des pixels. Sur les cartes graphiques récentes, les processeurs peuvent traiter indifféremment pixels et vertices.





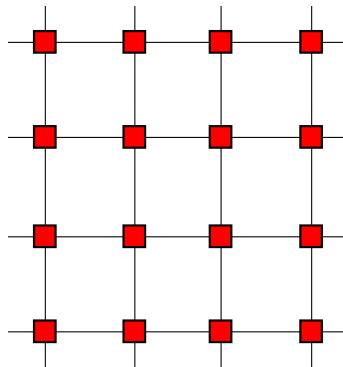
Carte 3D avec processeurs de shaders unifiés.

Le clavier

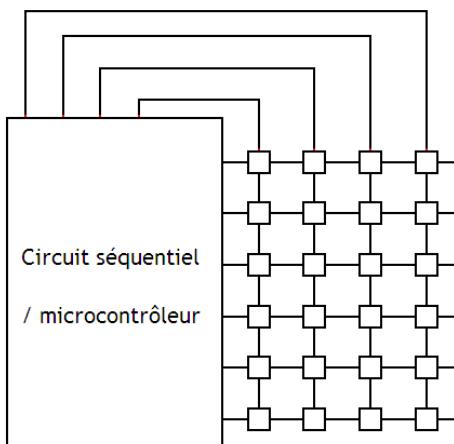
Le clavier communique avec notre ordinateur. Contrairement à ce qu'on pourrait penser, cette communication s'effectue dans les deux sens. Tout d'abord, notre clavier doit indiquer à notre ordinateur si une touche a été appuyée, et laquelle. Pour indiquer quelle touche a été appuyée, notre clavier va envoyer ce qu'on appelle un scancode à notre ordinateur. Ce scancode est un numéro : chaque touche du clavier se voit attribuer un scancode bien particulier, le même sur tous les claviers. Les scancodes des différentes touches sont standardisés, et il existe plusieurs standards de scancodes.

Les tout premiers PC utilisaient un jeu de scancode différent de celui utilisé actuellement. Le jeu de scancode utilisé s'appelait le **jeu de scancode XT**. Avec ce scancode, le scancode est, sauf exception, un octet dont les 7 bits de poids faible identifient la touche, et le bit de poids fort indique si la touche a été appuyée ou relâchée. De nos jours, les claviers utilisent le **jeu de scancode AT** s'ils sont reliés au PC via le port PS/2. Avec ce jeu de scancode, les numéros des touches sont modifiés. Le relâchement d'une touche est indiqué différemment : avec les scancodes AT, il faut ajouter un octet d'une valeur 0xF0 devant le numéro de la touche. Sans ce préfixe, on considère que la touche a été appuyée.

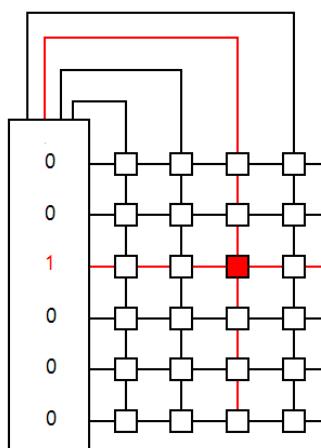
A l'intérieur d'un clavier, on trouve un circuit relié aux touches par des fils électriques, qui se charge de convertir l'appui d'une touche en scancode. Naïvement, on pourrait penser que ce contrôleur est relié à chaque touche, mais ce genre d'organisation n'est utilisable que pour de tout petits claviers. Avec un clavier à 102 touches, il nous faudrait utiliser 102 fils, ce qui serait difficile à mettre en œuvre. En fait, les touches sont reliées à des fils électriques, organisés en lignes et en colonnes, avec une touche du clavier à chaque intersection ligne/colonne. Pour simplifier, les touches agissent comme des interrupteurs : elles se comportent comme un interrupteur fermé quand elles sont appuyées, et comme un interrupteur ouvert quand elles sont relâchées.



Cette matrice est reliée à un circuit qui déduit les touches appuyées à partir de cette matrice de touches : le **Keyboard Encoder**. Ce circuit peut aussi bien être un circuit séquentiel fait sur mesure (rare), qu'un microcontrôleur. Dans les premiers claviers, il s'agissait d'un microcontrôleur Intel 8048. Les claviers actuels utilisent des microcontrôleurs similaires.



Pour savoir quelles sont les touches appuyées, ce micro-contrôleur va balayer les colonnes unes par unes, et regarder le résultat sur les lignes. Plus précisément, notre circuit va envoyer une tension sur la colonne à tester. Si une touche est enfoncée, elle connectera la ligne à la colonne, et on trouvera une tension sur la ligne en question. Cette tension est alors interprétée comme étant un bit, qui vaut 1. Si la touche à l'intersection entre ligne et colonne n'est pas enfoncée, la ligne sera déconnectée. Grâce à un petit circuit (des résistances de rappels au zéro volt intégrée dans le micro-contrôleur), cette déconnexion de la ligne et de la colonne est interprétée comme un zéro. Le microcontrôleur récupère alors le contenu des différentes lignes dans un octet. A partir de cet octet, il accède à une table stockée dans sa mémoire ROM, et récupère le scancode correspondant. Ce scancode est alors envoyé dans un registre à décalage, et est envoyé sur la liaison qui relie clavier et PC.



La souris

Les anciennes souris fonctionnaient avec une boule, en contact avec la surface/le tapis de souris, que le mouvement de la souris faisait rouler sur elle-même. Ce mouvement de rotation de la boule était capté par deux capteurs, un pour les déplacements sur l'axe gauche-droite, et un pour l'axe haut-bas.



Mécanisme interne d'une souris à boule.

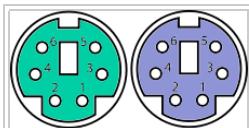
Les souris plus récentes contiennent une source de lumière, qui éclaire la surface sur laquelle est posée la souris (le tapis de souris). Cette source de lumière peut être une diode ou un laser, suivant la souris : on parle de **souris optique** si c'est une diode, et de **souris laser** si c'est un laser. Avec la source de lumière, on trouve une **caméra** qui photographie le tapis de souris intervalles réguliers, pour détecter tout mouvement de souris. Le tapis de souris n'est pas une surface parfaite, et contient des aspérités et des irrégularités à l'échelle microscopique. Quand on bouge la souris, les images successives prises par la caméra ne seront donc pas exactement les mêmes, mais seront en partie décalées à cause du mouvement de la souris. La caméra est reliée à un circuit qui détecte cette différence, et calcule le déplacement en question. Précisément, elle vérifie de combien l'image a bougé en vertical et en horizontal entre deux photographies de la surface. Ce déplacement est alors envoyé à l'ordinateur. Ce qui est envoyé est le nombre de pixels de différence entre deux images de caméra, en horizontal et en vertical. Le système d'exploitation va alors multiplier ce déplacement par un coefficient multiplicateur, la sensibilité souris, ce qui donne le déplacement du curseur sur l'écran en nombre de pixels. Plus celle-ci est faible, meilleure sera la précision.

Les OS actuels utilisent l'**accélération souris**, à savoir que la sensibilité est variable suivant le déplacement, le calcul du déplacement du curseur étant une fonction non-linéaire du déplacement de la souris. Sur les anciens OS Windows, la sensibilité augmentait par paliers : un déplacement souris faible donnait une sensibilité faible, alors qu'une sensibilité haute était utilisée pour les déplacements plus importants. Cette accélération souris est désactivable, mais cette désactivation n'est conseillée que pour jouer à des jeux vidéos, l'accélération étant utile sur le bureau et dans les autres interfaces graphiques. Elle permet en effet plus de précision lors des mouvements lents (vu que la sensibilité est faible), tout en permettant des mouvements de curseurs rapides quand la souris se déplace vite.

Performance de la souris

Plus la résolution de la caméra de la souris est élevée, plus celle-ci a tendance à être précise. Cette résolution de la caméra de la souris est mesurée en **DPI**, Dot Per Inch. Il s'agit du nombre de pixels que la caméra utilise pour capturer une distance de 1 pouce (2,54 cm). Il va de soi que plus le DPI est élevé, plus la souris sera sensible. Pour un même déplacement de souris et à sensibilité souris égale, une souris avec un fort DPI entraînera un déplacement du curseur plus grand qu'une souris à faible DPI. Cela est utile quand on utilise un écran à haute résolution, mais pas dans d'autres cas.

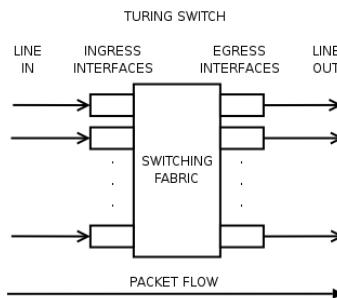
Un autre paramètre important de la souris est la **fréquence de rafraîchissement**, le nombre de fois qu'elle envoie des informations à l'ordinateur. Suivant le connecteur, celle-ci varie. Une souris connectée au connecteur PS/2 (celui situé l'arrière de l'unité centrale), a une fréquence par défaut de 40 hertz, mais peut monter à 200 Hz si on la configure convenablement dans le panneau de configuration. Un port USB a une fréquence de 125 Hz par défaut, mais peut monter jusqu'à 1000 Hz avec les options de configuration adéquate dans le pilote de la souris. Plus cette fréquence est élevée, meilleure sera la précision et plus le temps de réaction de la souris sera faible, ce qui est utile dans les jeux vidéos. Mais cela entraînera une occupation processeur plus importante pour gérer les interruptions matérielles générées par la souris.



Connecteurs PS/2 pour le clavier et la souris. Le vert est pour la souris et le violet pour le clavier.

Switchs et routeurs

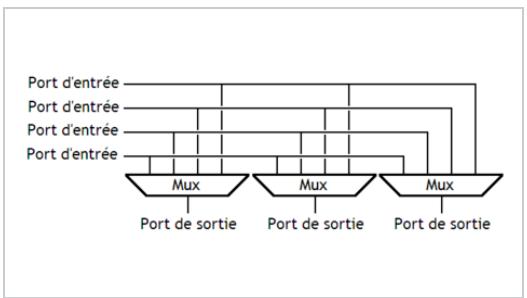
Un switch est un matériel réseau qui relie plusieurs composants électroniques ou informatiques entre eux, les interconnexions étant configurables. Il contient plusieurs ports d'entrées, sur lesquels on peut recevoir des paquets de données, et plusieurs ports de sorties, sur lesquels on peut envoyer des données. Dans certains cas, les ports d'entrée et de sortie sont confondus : un même port peut servir alternativement d'entrée et de sortie. Le rôle du switch, est de faire en sorte que les informations en provenance d'un port soient envoyées sur un autre, le port d'arrivée étant défini par la configuration du switch. Dans le cas le plus simple, on peut voir un switch comme un ensemble de connexions point à point configurables. Tout switch est composé d'au minimum une switch fabric, et de circuits qui gèrent les ports d'entrées-sortie.



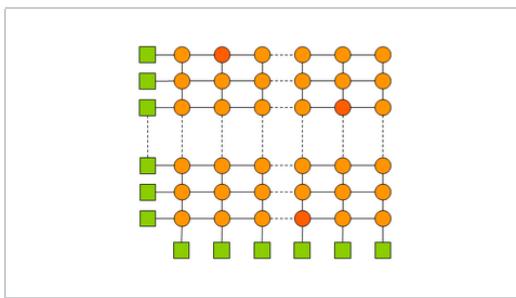
Switch fabric

Les techniques de broadcast ou de multicast permettent d'envoyer une donnée présentée sur un port d'entrée sur plusieurs sorties : on peut envoyer un message identique à plusieurs ordinateurs en même temps, sans devoir envoyer plusieurs copies. Certains switchs permettent de gérer cela directement dans le matériel, soit en dupliquant les paquets, soit en connectant une entrée sur plusieurs sorties. Certains switchs traitent les ports d'entrée à tour de rôle, l'un après l'autre : on parle de **switchs à partage de temps**, aussi appelés time-sharing switch chez les anglo-saxons. D'autres switchs permettent de gérer tous les ports d'entrée à la fois : on parle de **switchs à partage d'espace**, ou space-sharing switchs chez les anglo-saxons.

Les switchs à partage d'espace sont de loin les plus simples à comprendre. Ceux-ci sont simplement composés d'un ensemble de liaisons point à point, qui relient chacune un port d'entrée à un port de sortie. Chaque port d'entrée est relié à chaque port de sortie, chaque liaison pouvant être activée ou désactivée selon les besoins. Dans le cas le plus simple, on peut les concevoir avec des multiplexeurs, ou avec un **réseau crossbar**. Ce dernier est composé de fils organisés en lignes et en colonnes. À l'intersection de chaque ligne et de colonne, on trouve un interrupteur qui relie la ligne et la colonne. On peut utiliser plusieurs switchs crossbar pour former un switch plus gros, les différentes manières donnant respectivement un **réseau CLOS**, un **réseau de Benes**, un **réseau de banyan**, les **Switch Sunshine**, et bien d'autres.

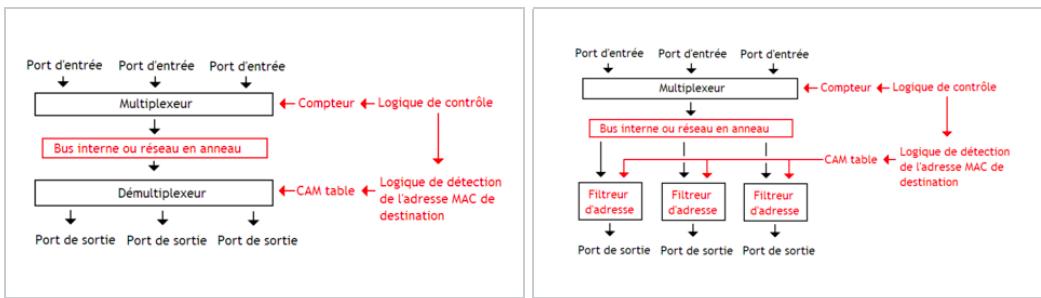


Switch conçu avec des multiplexeurs



Réseau Crossbar

Les switchs à partage de temps émulent les liaisons point à point à partir d'un bus ou d'un réseau en anneaux : ce sont les **switchs à média partagés**. Avec cette architecture, implémenter le multicast ou le broadcast est relativement complexe. Pour résoudre ce problème, il suffit de relier chaque port de sortie sur le bus interne directement, sans démultiplexeur. En faisant cela, chaque port de sortie doit filtrer les paquets qui ne lui sont pas destinés. Pour cela, on ajoute un filtre d'adresse pour comparer l'adresse MAC/IP associée au port (CAM table) et l'adresse MAC de destination : s'il y a égalité, alors on peut recopier la donnée sur le port de sortie. Ces switchs sont des switchs à partage de temps (sauf le tout premier).

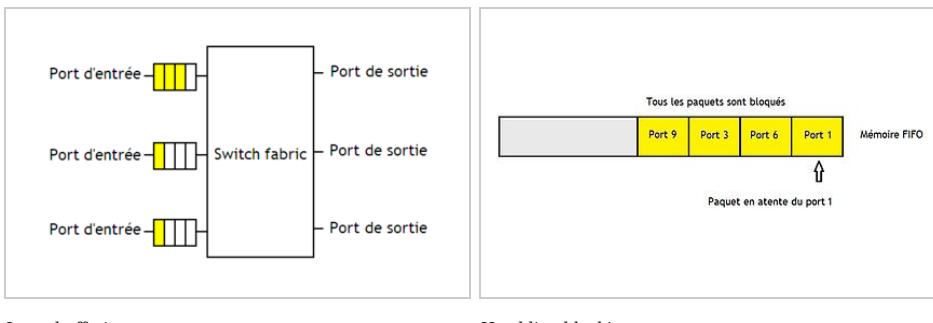


Switch à partage de temps conçu avec des multiplexeurs

Switch à média partagé

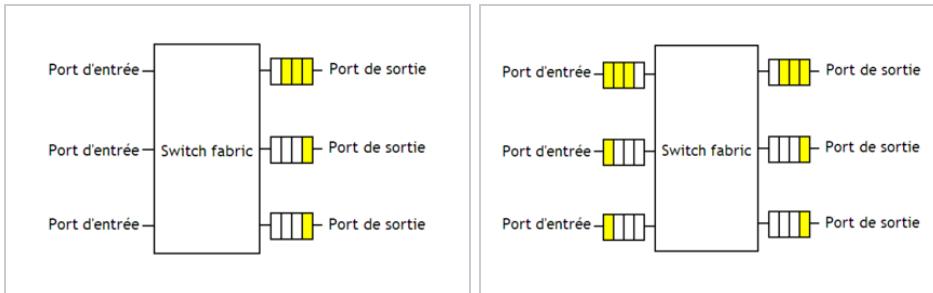
Arbitrage

La switch fabric est relativement lente comparée au temps d'envoi et de réception d'un paquet, ce qui fait qu'il est nécessaire de mettre ceux-ci en attente tant que la switch fabric est occupée avec le paquet précédent. De même, il faut gérer le cas où plusieurs paquets veulent accéder au même port de sortie, ce qui impose d'envoyer les paquets les uns après les autres, certains étant mis en attente (il arrive cependant que certains paquets soient perdus). Dans tous les cas, qui dit mise en attente dit : utilisation de mémoires tampons de type FIFO. Si les tampons sont remplies, les paquets en trop sont perdus et n'arrivent pas à destination : on laisse la situation entre les « mains » du logiciel. Si une requête est en attente via input buffering, elle va bloquer les requêtes suivantes sur le port d'entrée, même si celles-ci ont un port de sortie différent : on parle d'**head of line blocking**. Pour l'éliminer, certains switchs utilisent le Virtual output queing, à savoir découper chaque tampon FIFO en sous-tampons, chacun prenant en charge les paquets destinés à un port de sortie précis. Avec cette technique, on peut traiter les requêtes dans le désordre, afin de profiter au maximum des ports libres. Certains tampons FIFO sont placés entre la switch fabric et l'interface réseau de sortie, au cas où celle-ci soit plus lente que la switch fabric : on parle d'**output buffering**. D'autres sont placés en entrée de la switch fabric, celle-ci pouvant mettre du temps à traiter les paquets qui lui arrivent : on parle d'**input buffering**. Il est parfaitement possible d'utiliser les deux en même temps, ce qui porte le nom d'**input-output buffering**.



Input-buffering

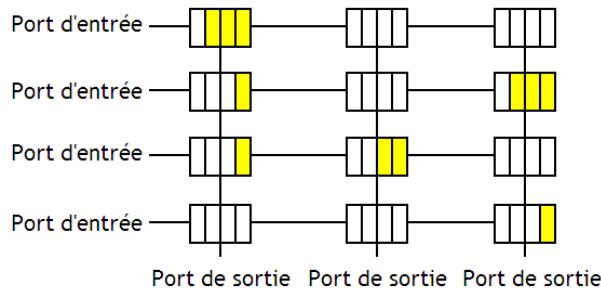
Head-line-blocking



Output-buffering

Input-output-buffering

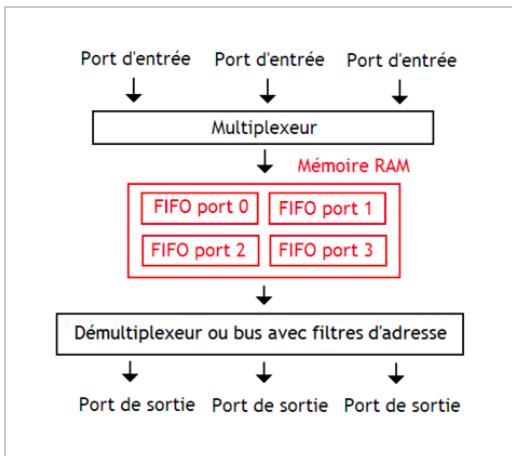
Enfin, certains switchs intègrent l'arbitrage directement dans la switch fabric, en intégrant les tampons FIFO dans celle-ci. Dans le cadre d'un switch à partage d'espace, à base de crossbar, l'incorporation de l'arbitrage est relativement aisée. Il suffit de remplacer les interrupteurs ligne/colonne par des tampons FIFO. Le nombre de FIFO est de $N * M$ pour un switch à N ports d'entrée et M ports de sortie. Pour les switchs qui ont un grand nombre de ports d'entrée et de sortie, cela devient rapidement impraticable. Pour éviter cela, les switchs à haute performance réduisent la taille des FIFO intégrées dans le crossbar, mais rajoutent de grosses FIFO sur les ports d'entrée (par ajout d'input buffering). Les simulations montrent que le rapport entre performance et nombre de portes logiques utilisées est meilleur avec cette technique.



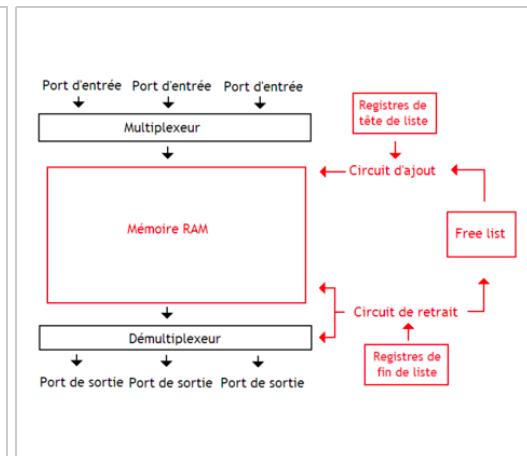
Pour les switch à partage d'espace, les tampons FIFO sont incorporés directement entre le MUX pour les ports d'entrée, et le reste (filtre).

rd'adresse ou DEMUX). Ces FIFOs sont généralement implémenté avec une mémoire RAM. D'où le nom de **switchs à mémoire partagée** donné à de tels switchs. Ces switchs permettent de gérer l'arbitrage directement dans la switch fabric : la mémoire centrale stocke les données des tampons FIFO d'arbitrage des ports d'entrée. Le switch présenté dans le schéma ci-dessous peut être transformé en switch sans partage de temps avec une mémoire multiport : il suffit d'avoir autant de ports d'écriture que de ports d'entrée, et d'adapter les circuits de gestion de la mémoire. Dans ce qui va suivre, mes schémas utiliseront un switch à partage de temps. Reste que ces tampons FIFO peuvent s'implémenter de différentes manières.

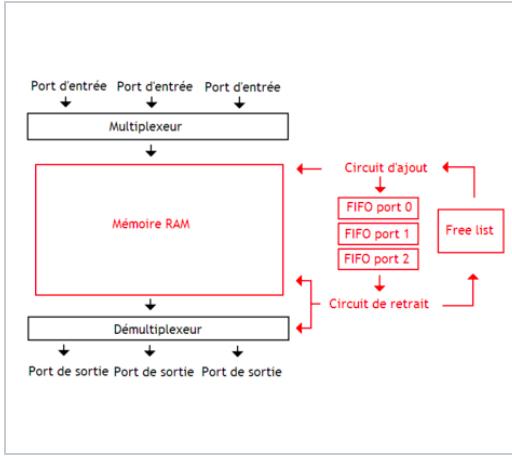
- Une première méthode mémorise les données dans des **listes doublement chaînées** dans la RAM. Les circuits qui gèrent la mémoire partagée doivent gérer eux-mêmes les listes chaînées, ce qui fait qu'ils contiennent deux registres par liste : un pour l'adresse en tête de liste et un pour le dernier élément. Il y a un circuit pour l'ajout et un autre pour le retrait des paquets. En plus de cela, les circuits de gestion de la mémoire doivent allouer dynamiquement les nœuds de la liste et libérer la mémoire. La mémoire utilise des Bytes démesurément grands, capables de mémoriser un paquet de plusieurs centaines de bits sans problèmes. Ainsi, le switch a juste besoin de mémoriser quels sont les Bytes libres et les Bytes occupés dans une mémoire annexe : la free list, souvent implémentée avec une mémoire FIFO.
- La deuxième méthode réserve la mémoire centrale pour le stockage des paquets et maintient l'ordre d'envoi des paquets dans des **tampons FIFO**, qui mémorisent les adresses des paquets dans la mémoire centrale. La taille fixe des mémoires FIFO ne permet pas d'allouer toute la mémoire pour seulement quelques ports. Mais cette organisation permet de gérer facilement le multicast et le broadcast : on peut facilement ajouter un paquet dans toutes les mémoires FIFO, simultanément avec un circuit d'ajout conçu pour.
- Une autre solution, nettement plus simple, consiste à remplacer la mémoire RAM du switch par un cache (une mémoire associative, plus précisément).



Switch à mémoire partagée.



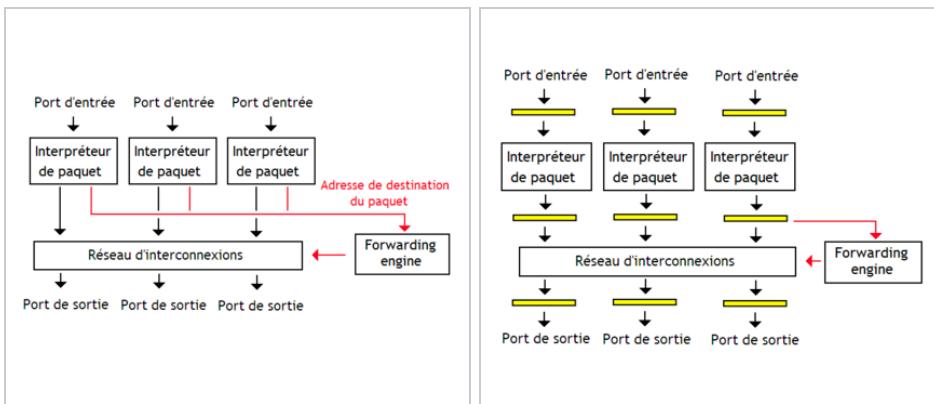
Linked-list switch



FIFO switch

Encapsulation

Une switch fabric ne suffit pas à créer un switch ou un routeur. En effet, l'un ou l'autre doit déterminer le port de destination en fonction de l'adresse MAC ou IP du paquet. La switch fabric doit être secondée par deux circuits. En premier lieu, on trouve un circuit qui va détecter les en-têtes IP ou ARP, et les interpréter : c'est l'**interpréteur de paquets**. A côté, on trouve un circuit qui va déterminer le port de destination : le **forwarding engine**. Le switch est un matériel qui utilise des adresses MAC, qui servent à identifier des cartes/équipements réseau (et non des ordinateurs, comme les adresses IP) : chaque paquet indique l'adresse MAC du destinataire et de l'émetteur. Pour les routeurs, le routage utilise des adresses IP. À l'intérieur du forwarding engine, on trouve des tables de correspondance qui associent chaque adresse à un port : la **CAM table** pour les adresses MAC et la **table de routage** pour les IP. Cette table de correspondance peut être implémentée de deux manières : avec une machine à états finis matérielle ou avec une mémoire RAM ou associative. La dernière solution est la plus utilisée, même si des solutions hybrides sont aussi relativement courantes (on peut notamment citer l'algorithme de routage nommé Logic-Based Distributed Routing).



Micro-architecture d'un switch. Les mémoires tampons de mise en attente des paquets ne sont pas représentées.

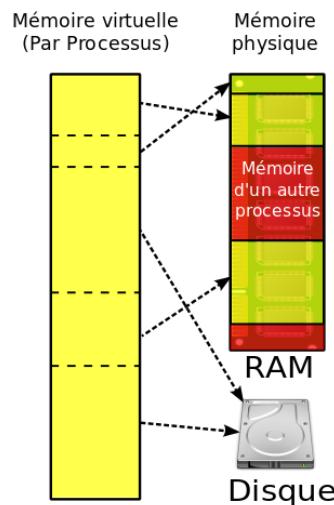
Il faut noter que le switch peut être pipeliné, pour gagner en performance : il suffit d'insérer des mémoires tampons entre les différents circuits cités au-dessus.

La mémoire virtuelle

Un programme peut être exécuté sur des ordinateurs ayant des capacités mémoires diverses et variées et dans des conditions très différentes. Et il faut faire en sorte qu'un programme fonctionne sur des ordinateur ayant peu de mémoire sans poser problème. Après tout, quand on conçoit un programme, on ne sait pas toujours quelle sera la quantité mémoire que notre ordinateur contiendra, et encore moins comment celle-ci sera partagée entre nos différentes programmes en cours d'exécution : s'affranchir de limitations sur la quantité de mémoire disponible est un plus vraiment appréciable. Ce besoin est appelé **l'abstraction matérielle de la mémoire**.

Enfin, plusieurs programmes sont présents en même temps dans notre ordinateur, et doivent se partager la mémoire physique. Si un programme pouvait modifier les données d'un autre programme, on se retrouverait rapidement avec une situation non prévue par le programmeur. Cela a des conséquences qui vont de comiques à catastrophiques, et cela fini très souvent par un joli plantage. Il faut donc introduire des mécanismes de **protection mémoire**. Pour éviter cela, chaque programme a accès à des portions de la mémoire dans laquelle lui seul peut écrire ou lire. Le reste de la mémoire est inaccessible en lecture et en écriture, à part pour la mémoire partagée entre différents programmes. Toute tentative d'accès à une partie de la mémoire non autorisée déclenchera une exception matérielle (rappelez-vous le chapitre sur les interruptions) qui devra être traitée par une routine du système d'exploitation. Généralement, le programme fautif est sauvagement arrêté et un message d'erreur est affiché à l'écran.

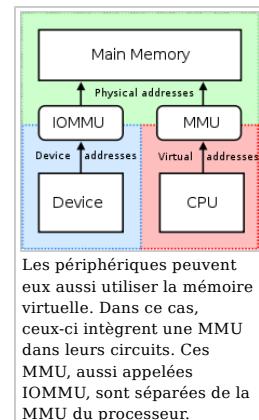
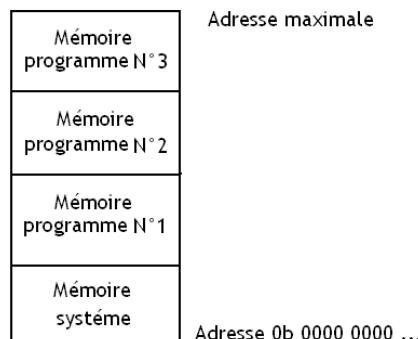
Ces détails concernant l'organisation et la gestion de la mémoire sont assez compliqués à gérer, et faire en sorte que les programmes applicatifs n'aient pas à s'en soucier est un plus vraiment appréciable. Ce genre de problèmes a eu des solutions purement logicielles, mais quelques techniques règlent ces problèmes directement au niveau du matériel : on parle de **mémoire virtuelle**. Avec la mémoire virtuelle, tout se passe comme si notre programme était seul au monde et pouvait lire et écrire à toutes les adresses disponibles à partir de l'adresse zéro. Chaque programme a accès à autant d'adresses que ce que le processeur peut adresser : on se moque du fait que des adresses soient réservées aux périphériques, de la quantité de mémoire réellement installée sur l'ordinateur, ou de la mémoire prise par d'autres programmes en cours d'exécution. Pour éviter que ce surplus de fausse mémoire pose problème, on utilise une partie des mémoires de masse (disques durs) d'un ordinateur en remplacement de la mémoire physique manquante.



Bien sûr, les adresses de cette fausse mémoire vue par le programme sont des adresses fictives, qui devront être traduites en adresses mémoires réelles pour être utilisées. Les fausses adresses sont ce qu'on appelle des adresses logiques, alors que les adresses réelles sont appelées adresses physiques. Pour implémenter cette technique, il faut rajouter un circuit qui traduit les adresses logiques en adresses physiques : ce circuit est appelé la **memory management unit**. Il faut préciser qu'il existe différentes méthodes pour gérer ces adresses logiques et les transformer en adresses physiques : les principales sont la segmentation et la pagination. La suite du chapitre va détailler ces deux techniques.

Partitions mémoire

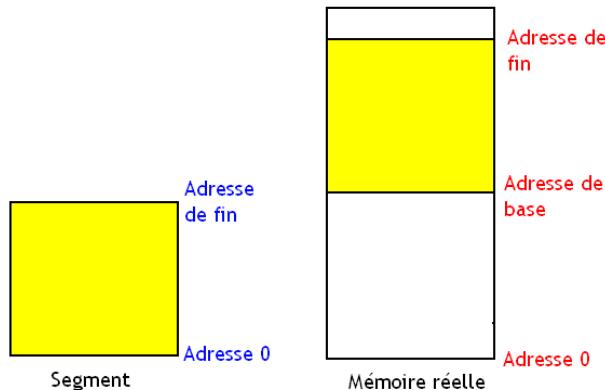
Sur les vieux systèmes d'exploitation, chaque programme recevait un bloc de RAM pour lui tout seul, une **partition mémoire**. Ces partitions n'ont pas une taille fixe, ce qui fait qu'un processus peut demander du rab ou libérer la mémoire qui lui est inutile, l'OS fournissant des appels système pour.



Relocalisation

Chaque partition/segment peut être placé n'importe où en mémoire physique par l'OS, ce qui fait qu'il n'a pas d'adresse fixée en mémoire physique et peut être déplacé comme bon nous semble. C'est le système d'exploitation qui gère le placement des partitions/segments dans la mémoire physique. Ainsi, les adresses de destination des branchements et les adresses des données ne sont jamais les mêmes. Or, chaque accès à une donnée ou instruction demande de connaître son adresse, qui varie à chaque exécution. Il nous faut donc trouver un moyen pour faire en sorte que nos programmes puissent fonctionner avec des adresses qui changent d'une exécution à l'autre. Ce besoin est appelé la **relocalisation**. Pour

résoudre ce problème, le compilateur considère que le programme commence à l'adresse zéro et laisse l'OS corriger les adresses du programme. Cette correction d'adresse demande simplement d'ajouter un décalage, égal à la première adresse de la partition mémoire. Cette correction peut se faire de deux manières : soit l'OS corrige chaque adresse lors du lancement du programme, soit le processeur s'en charge à chaque accès mémoire. Dans le second cas, cette première adresse est mémorisée dans un **registre de base**, mis à jour automatiquement lors de chaque changement de programme.



Protection mémoire

L'utilisation de partitions mémoire entraîne l'apparition de plusieurs problèmes. La première est qu'un programme ne doit avoir accès qu'à la partition qui lui est dédiée et pas aux autres, sauf dans quelques rares exceptions. Toute tentative d'accès à une autre partition doit déclencher une exception matérielle, qui entraîne souvent l'apparition d'un message d'erreur. Tout cela est pris en charge par le système d'exploitation, par l'intermédiaire de mécanismes de **protection mémoire**, que nous allons maintenant aborder.

Pour commencer, le processeur (ou l'OS) doivent détecter les accès hors-partition, à savoir un programme qui lit/écrit de la mémoire au-delà de la partition qui lui est réservée. Pour cela, le processeur incorpore un **registre limite** pour chaque partition, afin de mémoriser l'adresse à laquelle elle se termine. Une implémentation naïve de la protection mémoire consiste à vérifier pour chaque accès mémoire si l'adresse dépasse la valeur du registre limite.

Vient ensuite la **gestion des droits d'accès** : chaque partition/segment se voit attribuer un certain nombre d'autorisations d'accès qui indiquent si l'on peut lire ou écrire dedans, si celui-ci contient un programme exécutable, etc. Lorsqu'on exécute une opération interdite, la MMU déclenche une exception matérielle. Pour cela, l'OS ou la MMU doivent retenir les autorisations pour chaque segment. Ces autorisations et les autres informations (registre de base et limite) sont rassemblées dans un **descripteur de segment**. Quand on accède à un segment, son descripteur est chargé dans des registres du processeur. Pour se simplifier la tâche, les concepteurs de processeurs et de systèmes d'exploitation ont décidé de regrouper ces descripteurs dans une portion de la mémoire, spécialement réservée pour l'occasion : la **table des descripteurs de segment**.

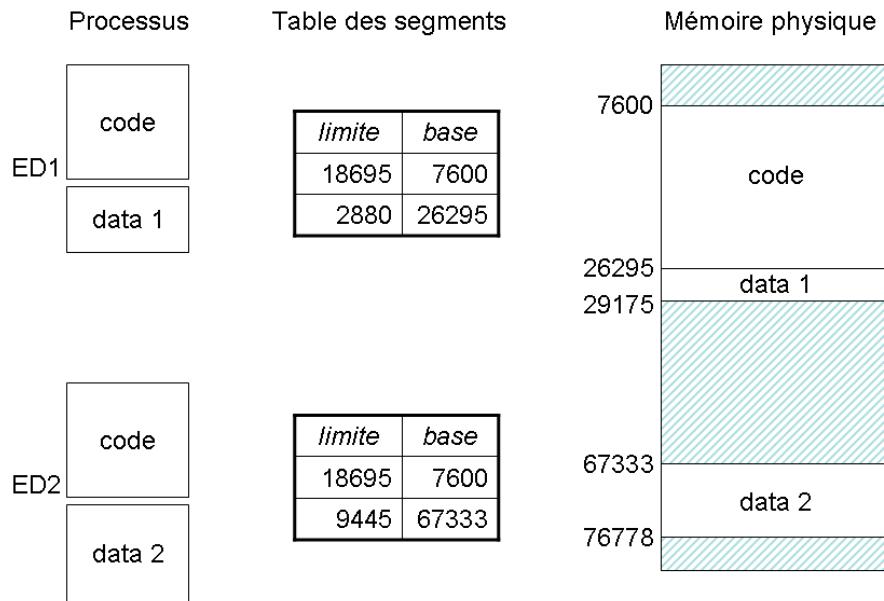
Gestion des partitions

Lorsqu'on démarre un programme, l'OS doit trouver une partition mémoire vide pour accueillir le programme démarré. Pour cela, l'OS garde la **liste des partitions vides**, soit dans un tableau, soit dans une liste chaînée. La première solution découpe la mémoire en blocs de quelques kibioctets, et utilise un bit pour savoir si ce bloc est occupé ou vide, l'ensemble étant mémorisé dans un tableau de bits. Les deux méthodes ont des avantages et inconvénients différents, l'un occupant plus de mémoire et l'autre permettant une recherche plus rapide.

A partir de ces informations, l'OS doit trouver une partition vide suffisamment grande pour accueillir le programme démarré. Si le programme n'utilise pas tout le segment, on découpe la partition de manière à créer une partition pour la zone non-occupée par le programme lancé. Mais si on choisit mal la partition, de multiples allocations entraînent un phénomène de **fragmentation externe** : on dispose de suffisamment de mémoire libre, mais celle-ci est dispersée dans beaucoup de segments qui sont trop petits. Si cela arrive, l'OS doit compacter les partitions dans la mémoire RAM, ce qui prend beaucoup de temps inutilement.

La solution la plus simple est de placer le programme dans la première partition trouvée suffisamment grande (algorithme **First-fit**). Une autre méthode consiste à prendre la partition vide adéquate la plus petite possible, celle qui contient juste ce qu'il faut de mémoire (algorithme **Best-fit**). Et enfin, il est aussi possible de prendre la partition la plus grande (algorithme **Worst-fit**). Les deux dernières méthodes demandent de parcourir toute la liste avant de faire un choix, ce qui rend la recherche plus longue. On peut rendre la recherche plus rapide en utilisant une liste de trous triée par taille. Le second algorithme (plus petite partition) a tendance à agraver la fragmentation externe, contrairement au dernier.

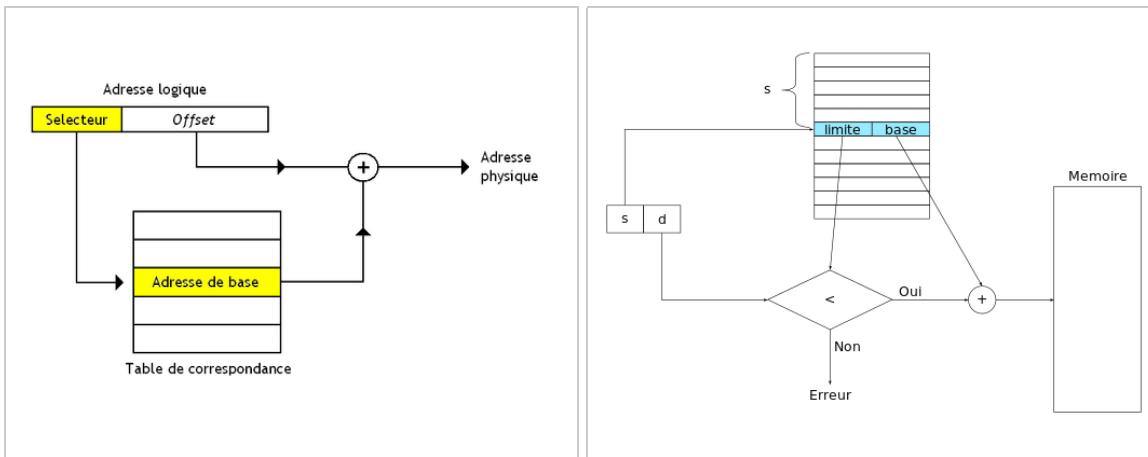
Il faut préciser qu'il est possible de partager des segments entre applications. Il suffit de configurer les tables de segment convenablement. Cela peut servir quand plusieurs instances d'une même application sont lancées simultanément : le code n'ayant pas de raison de changer, celui-ci est partagé entre toutes les instances.



Segmentation

La segmentation est une amélioration de la technique des partitions mémoire, qui permet à un seul processus d'avoir plusieurs espaces d'adresses, plusieurs mémoires virtuelles à lui tout seul. L'utilité est qu'un programme est rarement un tout unique, mais peut souvent être décomposé en plusieurs ensembles de données/instructions séparés, qui peuvent grossir ou se réduire suivant les circonstances. Par exemple, la pile a une taille qui varie beaucoup, tandis que le code du programme ne change pas. La segmentation permet de découper la mémoire virtuelle d'un processus en sous-partitions de taille variable qu'on appelle improprement des **segments**. Ainsi, un programme peut utiliser des segments différents pour la pile, le tas, le programme lui-même, et les variables globales. Chaque segment pourra grandir ou diminuer à sa guise, suivant les besoins, ce qui serait beaucoup plus difficile avec une seule partition mémoire.

Sur les processeurs x86, la mémoire virtuelle est découpée en 2^{16} segments de 2^{32} bits, ce qui donne des adresses de 48 bits. Pour identifier un segment, seuls les 16 bits de poids fort de l'adresse 48 bits sont utiles : ils forment ce qu'on appelle un sélecteur de segment. Les 32 bits servent à identifier la position de la donnée dans un segment : ils sont appelés le décalage (offset). La relocalisation se fait de la même manière pour un segment que pour une partition mémoire. Cependant, un simple registre de base ne suffit plus, vu qu'il y a plusieurs adresses de base à conserver par processus (une par segment) et non une seule. La seule manière de faire cela est d'utiliser une table de correspondance, qui associe chaque segment à son adresse de base (et son adresse de fin si possible). Celle-ci est appelée la **table des segments**. Il faut préciser que cette table de correspondances est unique pour chaque programme : la même adresse logique ne donnera pas la même adresse physique selon le programme.



Traduction d'adresse.

Traduction d'adresse avec vérification des accès hors-segment.

Dans le cas le plus simple, cette table de correspondances est stockée en mémoire RAM, le processeur ne contenant qu'un registre de base mis à jour à chaque changement de segment. Une autre solution consiste à charger la table de correspondance complète dans un banc de registre dédié.

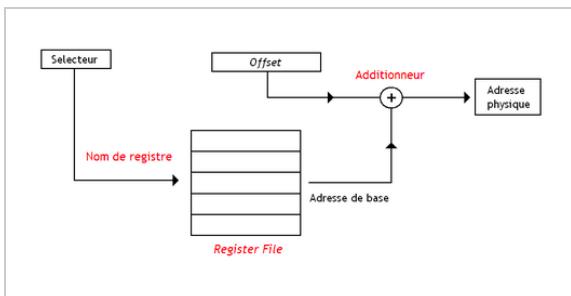
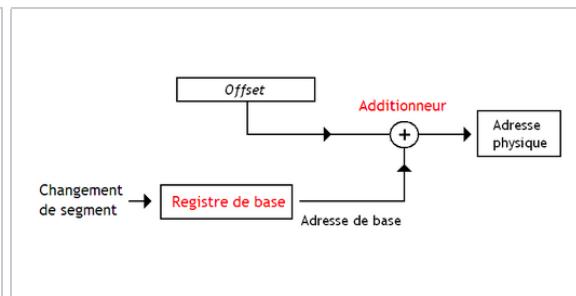


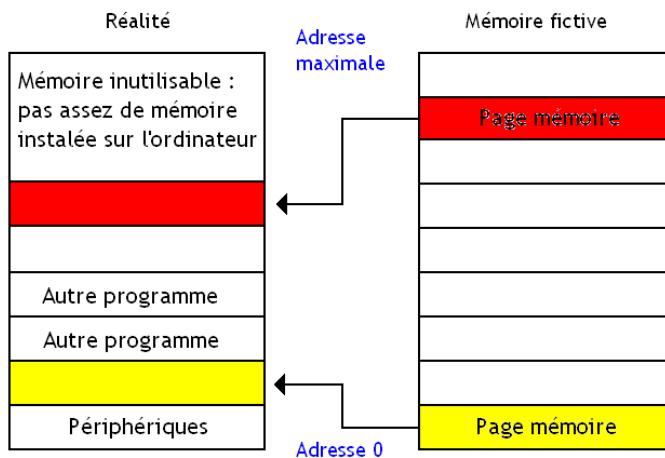
Table des segments dans un banc de registres.



Registre de base de segment.

Pagination

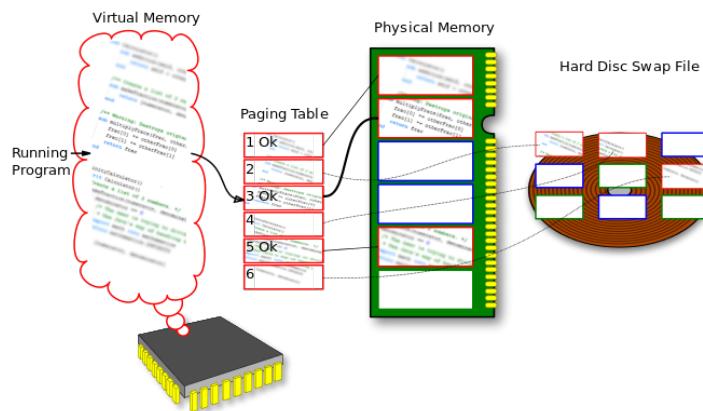
De nos jours, la segmentation est obsolète et n'est plus utilisée : à la place, les OS et processeurs utilisent la pagination. Avec la pagination, la mémoire virtuelle et la mémoire physique sont découpées en blocs de taille fixe (contrairement aux segments de taille variable) : ces blocs sont appelés des **pages mémoires**. La taille des pages varie suivant le processeur et le système d'exploitation, et tourne souvent autour de 4 kibioctets. Le contenu d'une page en mémoire fictive est rigoureusement le même que le contenu de la page correspondante en mémoire physique. Cependant, le nombre total de pages en mémoire virtuelle peut dépasser celui réellement présent en mémoire physique. Le surnombre est simplement placé sur le disque dur, dans un fichier appelé le **fichier d'échange**. Les pages virtuelles vont ainsi faire référence soit à une page en mémoire physique, soit à une page sur le disque dur. Tout accès à une page sur le disque dur va charger celle-ci dans la mémoire RAM, dans une page vide. Les pages font ainsi une sorte de va et vient entre le fichier d'échange et la RAM, suivant les besoins.



La protection mémoire est garantie avec des **clés de protection**, un nombre unique à chaque programme. Le processeur mémorise, pour chaque page, la clé de protection du programme qui a réservé ce bloc. A chaque accès mémoire, le processeur compare la clé de protection du programme en cours d'exécution, et celle de la page adressée. Si les deux clés sont différentes, alors un programme a effectué un accès hors des clous, et il se fait sauvagement arrêter. De plus, chaque page a des droits d'accès précis, qui permettent d'autoriser ou interdire les accès en lecture, écriture, exécution, etc.

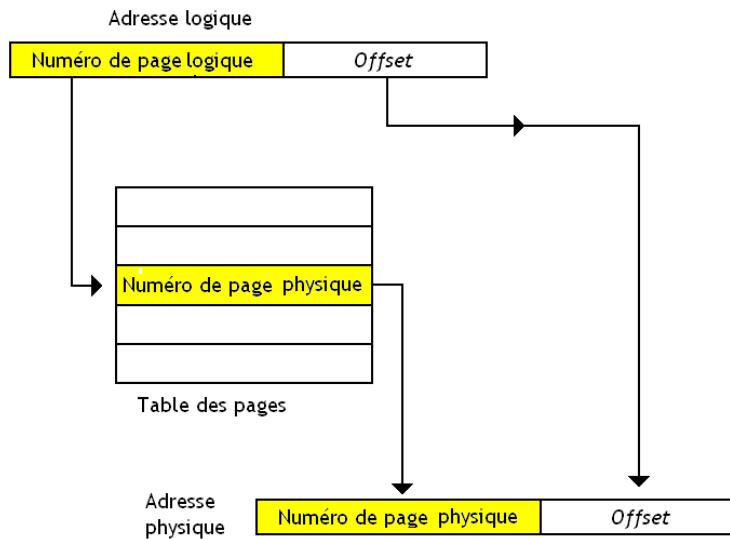
Traduction d'adresse

Une adresse (logique ou physique) se décompose donc en deux parties : un numéro de page qui identifie la page, et une autre permettant de localiser la donnée dans la page. Traduire l'adresse logique en adresse physique demande juste de remplacer le numéro de la page logique en un numéro de page physique. Pour faire cette traduction, il faut se souvenir des correspondances entre numéro de page et adresse de la page en mémoire fictive. Ces correspondances sont stockées dans une sorte de table, nommée la **table des pages**. Celle-ci contient aussi des bits pour savoir si une page est swappée sur le disque dur ou si elle est présente en RAM.

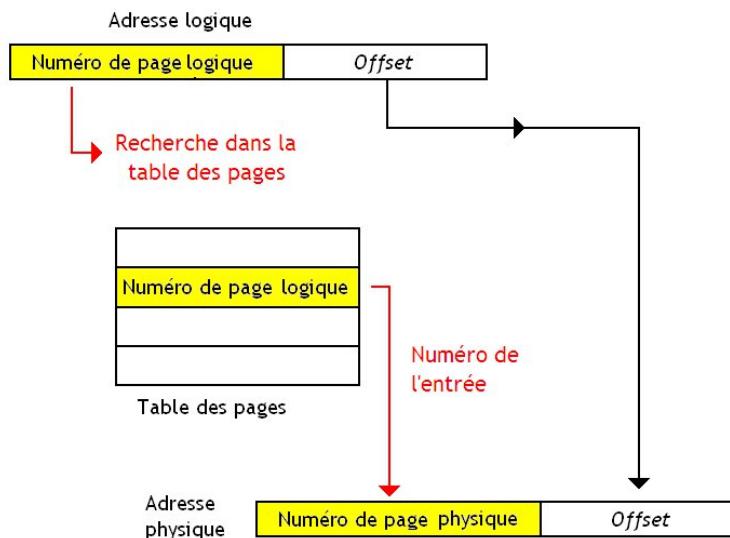


Dans le cas le plus simple, il n'y a qu'une seule table des pages, qui est adressée par les numéros de page logique. Ainsi, pour chaque numéro (ou

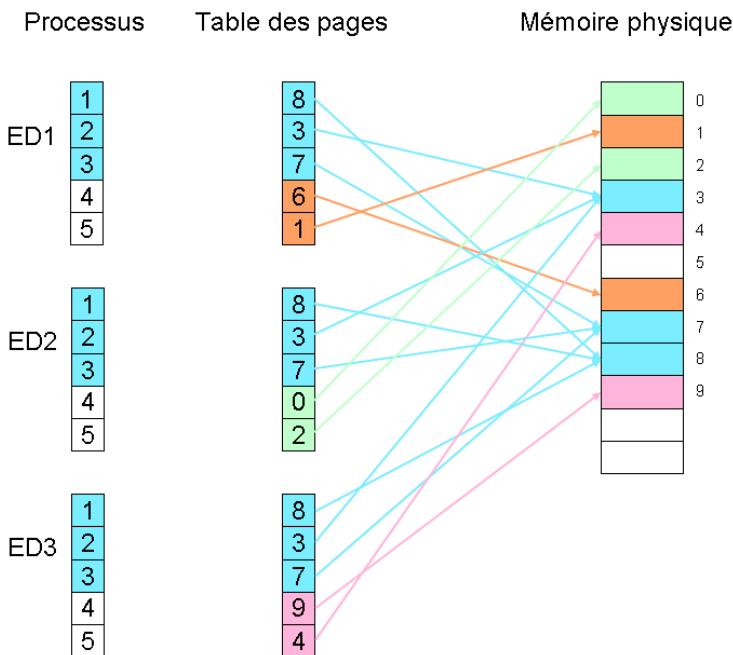
chaque adresse) de page logique, on stocke l'adresse de base de la page correspondante en mémoire physique. La table des pages est unique pour chaque programme, vu que les correspondances entre adresses physiques et logiques ne sont pas les mêmes. Cette table des pages est souvent stockée dans la mémoire RAM, à un endroit bien précis, connu du processeur. Accéder à la mémoire nécessite donc d'accéder d'abord à la table des pages en mémoire, puis de calculer l'adresse de notre donnée, et enfin d'accéder à la donnée voulue.



Sur certains systèmes, la taille d'une table des pages serait beaucoup trop grande en utilisant les techniques vues au-dessus. Pour éviter cela, on a inventé les tables des pages inversées. Elles se basent sur le fait que la mémoire physique réellement présente sur l'ordinateur est souvent plus petite que la mémoire virtuelle. Quand le processeur voudra convertir une adresse virtuelle en adresse physique, la MMU prendra le numéro de page de l'adresse virtuelle, et le recherchera dans la table des pages : le numéro de l'entrée à laquelle se trouve ce morceau d'adresse virtuelle est le morceau de l'adresse physique. Pour faciliter le processus de recherche dans la page, les concepteurs de systèmes d'exploitation peuvent stocker celle-ci avec ce que l'on appelle une table de hachage.

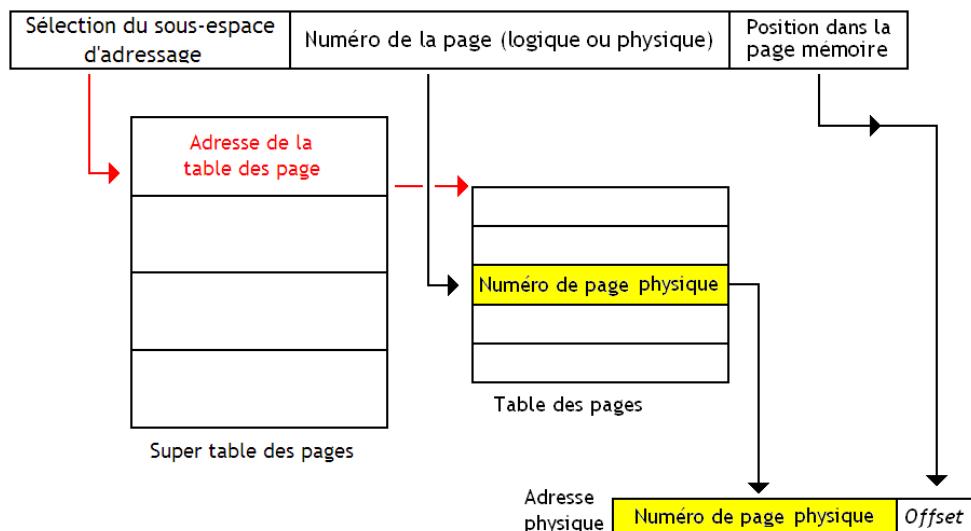


Dans les deux cas précédents, il y a une table des page par processus/programme lancé sur l'ordinateur.



Mais cela a un léger défaut : la table des pages de chaque processus bouffe beaucoup de mémoire. Pour éviter cela, les concepteurs de processeurs et de systèmes d'exploitation ont adapté les tables des pages précédentes pour limiter la casse. Ils ont remarqué que les adresses les plus hautes et/ou les plus basses sont les plus utilisées, alors que les adresses situées au milieu de l'espace d'adressage sont peu utilisées en raison du fonctionnement de la pile et du tas. Ainsi, les concepteurs d'OS ont décidé de découper l'espace d'adressage en plusieurs sous-espaces d'adressage de taille identique : certains espaces d'adresses sont localisés dans les adresses basses, d'autres au milieu, d'autres tout en haut, etc. Chaque sous-espace d'adressage se voit attribuer sa propre page des tables. L'avantage, c'est que si un sous-espace d'adressage n'est pas utilisé, il n'y a pas besoin d'utiliser de la mémoire pour stocker la table des pages associée : on ne stocke que les tables des pages pour les espaces d'adressage qui contiennent au moins une donnée.

Les premiers bits de l'adresse vont servir à sélectionner quelle table des pages utiliser. Pour cela, il faut connaître l'adresse à laquelle est stockée chaque table des pages. Pour cela, on utilise une super-table des pages, qui stocke les adresses de début des tables des pages de chaque sous-espace. On peut aussi aller plus loin et découper la table des pages de manière hiérarchique, chaque sous-espace d'adressage étant lui aussi découpé en sous-espaces d'adresses.

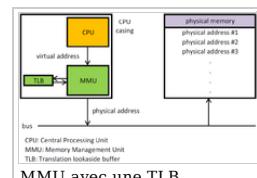


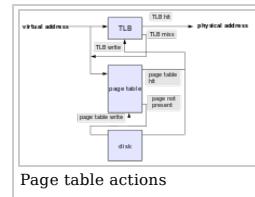
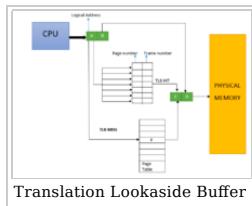
Translation Lookaside Buffer

Pour éviter d'avoir à lire la table des pages en mémoire RAM à chaque accès mémoire, les concepteurs de processeurs ont décidé d'implanter un cache qui stocke les entrées de la page des tables les plus récemment accédées. Ce cache s'appelle le **translation lookaside buffer**, ou TLB. À chaque accès mémoire, le processeur vérifie si le TLB contient l'adresse physique à laquelle accéder. Si c'est le cas, le processeur n'a pas à accéder à la mémoire RAM et va lire directement la donnée depuis ce TLB. L'accès à la RAM est inévitable dans le cas contraire.

Cet accès est géré de deux façons : soit le processeur gère tout seul la situation, soit il délègue cette tâche au système d'exploitation. Dans le premier cas, le processeur est conçu pour lire lui-même le contenu de la page des tables en mémoire et y chercher la bonne correspondance dans celle-ci. Si l'adresse cherchée n'est pas dans le TLB, le processeur va lever une exception matérielle qui exécutera une routine d'interruption chargée de gérer la situation. Pour des raisons de performances, ce cache est parfois découpé en plusieurs sous-caches L1, L2, L3, etc.

Sur les architectures Harvard, on trouve parfois deux TLB séparés : un pour les accès aux instructions, et un pour les accès aux données.





Remplacement des pages mémoires

Vous aurez remarqué que notre mémoire physique contient moins de pages que la mémoire fictive, et il faut trouver un moyen pour que cela ne pose pas de problème. La solution consiste à utiliser des mémoires de stockage comme mémoire d'appoint : si on a besoin de plus de pages mémoires que la mémoire physique n'en contient, certaines pages mémoires vont être déplacées sur le disque dur pour faire de la place. Les pages localisées sur le disque dur doivent être chargées en RAM, avant d'être utilisables. Lorsque l'on veut traduire l'adresse logique d'une page mémoire déplacée sur le disque dur, la MMU ne va pas pouvoir associer l'adresse logique à une adresse en mémoire RAM. Elle va alors lever une exception matérielle dont la routine rapatriera la page en mémoire RAM.

Charger une page en RAM ne pose aucun problème tant qu'il existe de la RAM disponible : on peut charger la donnée dans une page inoccupée. Mais si toute la RAM est pleine, il faut déplacer une page mémoire en RAM sur le disque dur pour faire de la place. Tout cela est effectué par la fameuse routine du système d'exploitation dont j'ai parlé plus haut. Il existe différents algorithmes qui permettent de décider quelle page supprimer de la RAM. Ces algorithmes ont une importance capitale en termes de performances : si on supprime une donnée dont on aura besoin dans le futur, il faudra recharger celle-ci, ce qui prend du temps. Pour éviter cela, le choix de la page doit être fait avec le plus grand soin. Ces algorithmes sont les suivants.

- Aléatoire : on choisit la page au hasard.
- FIFO : on supprime la donnée qui a été chargée dans la mémoire avant toute les autres.
- LRU : on supprime la donnée qui a été lue ou écrite pour la dernière fois avant toute les autres.
- LFU : on vire la page qui est lue ou écrite le moins souvent comparée aux autres.
- etc.

Ces algorithmes ont chacun deux variantes : une locale, et une globale. Avec la version locale, la page qui va être rapatriée sur le disque dur est une page réservée au programme qui est la cause du page miss. Avec la version globale, le système d'exploitation va choisir la page à virer parmi toutes les pages présentes en mémoire vive.

Sur la majorité des systèmes d'exploitation, il est possible d'interdire le déplacement de certaines pages sur le disque dur. Ces pages restent alors en mémoire RAM durant un temps plus ou moins long, parfois en permanence. Cette possibilité simplifie la vie des programmeurs qui conçoivent des systèmes d'exploitation : essayez d'exécuter une interruption de gestion de page miss alors que la page contenant le code de l'interruption est placée sur le disque dur.

Les mémoires cache

Le cache est une mémoire intercalée entre la mémoire et un processeur ou un périphérique, qui est souvent fabriquée avec de la mémoire SRAM, parfois avec de l'eDRAM. Sans lui, on se croirait à l'âge de pierre tellement nos PC seraient lents ! En effet, la mémoire est lente comparée au processeur. Or, le temps mis pour accéder à la mémoire est du temps durant lequel le processeur n'exécute pas d'instruction (sauf cas particuliers impliquant un pipeline). Il a fallu trouver une solution pour diminuer ce temps d'attente, et on a décidé d'intercaler une mémoire entre le processeur et la mémoire. Comme cela, le processeur accède directement à une mémoire cache très rapide plutôt que d'accéder à une mémoire RAM qui répondra de toute façon trop tard.

Accès au cache

Le cache est divisé en groupes de plusieurs bytes (de 64 à 256 octets chacun), qui portent le nom de **lignes de cache**. Sur les caches actuels, on transfère les données entre le cache et la RAM ligne de cache par ligne de cache. Un cache est une mémoire associative : tout accès mémoire sera intercepté par le cache, qui vérifiera si la donnée demandée est présente ou non dans le cache. Si c'est le cas, la donnée voulue est présente dans le cache : on a un **succès de cache** (cache hit) et on accède à la donnée depuis le cache. Sinon, c'est un **défaut de cache** (cache miss) : on est obligé d'accéder à la RAM ou de recopier notre donnée de la RAM dans le cache. Le nombre de succès de cache par nombre d'accès mémoire, appelé le **taux de succès** (hit ratio), est déterminant pour les performances. En effet, plus celui-ci est élevé, plus on accède au cache à la place de la RAM et plus le cache est efficace.

Lorsqu'on exécute une instruction ou qu'on accède à une donnée pour la première fois, celle-ci (l'instruction ou la donnée) n'a pas encore été chargée dans le cache. Le défaut de cache est inévitable : ce genre de défaut de cache s'appelle un **défaut à froid** (cold miss). Si on accède à beaucoup de données, le cache finit par être trop petit pour conserver les anciennes données : elles vont quitter le cache, et toute tentative ultérieure d'accès tombera en RAM, donnant un **défaut de volume de cache** (capacity cache miss). Les seules solutions pour éviter cela consistent à augmenter la taille du cache, faire en sorte que le programme prenne moins de mémoire cache, et améliorer la localité du programme exécuté.

Tag d'une ligne de cache

Les données présentes dans le cache sont été (pré)chargées depuis la mémoire : toute donnée présente dans le cache est la copie d'une donnée en mémoire RAM. Pour faire la correspondance entre une ligne de cache et l'adresse mémoire correspondante, on ajoute des bits supplémentaires à chaque ligne de cache, qui contiennent une partie (voir la totalité) des bits de l'adresse mémoire correspondante. Ces bits supplémentaires forment ce qu'on appelle le **tag**. Quand notre cache reçoit une demande de lecture ou écriture, il va comparer le tag de chaque ligne avec les bits de poids fort de l'adresse à lire ou écrire. Si une ligne contient ce tag, alors c'est que cette ligne correspond à l'adresse, et c'est un défaut de cache sinon. Cela demande de comparer le tag avec un nombre de lignes de cache qui peut être conséquent, et qui varie suivant l'organisation du cache.

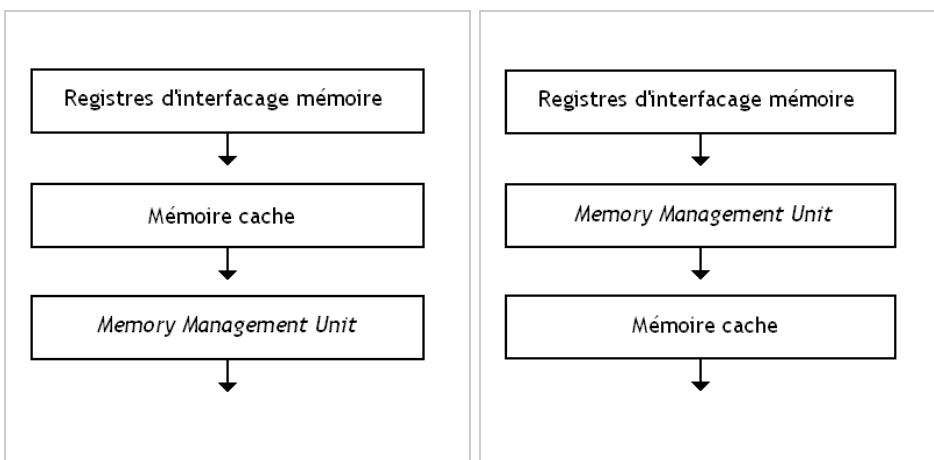
Tag	Données	Bits de contrôle
-----	---------	------------------

Sur certains caches assez anciens, on pouvait transférer nos lignes de caches morceaux par morceaux. Ces caches avaient des lignes de cache divisées en sous-secteurs, ces sous-secteurs étant des morceaux de ligne de cache qu'on pouvait charger indépendamment les uns des autres (mais qui sont consécutifs en RAM). Chaque secteur avait ses propres bits de contrôle, mais le tag était commun à tous les secteurs.

Tag	Données	Bits de contrôle	Données	Bits de contrôle	Données	Bits de contrôle
Secteur						

Adresses physiques ou logiques ?

L'interaction entre caches et mémoire virtuelle donne lieu à un petit problème : l'adresse utilisée lors de l'accès au cache est-elle une adresse physique ou virtuelle ? Et bien cela varie suivant le processeur : certains caches utilisent une correspondance entre une adresse virtuelle et une ligne de cache, tandis que d'autres l'effectuent avec l'adresse physique. Notre cache peut donc être placé aussi bien avant qu'après la MMU. On parle de cache **virtuellement tagué** dans le premier cas et de cache **physiquement tagué** dans le second. Un cache virtuellement tagué n'a pas besoin d'attendre que la MMU ait fini de traduire l'adresse logique en adresse physique pour vérifier la présence de la donnée dans le cache : ces caches sont donc plus rapides. Mais les problèmes arrivent facilement quand on utilise plusieurs programmes : une adresse logique correspond à des adresses physiques différentes suivant le programme. Pour éviter toute confusion, on peut rajouter des bits de contrôle pour identifier le programme qui possède la ligne de cache. On peut aussi vider le cache en changeant de programme. Les caches physiquement tagués ont les avantages et inconvénients inverses : moins rapides, ils permettent un partage du cache entre plusieurs programmes aisément.



Cache tagué virtuellement.

Cache tagué physiquement.

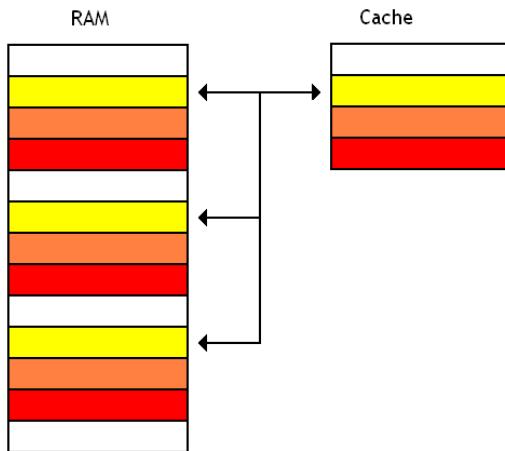
Vérification des tags

Lorsqu'on souhaite accéder au cache, il faut trouver quelle est la ligne de cache dont le tag correspond à l'adresse demandée. On peut classifier les caches selon leur stratégie de recherche de la ligne correspondante en quatre types de caches :

- directement adressés, ou direct mapped ;
- totalement associatifs ;
- associatifs par voie ;
- pseudo-associatifs.

Les caches directement adressés

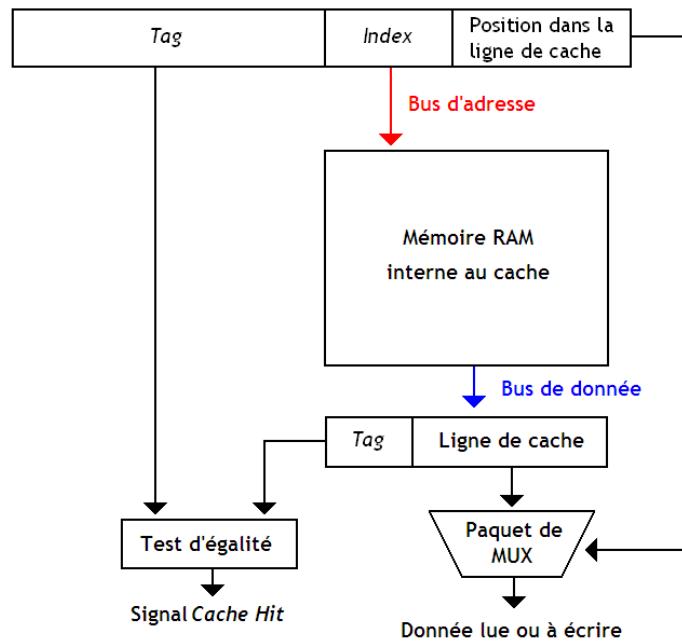
Avec ce genre de caches, le contenu d'une adresse mémoire sera chargée dans une ligne de cache prédéfinie, toujours la même, quelles que soient les circonstances. L'accès au cache a donc l'avantage d'être très rapide vu qu'il suffit de vérifier une seule ligne de cache : celle prédéfinie. Mais ces caches ne sont cependant pas sans défauts. Vu que le cache est plus petit que la mémoire, certaines adresses mémoires devront se partager la même ligne de cache. Si on a besoin de manipuler fréquemment des données qui se partagent la même ligne de cache, chaque accès à une donnée supprimera l'autre du cache : tout accès à l'ancienne donnée se soldera par un défaut de cache. Ce genre de défauts de cache causés par le fait que deux adresses mémoires ne peuvent utiliser la même ligne de cache s'appelle un **défaut par conflit** (conflict miss). Ainsi, le taux de succès de ce genre de cache est quelque peu... comique, ce qui réduit les gains de performances gagnés de part leur faible temps d'accès.



Les concepteurs de caches s'arrangent pour que des adresses consécutives en mémoire RAM occupent des lignes de cache consécutives, afin de simplifier les circuits de gestion du cache. Chaque ligne de cache possède un index, une sorte d'adresse interne qui permet de l'identifier et la sélectionner parmi toutes les autres lignes. Il ne s'agit pas d'une adresse, vu que le cache n'est pas adressable via le bus d'adresse. Avec cette implémentation, l'adresse mémoire doit permettre de spécifier l'index de la donnée. Le tag correspond aux bits de poids fort de l'adresse mémoire correspondant à notre ligne de cache. Le reste des bits de l'adresse représente l'index de notre ligne, et la position de la donnée dans la ligne.

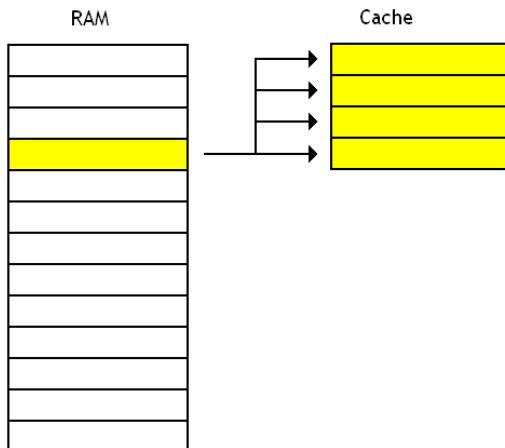
<i>Tag</i>	<i>Index</i>	<i>Position dans la ligne de cache</i>
Adresse mémoire		

Un cache directement adressé est conçu avec une RAM, un comparateur, et un paquet de multiplexeurs. La mémoire RAM stocke les lignes de caches et les tags. Un mot mémoire de cette RAM contient une ligne de cache, avec son tag (parfois, on utilise une mémoire séparée pour les tags). Chaque ligne étant sélectionnée par son index, on devine aisément que l'index de notre ligne de cache sera envoyée sur le bus d'adresse de notre mémoire RAM pour sélectionner celle-ci. Ensuite, il suffira de comparer le tag récupéré avec le tag de l'adresse à lire ou écrire. On saura alors si on doit faire face à un défaut de cache. Ensuite, on devra sélectionner la bonne donnée dans notre ligne de cache avec un ensemble de multiplexeurs.

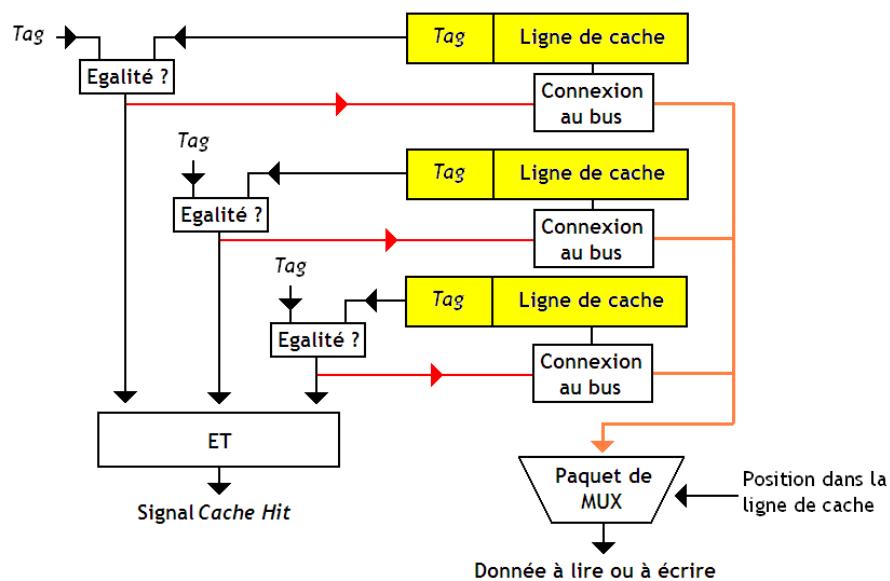


Les caches totalement associatifs

Avec les caches totalement associatifs, toute donnée chargée depuis la mémoire peut être placée dans n'importe quelle ligne de cache, sans aucune restriction. Ces caches ont un taux de succès très élevé, vu qu'il n'y a pas de possibilité de défaut par conflit.

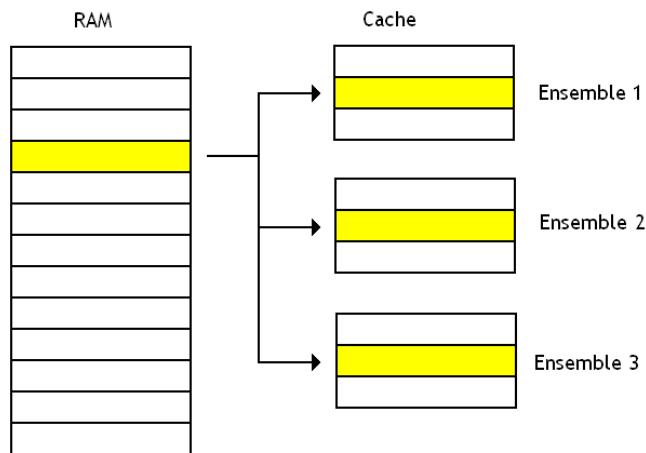


Une adresse mémoire ne peut pas servir à identifier une ligne en particulier : elle est donc découpée en un tag, et de quoi identifier la position de la donnée dans la ligne de cache correspondante. Pour déterminer l'occurrence d'un défaut de cache, il suffit de comparer le tag de l'adresse avec tout les tags présents dans le cache : si il y a une seule égalité, pas de défaut de cache. Quelques comparateurs (un par ligne de cache), et un arbre de portes ET suffit. Toutes nos lignes de caches sont reliées à un bus interne qui permettra de relier chaque ligne de cache à l'extérieur. Si les deux tags sont identiques, la ligne de cache associée est la bonne, et elle doit être connectée sur le bus : on relie donc la sortie du comparateur à des transistors chargés de connecter ou de connecter les lignes de cache sur notre bus. Ensuite, il ne reste plus qu'à sélectionner la portion de la ligne de cache qui nous intéresse, grâce à un paquet de multiplexeurs, comme pour les autres caches.

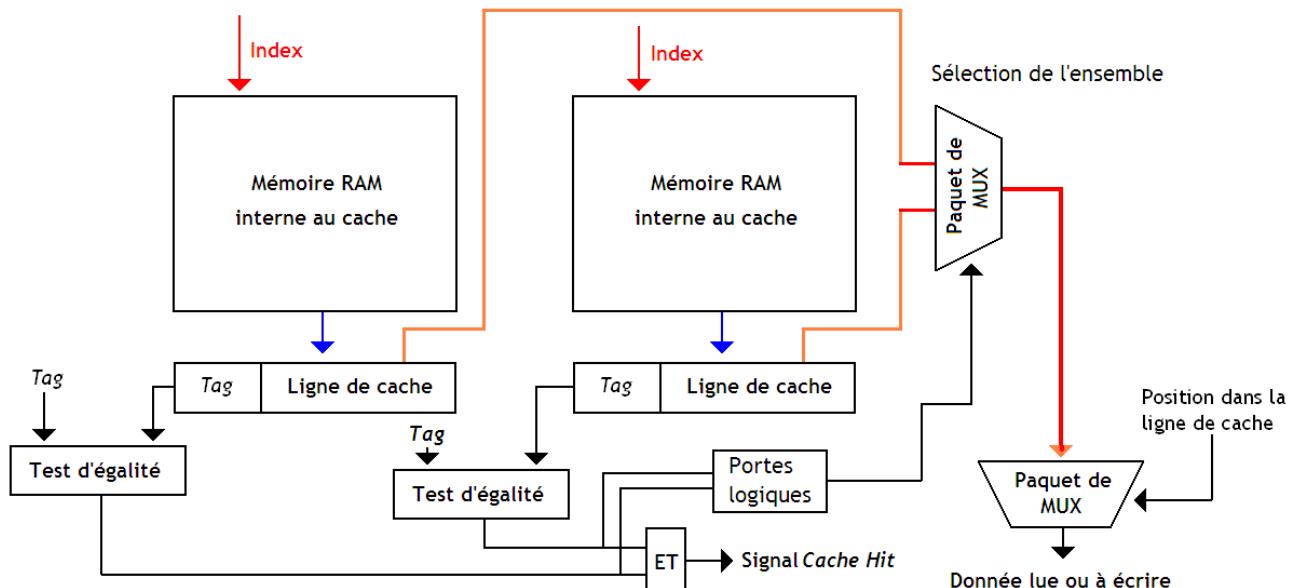


Les caches associatifs par voie

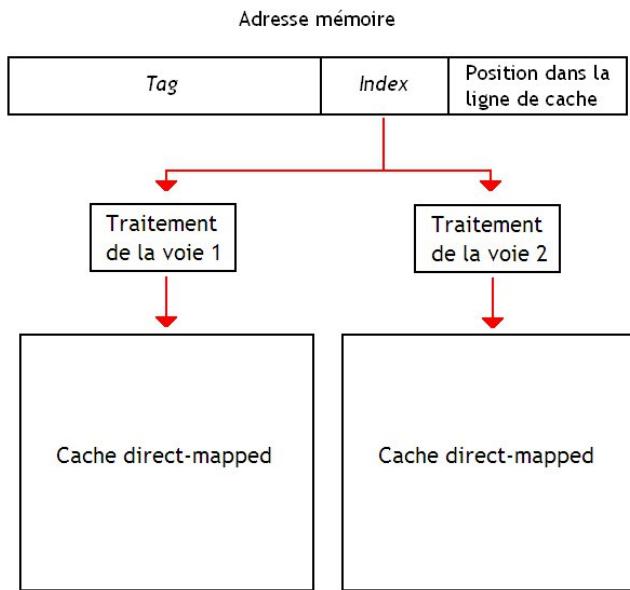
Les caches précédents ont chacun leur défaut : un taux de succès médiocre pour les premiers, et un temps d'accès trop long pour les autres. Certains caches implémentent une sorte de compromis destiné à trouver un juste milieu : ce sont les caches associatifs par voie. Pour simplifier, ces caches sont composés de plusieurs caches directement adressés accessibles en parallèle, chaque cache étant appelé une voie.



L'adresse d'une case mémoire est découpée en trois parties : un tag, un index, et un décalage, comme sur les caches directement adressés. Comme vous pouvez le voir, l'organisation est identique à celle d'un cache totalement associatif, à part que chaque ensemble tag-ligne de cache est remplacé par une mémoire RAM qui en contient plusieurs.



Vous aurez remarqué que dans une voie, les lignes sont accédées en adressage direct : les défauts par conflit sont possibles sur un cache associatif par voie. Pour éviter cela, certains chercheurs ont créé des caches skew associative (ou associatifs à biais). Pour faire simple, les index des lignes de cache subissent un petit traitement avant d'être utilisés. Le traitement en question est différent suivant la voie de destination, histoire que deux adresses mémoires avec des index identiques donnent des index différents après traitement. Le traitement en question est souvent une permutation des bits de l'index, qui est différente suivant la voie prise, ou un simple XOR avec un nombre qui dépend de la voie.



Les caches associatifs par voie sont donc une sorte de compromis entre caches directement adressés et caches totalement associatifs, avec un taux de succès et un temps d'accès intermédiaire. Pour réduire ce temps d'accès, certains chercheurs ont inventé la **prédition de voie**, qui consiste à faire des paris sur la prochaine voie accédée. Au lieu d'attendre que les comparaisons de tags donnent leur résultat, le processeur sélectionne automatiquement une voie et configure les multiplexeurs à l'avance. Si le processeur ne se trompe pas, le processeur va accéder à la donnée de façon précoce, et commencer à l'utiliser un à deux cycles plus tôt que prévu. S'il se trompe, le processeur devra annuler la lecture effectuée en avance, et recommencer en allant chercher la donnée dans le bon ensemble. Cette technique peut être adaptée de façon à diminuer la consommation énergétique du processeur. Pour cela, il suffit de mettre en veille tous les caches directement adressés sur lesquels le processeur n'a pas parié. C'est plus efficace que d'aller lire plusieurs données dans des mémoires différentes, et n'en garder qu'une.

En vertu du principe de localité, on peut décentrement penser que si on a accédé à une voie, les accès futurs auront lieu dans celle-ci. Il suffit de retenir la voie la plus récemment accédée dans un registre, qui sera utilisée comme prédition. Pour vérifier que la prédition est correcte, il suffit de comparer le registre au résultat obtenu après vérification des tags. Cependant, on peut complexifier l'implémentation pour prendre en compte un paramètre assez important : on peut discriminer la voie à choisir en tenant compte de paramètres comme l'adresse à lire/écrire, ou l'instruction à l'origine de l'accès mémoire. En effet, des instructions différentes ont tendance à aller chercher leurs données dans des ensembles différents, et la voie à choisir n'est pas la même. Pour cela, il suffit d'utiliser un cache pour stocker les voies à mémoriser : une par instruction. On peut aussi utiliser le même mécanisme pour faire la différence non pas suivant l'instruction à l'origine de l'accès au cache, mais en fonction de l'adresse à lire, ou des numéros de registre, voire des données utilisées pour calculer l'adresse.

Pour plus de simplicité, la mémoire cache des prédictions est parfois remplacée par une RAM, qui est adressée :

- soit par le program counter de l'instruction à l'origine de l'accès (en réalité, seulement quelques bits de poids faible de l'adresse) ;
- soit par l'adresse à accéder (là encore, quelques bits de poids faible) ;
- soit (pour les modes d'adressage qui utilisent un registre de base et un décalage) par un XOR entre les bits de poids faible de l'adresse de base et le décalage ;
- soit par autre chose.

Caches pseudo-associatifs

Les caches pseudo-associatifs sont identiques aux caches associatifs par voie, si ce n'est qu'ils vérifient chaque voie une par une. Le temps d'accès dans le meilleur des cas est plus faible pour les caches pseudo-associatifs. Dans le pire des cas, on doit tester tous les caches avant de tomber sur le bon.

Remplacement des lignes de cache

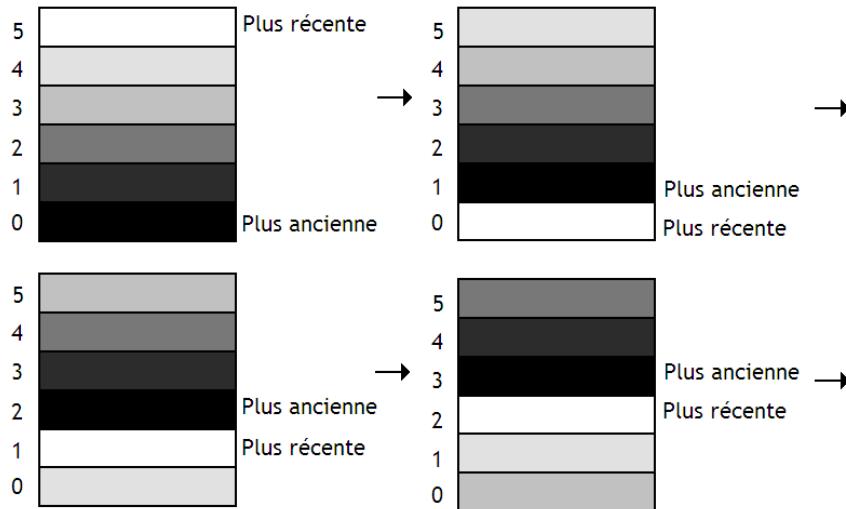
Lorsqu'un cache est rempli et qu'on charge une nouvelle donnée dedans, il faut faire de la place pour cette dernière. Dans le cas d'un cache directement adressé, il n'y rien à faire vu que la ligne de cache à évincer est déterminée lors de la conception du cache. Mais pour les autres caches, la donnée peut aller dans n'importe quelle ligne ou voie. Or, le choix des données à rapatrier en RAM doit être le plus judicieux possible : on doit virer de préférence des données inutiles. Rapatrier une donnée qui sera sûrement utilisée sous peu est inutile, et il vaudrait mieux supprimer des données qui ne serviront plus ou dans alors dans longtemps. Il existe différents algorithmes spécialement dédiés à résoudre ce problème efficacement, directement câblés dans les unités de gestion du cache. Certains sont vraiment très complexes, aussi je vais vous présenter quelques algorithmes particulièrement simples. Mais avant de voir ces algorithmes, il faut absolument que je vous parle d'une chose très importante. Quel que soit l'algorithme en question, il va obligatoirement choisir une ligne de cache à remplacer et recopier son contenu dans la RAM. Difficile de faire la sélection en utilisant nos tags. Pour résoudre ce problème, le circuit de remplacement des lignes de cache va adresser chaque ligne de cache ! Eh oui, vous avez bien vu : chaque ligne de cache sera numérotée par une adresse, interne au cache.

Aléatoire

Premier algorithme : la donnée effacée du cache est choisie au hasard ! Si l'on dispose d'un cache avec n lignes, cet algorithme s'implémente avec un circuit qui fournit un nombre pseudo-aléatoire compris entre 0 et n . Contraintivement, cet algorithme donne des résultats assez honorables, en plus d'utiliser très peu de portes logiques. Reste à implémenter cet algorithme. Pour cela, on peut utiliser un compteur qui s'incrémenté à chaque cycle d'horloge. Généralement, les défauts de cache sont séparés par un nombre assez important et irrégulier de cycles d'horloge. Dans ces conditions, cette technique donne un bon résultat. Mais il est aussi possible d'utiliser des circuits un peu plus élaborés, à savoir des registres à décalage à rétroaction linéaire.

FIFO : first in, first out

Avec l'algorithme FIFO, la donnée effacée du cache est la plus ancienne, celle chargée dans le cache avant les autres. Cet algorithme possède une petite particularité sur les caches associatifs par voie : en augmentant le nombre d'ensembles, les performances peuvent se dégrader : c'est ce qu'on appelle l'**anomalie de Bélády**. Cet algorithme est un des plus simples à implémenter en circuit : un vulgaire compteur suffit. On peut insérer les données dans le cache les unes à la suite des autres. Exemple : si j'ai inséré une donnée dans la ligne de cache numéro X, alors la prochaine donnée ira dans la ligne numéro X+1. Si jamais on déborde, on revient automatiquement à zéro. En faisant ainsi, nos lignes de cache seront triées de la plus ancienne à la plus récente automatiquement. La ligne de cache la plus ancienne sera localisée à un certain endroit, et la plus récente sera localisée juste avant. Il suffit de se souvenir de la localisation de la donnée la plus ancienne avec un compteur par voie, et le tour est joué.

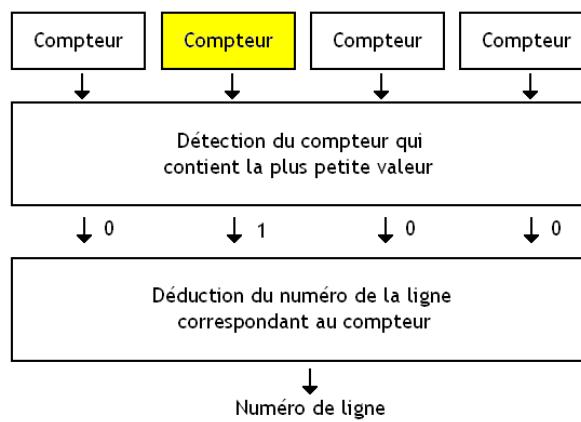


MRU : most recently used

Avec l'algorithme MRU, la donnée qui est effacée est celle qui a été utilisée le plus récemment. Cet algorithme s'implémente simplement avec un registre, dans lequel on place le numéro de la dernière ligne de cache utilisée. Cet algorithme de remplacement est très utile quand un programme traverse des tableaux du premier élément jusqu'au dernier : les données du tableau sont rarement réutilisées, rendant le cache inutile. Il est prouvé que dans ces conditions, l'algorithme MRU est optimal. Mais dans toutes les autres conditions, cet algorithme a des performances assez misérables.

LFU : least frequently used

Avec l'algorithme LFU, la donnée supprimée est celle qui est utilisée le moins fréquemment. Cet algorithme s'implémente en associant un compteur à chaque ligne de cache. À chaque fois qu'on lit ou écrit dans cette ligne de cache, le compteur associé est incrémenté. La ligne la moins récemment utilisée est celle dont le compteur associé a la plus petite valeur. Implémenter cet algorithme prend pas mal de transistors : il faut rajouter autant de compteurs qu'il y a de lignes de cache, en plus d'un circuit pour déduire quel compteur contient la plus petite valeur et en déduire la ligne de cache en question. Le circuit qui détermine quel compteur a la plus petite valeur est composé d'un grand nombre de comparateurs et quelques portes logiques ET. L'autre circuit est un encodeur.



LRU : least recently used

Avec l'algorithme LRU, la donnée remplacée est celle qui a été utilisée le moins récemment. Cet algorithme se base sur le principe de localité temporelle, qui stipule que si une donnée a été accédée récemment, alors elle a de fortes chances d'être réutilisée dans un futur proche. Et inversement, toute donnée peu utilisée récemment a peu de chances d'être réutilisée dans le futur. D'après le principe de localité temporelle, la donnée la moins récemment utilisée du cache est donc celle qui a le plus de chance de ne servir à rien dans le futur. Autant la supprimer en priorité pour faire de la place à des données potentiellement utiles.

Implémenter cet algorithme LRU peut se faire de différentes manières. Dans tous les cas, ces techniques sont basées sur un même principe : les circuits reliés au cache doivent enregistrer les accès au cache pour en déduire la ligne la moins récemment accédée. La première technique demande d'utiliser un compteur pour chaque ligne de mémoire cache, un peu comme le LFU. La différence avec le LFU est que le compteur n'est pas incrémenté lors d'un accès mémoire. À la place, ce compteur est incrémenté régulièrement, chaque incrémantation ayant lieu en même temps pour tous les compteurs. Quand un bloc est chargé dans le cache, ce compteur est mis à zéro. Quand une ligne de cache doit être remplacée, un circuit va vérifier la valeur de tous les compteurs : la ligne LRU (la moins récemment utilisée), est celle dont le compteur a la valeur la plus haute. Le circuit est composé d'un paquet de comparateurs, et d'un encodeur, comme pour l'algorithme LFU.

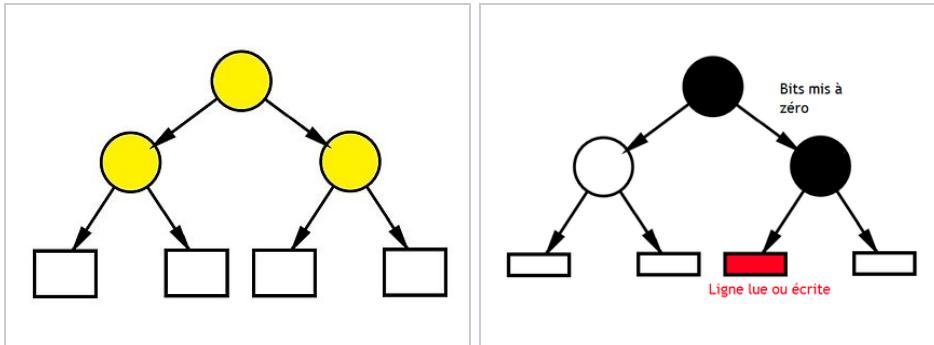
Approximations du LRU

Comme on l'a vu, implémenter le LRU coûte cher en transistors, le nombre de transistors utilisés étant proportionnel au carré du nombre de lignes de cache. Autant dire que le LRU devient impraticable sur de gros caches. Pour résoudre ce problème, nos processeurs implémentent des variantes du LRU, moins coûteuses en transistors, mais qui ne sont pas exactement du LRU : ils donnent un résultat assez semblable au LRU, mais un peu plus approximatif. En clair, ils ne sélectionnent pas toujours la ligne de cache la moins récemment utilisée, mais une ligne de cache parmi les moins récemment utilisées. Il faut dire que les lignes les moins récemment utilisées ont toutes assez peu de chance d'être utilisées dans le futur. Entre choisir de remplacer une ligne qui a 0,5 % de chances d'être utilisée dans le futur et une autre qui a une chance de seulement 1 %, la différence est négligeable : cela aura une influence assez faible en termes de taux de succès. Mais les gains en termes de circuits ou de temps d'accès au cache de ces algorithmes peuvent donner des résultats impressionnantes.

L'algorithme le plus simple consiste à couper le cache (ou chaque ensemble du cache s'il est associatif à voie) en deux. L'algorithme consiste à choisir le morceau de cache le moins récemment utilisé, et de choisir aléatoirement une ligne de cache dans ce morceau. Pour implémenter cet algorithme, il nous suffit d'une simple bascule qui mémorise le morceau le moins récemment utilisé, et d'un circuit qui choisit aléatoirement une ligne de cache dans ce morceau. On peut aussi généraliser cette technique avec plus de deux morceaux de cache : il suffit juste de rajouter quelques circuits. Dans ce cas, cette technique s'adapte particulièrement bien avec des cache associatifs à n voies : il suffit d'utiliser autant de morceaux que de voies, de sous-caches adressés directement. En effet, avec un cache adressé directement, on a juste à savoir quelle est la voie la moins utilisée, sans se préoccuper de la ligne exacte : la ligne dans laquelle écrire dépendra uniquement de l'adresse de la donnée chargée, en raison du caractère directement adressé de la voie.

Autre algorithme, un peu plus efficace : le pseudo-LRU de type m. Cet algorithme est assez simple : à chaque ligne de cache, on attribue un bit. Ce bit sert à indiquer de façon approximative si la ligne de cache associée est une candidate pour un remplacement ou non. Si ce bit est à 1, cela veut dire : attention, cette ligne n'est pas une candidate pour un remplacement. Inversement, si ce bit est à zéro, la ligne peut potentiellement être choisie pour laisser la place aux jeunes. Lorsqu'on lit ou écrit dans une ligne de cache, ce bit est mis à 1. Aussi, l'algorithme permet de remettre les pendules à l'heure. Si tous les bits sont à 1, on les remet tous à zéro, sauf pour la dernière ligne de cache accédée. L'idée derrière cet algorithme est d'encercler la ligne de cache la moins récemment utilisée au fur et à mesure des accès. Tout commence lorsque l'on remet tous les bits associés aux lignes de cache à 0 (sauf pour la ligne accédée en dernier). Puis, au fur et à mesure que nos lignes de cache voient leurs bits passer à un, l'étau se resserre autour de notre ligne de cache la moins utilisée. Et on finit par l'encercler de plus en plus : au final, après quelques accès, l'algorithme donne une estimation particulièrement fiable. Et comme les remplacements de lignes de cache sont rares comparés aux accès aux lignes, cet algorithme finit par donner une bonne estimation avant qu'on ait besoin d'effectuer un remplacement.

Le dernier algorithme d'approximation, le PLURt, se base sur ce qu'on appelle un arbre de décision. Il a besoin de $n - 1$ bits pour déterminer la ligne LRU. Ces bits doivent être organisés en arbre, comme illustré plus bas. Chacun de ces bits sert à dire : le LRU est à ma droite ou à ma gauche : il est à gauche si je veux 0, et à droite si je veux 1. Trouver le LRU se fait en traversant cet arbre, et en interprétant les bits un par un. Au fur et à mesure des lectures, les bits sont mis à jour dans cet arbre, et pointent plus ou moins bien sur le LRU. La mise à jour des bits s'effectue lors des lectures et écritures : quand une ligne est lue ou écrite, elle n'est pas la ligne LRU. Pour l'indiquer, les bits à 1 qui pointent vers la ligne de cache sont mis à 0 lors de la lecture ou écriture.



Organisation des bits avec l'algorithme PLURt.

Ligne de cache pointée par les bits de l'algorithme.

LRU amélioré

L'algorithme LRU, ainsi que ses variantes approximatives, sont très efficaces dans la majorité des programmes. Du moment que notre programme respecte relativement bien la localité temporelle, cet algorithme donnera de très bons résultats : le taux de succès du cache sera très élevé. Par contre, cet algorithme se comporte assez mal dans certaines circonstances, et notamment quand on traverse des tableaux. Dans ces conditions, on n'a pas la moindre localité temporelle, mais une bonne localité spatiale. Pour résoudre ce petit inconvénient, des variantes du LRU existent. Une première de ces variantes, l'algorithme 2Q, utilise deux caches : un cache FIFO pour les données accédées une seule fois et un second cache LRU. Évidemment, les données lues une seconde fois doivent migrer du cache FIFO vers le cache LRU : cela consomme du matériel, de l'énergie et des cycles d'horloge. Les processeurs n'utilisent donc pas cette technique, mais celle-ci est utilisée dans les caches de disque dur. D'autres variantes du LRU combinent plusieurs algorithmes à la fois et vont choisir lequel de ces algorithmes est le plus adapté à la situation. Notre cache pourra ainsi détecter s'il vaut mieux utiliser du MRU, du LRU, ou du LFU suivant la situation.

Write-back et write-through

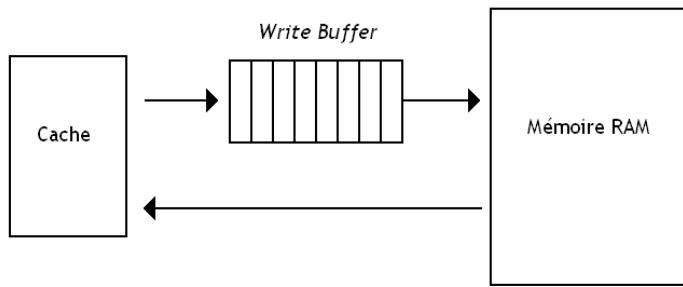
L'écriture dans un cache fait face à diverses situations, qu'il faut gérer au mieux. Pour gérer certaines situations embarrassantes, deux stratégies d'écritures sont couramment implémentées dans les circuits de gestion du cache :

- le write-back ;
- et le write-through.

Write-back

Si la donnée à mettre à jour est présente dans le cache, on écrit dans celui-ci sans écrire dans la mémoire RAM. Dans ces conditions, une donnée n'est enregistrée en mémoire que si celle-ci quitte le cache, ce qui évite de nombreuses écritures mémoires inutiles. On parle alors de caches **Write-Back**.

En suivant la procédure habituelle de remplacement des lignes de cache, on doit rapatrier la ligne en RAM avant d'en charger une nouvelle. On peut améliorer la situation en faisant l'inverse : on charge la nouvelle ligne pendant que l'ancienne donnée soit rapatriée en RAM. Ainsi, la nouvelle ligne est disponible plus tôt pour le processeur, diminuant son temps d'attente. Pour implémenter cette technique, on doit mémoriser l'ancienne ligne de cache temporairement dans un **cache d'éviction** (ou write-back buffer).



Les caches directement adressés ou associatifs par voie possèdent aussi un tampon d'écriture amélioré, qui devient un cache en supplément du cache principal. Pour limiter les défauts par conflit de ces caches, des scientifiques ont eu l'idée d'insérer un cache pour stocker les données virées du cache. En faisant ainsi, si une donnée est virée du cache, on peut alors la retrouver dans ce cache spécialisé. Ce cache s'appelle le **cache de victime**. Ce cache de victime est géré par un algorithme de suppression des lignes de cache de type FIFO. Petit détail : ce cache utilise un tag légèrement plus long que celui du cache directement adressé au-dessus de lui. L'index de la ligne de cache doit en effet être contenu dans le tag du cache de victime, pour bien distinguer deux adresses différentes, qui iraient dans la même ligne du cache juste au-dessus.

Write-through

Avec les caches **Write-Through**, toute écriture dans le cache est propagée en RAM. Ces caches ont tendance commettre beaucoup d'écritures dans la mémoire RAM, ce qui peut saturer le bus reliant le processeur à la mémoire. De plus, on ne peut écrire dans ces caches lorsqu'une écriture en RAM a lieu en même temps : cela forcerait à effectuer deux écritures simultanées, en comptant celle imposée par l'écriture dans le cache.

Pour éviter ces temps d'attentes, certains processeurs avec un cache write-through intègrent un **tampon d'écriture**, une mémoire FIFO dans laquelle on place temporairement les données à transférer du cache vers la RAM en attendant que la RAM soit libre. On n'a pas à se soucier du fait que la mémoire soit occupée, et on peut continuer à écrire dedans tant que celui-ci n'est pas plein, évitant les temps d'attente dus à la RAM. Par souci d'efficacité, des écritures à la même adresse en attente dans le tampon d'écriture sont fusionnées en une seule. Cela fait un peu de place dans le tampon d'écriture, et lui permet d'accumuler plus d'écritures avant de devoir bloquer le cache. Il est aussi possible de fusionner des écritures à adresses consécutives de la mémoire en une seule écriture en rafale. Ainsi, si la taille d'une ligne de cache est petite, comparée à ce qu'une mémoire accepte en écriture en rafale, on peut gagner un peu de performances. Dans les deux cas, on parle de **combinaison d'écriture**.

Reste à gérer le cas où le processeur veut lire une donnée en attente dans le tampon d'écriture. La première manière de gérer cette situation est de mettre en attente la lecture tant que la donnée n'a pas été écrite en mémoire. On peut aussi lire la donnée directement dans le tampon d'écriture, cette optimisation portant le nom de **store-to-load forwarding**. Dans tous les cas, il faut détecter le cas où une lecture accède à une donnée dans le tampon d'écriture. Pour cela, chaque entrée du tampon d'écriture contient un comparateur : à chaque lecture, l'adresse à lire est envoyée à ce comparateur, qui vérifie que l'adresse de la lecture et l'adresse de la donnée à écrire sont différentes. Si jamais ces deux adresses sont identiques, alors la lecture souhaite lire la donnée présente dans l'entrée, et la lecture est mise en attente. Sinon, la lecture a lieu sans attente.

Comportement d'allocation sur écriture

Les écritures posent un autre problème : l'écriture ne va modifier qu'une portion d'une ligne de cache. Que faire quand l'écriture modifie une donnée qui n'est pas dans le cache ? Doit-on charger la ligne de cache correspondante, ou non ?

Allocation sur écriture

La première solution est simple : on alloue une ligne de cache pour l'écriture. On parle de stratégie d'allocation sur écriture (ou write-allocate). Cette solution permet d'implémenter les écritures relativement facilement. Elle peut se décliner en deux sous-catégories : le chargement à la demande et l'écriture immédiate. Dans le premier cas, on charge le bloc à modifier dans le cache, et on effectue l'écriture sur la donnée chargée dans le cache. Dans l'écriture immédiate, le bloc de mémoire n'est pas chargé dans le cache. Une ligne de cache est allouée, et l'écriture a lieu dedans. Évidemment, seule une portion de la ligne de cache contient la donnée écrite (valide), et le reste contient des données invalides. Le cache doit savoir quelles sont les portions du cache qui sont valides : cela demande d'utiliser un sector cache.

Pas d'allocation sur écriture

Sur certains caches, on ne réserve pas de ligne de cache pour effectuer l'écriture. Dans ce cas, l'écriture est transférée directement aux niveaux de cache inférieurs, ou à la mémoire. Le cache ne prend pas en charge l'écriture. Certains caches de ce genre utilisent une petite optimisation : lors de toute écriture, ils supposent que l'écriture donnera un succès de cache. Si c'est le cas, la ligne de cache qui contient la donnée est mise à jour avec la donnée à écrire. Mais si ce n'est pas le cas, la ligne de cache est invalidée, et l'écriture est transférée directement à la mémoire ou aux niveaux de cache inférieurs.

Cohérence des caches

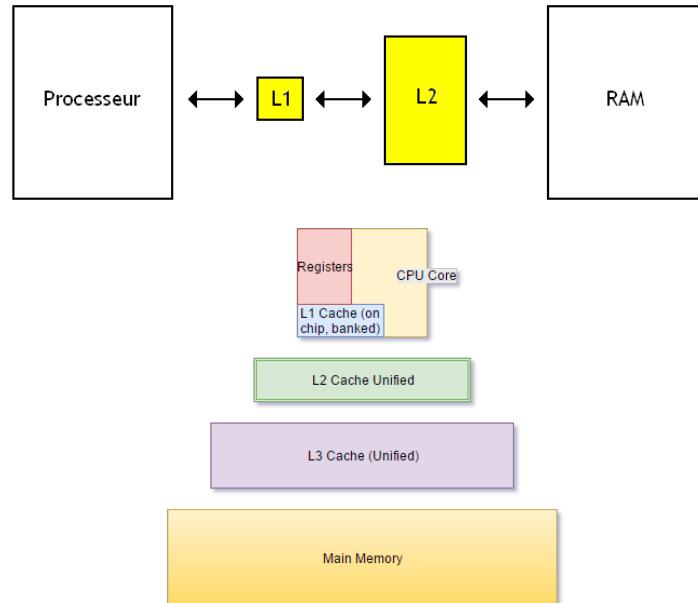
Il arrive parfois que la mémoire d'un ordinateur soit mise à jour, sans que les modifications soient répercutées dans les mémoires cache. Dans ce cas, le cache contient une donnée périmée. Or, un processeur doit toujours éviter de se retrouver avec une donnée périmée et doit toujours avoir la valeur correcte dans ses caches : cela s'appelle la **cohérence des caches**. Il est possible de se retrouver avec des valeurs périmées dans le cache sur les ordinateurs avec plusieurs processeurs, ou si un périphérique écrit en RAM, les modifications ne sont pas répercutées automatiquement dans les mémoires cache. Pour résoudre ce problème, on peut interdire de charger dans le cache des données stockées dans les zones de la mémoire dédiées aux périphériques. Toute lecture ou écriture dans ces zones de mémoire ira donc directement dans la mémoire RAM, sans passer par la ou les mémoires cache. Autre solution : utiliser le fait que les périphériques déclenchent une interruption matérielle pour laisser le contrôleur DMA accéder à la mémoire. Dans ce cas, il suffit de vider les caches à chaque interruption matérielle. Le processeur peut le faire automatiquement, ou fournir des instructions pour.

On n'a pas qu'un seul cache !

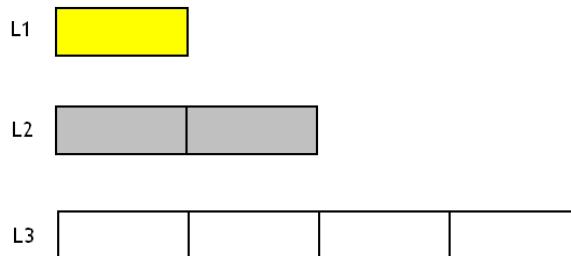
On pourrait croire qu'un seul gros cache est largement suffisant pour compenser la lenteur de la mémoire. Hélas, nos processeurs sont devenus tellement rapides que nos caches sont eux mêmes trop lents ! Pour rappel, plus une mémoire peut contenir de données, plus elle est lente. Et nos caches ne sont pas épargnés. Après tout, qui dit plus de cases mémoires à adresser dans un cache, dit décodeur ayant plus de portes logiques : les temps de propagation s'additionnent et cela finit par se sentir. Si on devait utiliser un seul gros cache, celui-ci serait beaucoup trop lent. La situation qu'on cherche à éviter avec la mémoire principale (la RAM) revient de plus belle.

Hiérarchie de caches

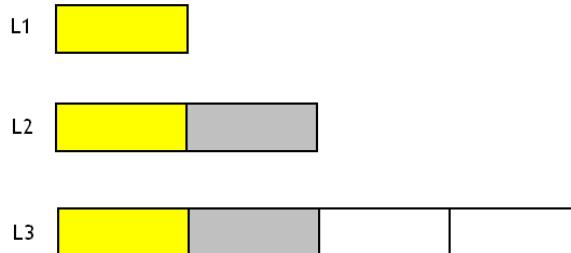
Même problème, même solution : si on a décidé de diviser la mémoire principale en plusieurs mémoires de taille et de vitesse différentes, on peut bien faire la même chose avec la mémoire cache. Depuis environ une vingtaine d'années, nos caches sont segmentés en plusieurs sous-caches : les caches L1, L2 et parfois un cache L3. Certains de ces caches sont petits, mais très rapides : c'est ceux auxquels on va accéder en priorité. Viennent ensuite d'autres caches, de taille variable, mais plus lents. L'accès au cache est simple : on commence par vérifier si notre adresse correspond à une donnée du cache le plus rapide (qui est souvent le plus petit) : j'ai nommé le cache L1. Si elle n'y est pas, on effectue la même vérification pour le cache de niveau inférieur (le L2). Si une donnée est présente dans un des caches, on la charge directement dans le séquenceur (instruction) ou en entrée des unités de calcul (donnée). Dans le cas contraire, on vérifie le cache du niveau inférieur. Et on répète cette opération, jusqu'à avoir vérifié tous les caches : on doit alors aller chercher la donnée en mémoire.



Ces caches sont organisés différemment, et leur contenu varie suivant le cache. Dans les **caches exclusifs**, le contenu d'un cache n'est pas recopié dans le cache de niveau inférieur. Ainsi, le cache ne contient pas de donnée en double et on utilise 100 % de la capacité du cache, ce qui améliore le taux de succès. Par contre, le temps d'accès est plus long, vu qu'il faut vérifier tous les caches de niveau inférieur. Par exemple, si une donnée n'est pas dans le cache L1, on doit vérifier l'intégralité du cache L2, puis du cache L3. De plus, assurer qu'une donnée n'est présente que dans un seul cache nécessite aux différents niveaux de caches de communiquer entre eux pour garantir que l'on a pas de copies en trop d'une ligne de cache, ce qui peut prendre du temps.



Dans le cas des **caches inclusifs**, le contenu d'un cache est recopié dans les caches de niveau inférieur. Par exemple, le cache L1 est recopié dans le cache L2 et éventuellement dans le cache L3.



Maintenir cette inclusion demande toutefois d'échanger des données entre mémoires cache, toute éviction de donnée devant être propagée aux caches de niveau inférieur et supérieur. Premièrement, toute donnée chargée dans un cache doit aussi l'être dans tous les caches de niveau inférieur. Cette contrainte est respectée en maintenant une hiérarchie entre caches lors des accès. Par exemple, si un défaut de cache a lieu dans le L1, celui-ci doit déclencher une lecture dans le L2, lecture qui déclenchera potentiellement un défaut de cache dans le L3, et ainsi de suite jusqu'à trouver la bonne donnée : tous les caches seront parcourus dans l'ordre descendant jusqu'à trouver la donnée. Chaque défaut de cache chargerà la donnée dans le cache correspondant, ce qui fait que tous les caches parcourus auront une copie de la donnée.

Ensuite, quand une donnée est présente dans un cache, elle doit être maintenue dans les niveaux de cache inférieurs. De plus, toute donnée

effacée d'un cache doit être effacée des niveaux de cache supérieurs : si une donnée quitte le cache L2, elle doit être effacée du L1. Ces contraintes posent des problèmes si chaque cache décide du remplacement des lignes de cache en utilisant un algorithme comme LRU, LFU, MRU, ou autre. En effet, dans ce cas, le cache décide de remplacer les lignes de cache selon l'historique des accès, historique qui varie suivant chaque niveau de cache. Par exemple, une donnée rarement utilisée dans le L2 peut parfaitement être très fréquemment utilisée dans le L1 : la donnée sera alors remplacée dans le L2, mais sera maintenue dans le L1. On remarque que ce genre de choses n'a pas lieu sur les caches adressés directement : il n'existe qu'une seule politique de remplacement qui n'utilise pas l'historique des accès.

Pour résoudre ce problème, les caches doivent communiquer entre eux, et se transmettre des informations qui permettent de maintenir l'inclusion. Par exemple, les caches de niveaux inférieurs doivent prévenir les niveaux de cache supérieurs quand ils décident de remplacer une ligne de cache. Ou alors, les caches de niveau inférieurs doivent garder en mémoire les lignes de cache présentes dans le niveau supérieur et refuser de remplacer celles-ci. De même, les écritures dans un niveau de cache doivent être propagées dans les niveaux inférieurs (ou du moins, les niveaux de cache inférieurs doivent être prévenus que leur version de la donnée n'est pas valide).

On observe aussi des problèmes quand il existe plusieurs caches à un seul niveau : chaque cache peut remplacer les lignes de cache d'une manière indépendante des autres caches du même niveau, donnant lieu au même type de problème. Enfin, il faut aussi savoir que la taille des lignes de cache n'est pas la même suivant les niveaux de cache : le L2 peut avoir des lignes plus grandes que celles du L1. Dans ce cas, l'inclusion est plus difficile à maintenir, pour des raisons assez techniques.

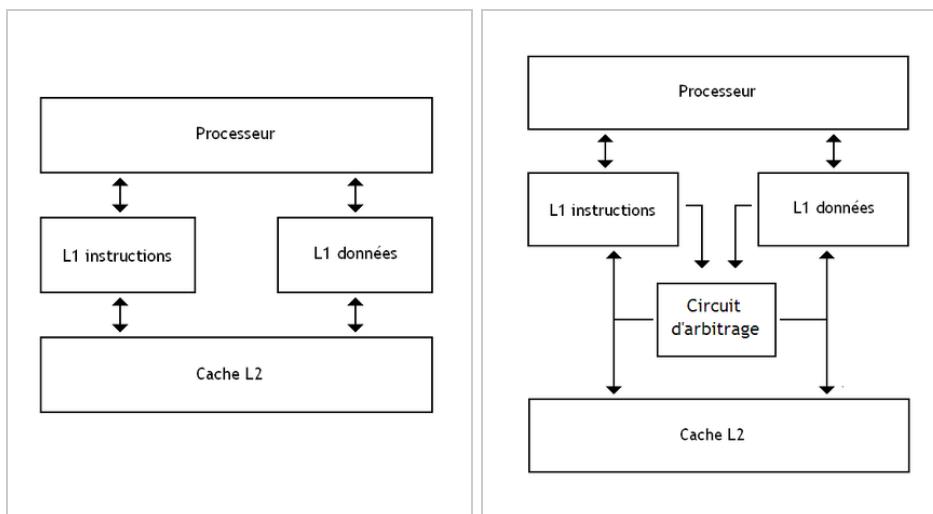
Ce genre de cache a un avantage : le temps d'accès à une donnée présente dans le cache est plus faible. Cela est dû en partie à la présence des copies des caches de niveaux supérieurs dans les caches de niveaux inférieurs (mais pas que). Pour expliquer pourquoi, prenons un exemple : si la donnée voulue n'est pas dans le cache L1, on n'est pas obligé de vérifier la partie du cache L2 qui contient la copie du L1. Ainsi, les circuits qui doivent vérifier si une adresse correspond bien à une donnée placée dans le cache peuvent être simplifiés et ne pas vérifier certaines portions du cache. Ce qui donne un résultat plus rapide.

En contrepartie, une partie des caches de niveau inférieur contient les données contenues dans le cache de niveau supérieur, qui sont donc en double. Cette redondance fait qu'une partie du cache est mal utilisée, comme si le cache était plus petit. De plus, la mise à jour du contenu d'un niveau de cache doit être répercutée dans les niveaux de cache inférieurs et/ou supérieurs. On doit donc transférer des informations de mise à jour entre les différents niveaux de cache. Généralement, le contenu des caches d'instruction n'est pas inclus dans les caches de niveau inférieurs, afin d'éviter que les instructions et les données se marchent sur les pieds.

Et enfin, je me dois de parler des caches qui ne sont ni inclusifs, ni exclusifs. Sur ces caches, chaque niveau de cache gère lui-même ses données, sans se préoccuper du contenu des autres caches. Pas besoin de mettre à jour les niveaux de cache antérieurs en cas de mise à jour de son contenu, ou en cas d'éviction d'une ligne de cache. La conception de tels caches est bien plus simple.

Caches d'instructions

Au fait, sur certains processeurs, le cache L1 est segmenté en deux sous-caches : un qui contient des instructions, et un autre qui ne contient que des données. Ces deux caches étant dans des mémoires séparées et étant reliés au reste du processeur par des bus séparés, on peut charger une instruction et manipuler une donnée en même temps, ce qui est un gros avantage en termes de performances. Ceci dit, cette organisation pose parfois de légers problèmes : rien n'empêche aux deux mémoires cache L1 de vouloir lire ou écrire dans le cache L2 simultanément. Si le cache L2 gère ces écritures/lectures simultanées, pas de problème. Mais s'il ne peut pas, on doit effectuer un arbitrage pour décider quel cache a la priorité. Cet arbitrage est effectué par un circuit spécialisé.



Cache d'instructions.

Circuit d'arbitrage du cache.

Le cache L1 dédié aux instructions est souvent en « lecture seule » : on ne peut pas modifier son contenu, mais juste le lire ou charger des instructions dedans. Cela pose parfois quelques légers problèmes quand on cherche à utiliser du code automodifiant, c'est-à-dire un programme dont certaines instructions vont aller en modifier d'autres, ce qui sert pour faire de l'optimisation ou est utilisé pour compresser voire cacher un programme (les virus informatiques utilisent beaucoup de genre de procédés). Quand le processeur exécute ce genre de code, il ne peut pas écrire dans ce cache L1 d'instructions, mais va devoir écrire en mémoire cache L2 ou en RAM, et va ensuite devoir recharger les instructions modifiées dans le cache L1, ce qui prend du temps ! Et pire : cela peut parfois donner lieu à des erreurs si le cache L1 n'est pas mis à jour.

Sur certains processeurs, l'étape de décodage est assez complexe et lente. Pour accélérer cette étape, certains concepteurs de processeurs ont décidé d'utiliser la (ou les) mémoire cache dédiée aux instructions pour accélérer ce décodage. Lorsque ces instructions sont chargées depuis la RAM ou les niveaux de cache inférieurs, celles-ci sont partiellement décodées. On peut par exemple rajouter des informations qui permettent de délimiter nos instructions ou déterminer leur taille, ce qui est utile pour décoder les instructions de taille variable. Bref, notre cache d'instructions peut se charger d'une partie du décodage des instructions, grâce à un circuit séparé de l'unité de décodage d'instruction.

Enfin, il existe une dernière solution pour limiter les effets de la hiérarchie mémoire. Pour les caches de grande capacité, il arrive souvent que le temps de propagation des signaux varie fortement suivant la ligne de cache à lire. D'ordinaire, on se calle sur la ligne de cache la plus lente pour caler la fréquence d'horloge du cache, mais cela gâche les faibles latences des lignes de cache qui sont tout près du contrôleur de cache. Certains chercheurs ont alors décidé de ruser : ils acceptent d'avoir une latence différente pour chaque ligne d'un même cache. Les caches qui fonctionnent sur ce principe sont appelés des **caches à accès non uniforme**. La première version de ce genre de caches a une correspondance ligne de cache → bloc de mémoire statique : on ne peut pas déplacer le contenu d'une ligne de cache dans une autre portion de mémoire plus rapide suivant les besoins. Mais des versions plus optimisées en sont capables : la correspondance entre une ligne de cache et un bloc de mémoire cache peut varier à l'exécution. Ainsi, les lignes de cache les plus utilisées peuvent migrer dans un bloc de mémoire plus rapide : cela demande juste de copier les

données entre blocs de mémoire et de mettre à jour la correspondance entre ligne de cache et bloc de mémoire.

Cache de boucle

Il est possible d'optimiser le fonctionnement des boucles sur les processeurs pipelinés. D'ordinaire, lorsqu'une instruction doit être exécutée plusieurs fois dans un temps très court, elle doit être chargée depuis la mémoire et décodée, puis exécutée plusieurs fois. Sur les processeurs qui disposent de pipelines, ce chargement répété peut être omis en utilisant un cache de boucle, un cache chargé de stocker les dernières instructions chargées et/ou décodées. Si une instruction doit être réexécutée (signe d'une boucle), il suffit d'aller chercher l'instruction directement dans le cache de boucle, au lieu de la recharger. Néanmoins, ce cache ne peut stocker qu'un nombre limité d'instructions, ce qui limite la taille des boucles pouvant profiter de ce cache.

Caches non bloquants

Un **cache bloquant** est un cache auquel le processeur ne peut pas accéder pendant un défaut de cache. Il faut alors attendre que la donnée voulue soit lue ou écrite dans la RAM avant de pouvoir utiliser de nouveau le cache. Un **cache non bloquant** n'a pas ce problème : on peut l'utiliser immédiatement après un défaut de cache. Cela permet d'accéder à la mémoire cache en attendant des données en provenance de la RAM. Tous les caches non bloquants peuvent ainsi permettre de démarrer une nouvelle lecture ou écriture alors qu'une autre est en cours. On peut ainsi exécuter plusieurs lectures ou écritures en même temps : c'est ce qu'on appelle du parallélisme au niveau de la mémoire (memory level parallelism). Mais au bout d'un certain nombre d'accès simultané, le cache va saturer et va dire « stop ». Il ne peut supporter qu'un nombre limité d'accès mémoires simultanés (pipelinés).

Succès après défaut

Ces caches non bloquants sont de deux types. Il y a ceux qui autorisent les lectures et écritures dans le cache pendant qu'un défaut de cache a lieu. Pendant un défaut de cache, le contrôleur mémoire va charger la donnée du défaut de cache depuis la mémoire : le cache n'est pas utilisé, ni en lecture, ni en écriture. Dans ces conditions, on peut modifier le contrôleur mémoire pour qu'il gère les succès de cache qui suivent le défaut : on peut donc lire ou écrire dans le cache. Mais le prochain défaut bloquera toute la mémoire cache, empêchant toute lecture ou écriture dedans. Les éventuels succès de cache qui pourront suivre ce deuxième défaut seront alors mis en attente.

Sur certaines mémoires cache, lors d'un défaut de cache, on doit attendre que toute la ligne concernée par le défaut de cache soit chargée avant d'être utilisable. Un cache stocke des lignes de cache dont la taille est supérieure à la largeur du bus mémoire. En conséquence, une ligne de cache est chargée en plusieurs fois, morceaux par morceaux. Certains se sont demandés si on ne pouvait pas gagner quelques cycles en jouant dessus. Eh bien c'est le cas ! Certains processeurs sont capables d'utiliser un morceau de ligne de cache tout juste chargé alors que la ligne de cache complète n'est pas totalement lue depuis la mémoire. Avec cette technique, le cache est utilisable lors d'un défaut de cache, ce qui fait qu'il est légèrement non bloquant. Ces processeurs incorporent un tampon de remplissage de ligne (line-fill buffer), une sorte de registre, qui stocke le dernier morceau d'une ligne de cache à avoir été chargé depuis la mémoire. Ce morceau est stocké avec un tag, qui indique l'adresse du bloc stocké dans le tampon de remplissage de ligne. Ainsi, un processeur qui veut lire dans le cache après un défaut peut accéder à la donnée directement depuis le tampon de remplissage de ligne, alors que la ligne de cache n'a pas encore été totalement recopiée en mémoire.

Pour améliorer encore plus les performances, et utiliser au mieux ce tampon de remplissage de ligne, les processeurs actuels implémentent des techniques de critical word load. Pour faire simple, on va comparer le chargement d'une ligne de cache avec et sans critical word load. Sans critical word load, la ligne de cache est chargée morceau par morceau, dans l'ordre de placement de ces morceaux en mémoire. Mais si l'adresse lue ou écrite par le processeur n'est pas le premier mot mémoire de la ligne de cache, il devra attendre que celui-ci soit chargé. Durant ce temps d'attente, les blocs qui précèdent la donnée demandée par le processeur seront chargés. Avec le critical word load, le contrôleur mémoire va charger les blocs en commençant par le bloc dans lequel se trouve la donnée demandée, avant de poursuivre avec les blocs suivants, avant de revenir au début du bloc pour charger les blocs restants. Ainsi, la donnée demandée par le processeur sera la première disponible.

Défaut après défaut

Au-dessus, on a vu des caches capables de lire ou d'écrire lors d'un défaut. Ceux-ci sont incapables de mettre en attente d'autres défauts en attendant que le défaut en cours soit terminé. Mais sur d'autres caches, on peut gérer un nombre arbitraire de défaut de cache avant de bloquer, permettant ainsi aux lectures et écritures qui suivent un deuxième ou troisième défaut de fonctionner. Ceux-ci sont dits de type défaut après défaut (miss under miss).

Ceci dit, ces caches doivent être adaptés. Pendant qu'une lecture ou une écriture est démarrée par le contrôleur mémoire, il ne peut pas démarrer de nouvelle lecture ou écriture immédiatement après (sauf dans certains cas exceptionnels) : il y a toujours un temps d'attente entre l'envoi de deux requêtes d'accès mémoire. Dans ces conditions, les autres défauts doivent être mis en attente : la lecture/écriture doit être mémorisée, et est démarrée par le contrôleur mémoire une fois que celui-ci est libre, qu'il peut démarrer une nouvelle requête. Cette mise en attente s'effectue avec des miss status handling registers. Les miss status handling registers, que j'appellerais dorénavant MSHR, sont des registres qui vont mettre en attente les défauts de cache. Le nombre de MSHR indique combien de blocs de mémoire, de lignes de cache, peuvent être lues en même temps depuis la mémoire. Chacun de ces registres stocke au minimum trois informations :

- l'adresse à l'origine du défaut de cache ;
- sa destination : le numéro de la ligne de cache où stocker la donnée du défaut de cache ;
- un bit de validité qui indique si le MSHR est vide ou pas.

Quand la donnée lue depuis la mémoire est devenue disponible, le MSHR correspondant (celui qui stockait l'adresse à lire) est invalidé : son bit de validité est mis à 0. D'autres informations peuvent être stockées dans un MSHR, mais cela dépend fortement du processeur. Sur les caches directement adressés ou associatifs à n voies, on peut par exemple stocker le numéro de la ligne de cache qui doit mémoriser la donnée chargée depuis la mémoire.

Je vais maintenant décrire le fonctionnement d'un cache non bloquant qui utilise des MSHR. Pour commencer, nous allons prendre le cas où l'adresse à lire ou écrire n'a pas subi de défaut de cache récent : il n'y a aucun défaut de cache en attente à cette adresse. Dans ce cas, il suffit de réservé un MSHR vide pour ce défaut. Maintenant, prenons un autre cas : un ou plusieurs défaut de cache sont déjà en attente pour cette même adresse. Déetecter cette situation est simple : il suffit de comparer l'adresse du défaut avec celles déjà mises en attente dans les MSHR. Pour ce faire, chaque MSHR est relié à un comparateur, et avec quelques circuits. Dans ces conditions, certains caches mettent le cache en pause : celui-ci ne peut alors plus accepter de nouveau défaut de cache tant que le premier défaut de cache n'est pas résolu. Si deux défauts de cache différents veulent lire un même bloc de mémoire, le cache se met en pause au deuxième défaut de cache : il reste en pause tant que le premier défaut de cache n'est pas résolu. D'autres caches sont plus malins, et peuvent mettre en attente plusieurs défauts de cache qui tombent sur la même adresse.

Les MSHR sont une première avancée, qui permet à un processeur de gérer plusieurs défauts de cache dans des zones du cache différentes. Toutefois, un second accès mémoire à la même ligne de cache va bloquer le cache. On se retrouve alors dans deux cas. Dans le premier cas, les adresses des deux défauts de cache ont des tags différents : les zones de mémoire adressées lors des deux défauts de cache sont séparées dans la mémoire, et ne font pas partie du même bloc. Dans ce cas, le blocage du cache est inévitable. Le premier défaut de cache doit être géré avant que le second puisse démarrer. Dans l'autre cas, les tags sont identiques, mais pas les index et les décalages. Les données sont proches les unes des autres en mémoire, font partie du même bloc, mais ne se recouvrent pas : il s'agit de données placées les unes à côté des autres, qui seront localisées dans la même ligne de cache. Quand le premier défaut de cache sera résolu, les données du second défaut de cache seront présentes dans le bloc. Dans ces conditions, il est inutile de bloquer le cache : le second défaut de cache peut être fusionné avec le premier.

Pour éviter de bloquer le cache dans la seconde situation, certains caches traitent les requêtes avec une granularité plus fine. Leur principe est simple : quand un défaut de cache a lieu et que celui-ci va lire les données chargées par un défaut de cache précédent, les deux défauts sont fusionnés dans le même MSHR. Cette détection de coïncidence de l'adresse s'effectue lors de la vérification des MSHR : le tag contenu dans le

MSHR est comparé avec le tag de l'adresse responsable du nouveau défaut de cache. Reste que les différents défauts de cache vont lire des données différentes dans le cache : les index et les décalages des adresses sont différents. Lorsque le défaut sera résolu, et que la donnée sera disponible, il faudra bien configurer le cache avec les index et décalages corrects. Dans ces conditions, il faut bien les mémoriser quelque part. Et les MSHR sont tout indiqués pour cela.

La solution la plus simple consiste à utiliser des MSHR adressés implicitement. Ces MSHR contiennent une entrée pour chaque mot mémoire dans la ligne de cache concernée par le défaut. Chacune de ces entrées dira si un défaut de cache compte lire ou écrire dans le mot mémoire en question. Ces entrées contiennent diverses informations :

- la destination de la lecture effectuée par le défaut de cache (généralement un registre) ;
- un format, qui donne des informations sur l'instruction à l'origine du défaut ;
- un bit de validité, qui dit si le mot mémoire associé à l'entrée est lu ou non par le défaut de cache ;
- un bit empty, qui dit si l'entrée est vide ou non.

La fusion de deux défauts de cache est ainsi assez simple. Si deux défauts de cache vont lire des mots mémoire différents, il suffira de configurer les entrées de mot mémoire de chaque défaut convenablement. Le premier défaut configura ses entrées, et le second aura les siennes. Le même MSHR sera réutilisé pour les deux défauts. Et on peut ainsi gérer bien plus de deux défauts : tant que des défauts de cache vont lire des mots mémoire différents, on peut les fusionner, sans vraiment de limite. Par contre, si deux défauts de cache veulent réserver la même entrée, alors là, c'est le drame : le cache se bloque automatiquement ! Avec cette organisation, le nombre de MSHR indique combien de lignes de cache peuvent être lues en même temps depuis la mémoire. Quant aux nombre d'entrées par MSHR, il détermine combien d'accès mémoires qui ne se recouvrent pas peuvent avoir lieu en même temps.

Pour éviter de bloquer le cache lors d'accès à la même ligne de cache, certains chercheurs ont inventés des MSHR adressés explicitement. Ces MSHR sont aussi constitués d'entrées. La différence, c'est qu'une entrée est réservée à un accès mémoire, et non à un mot mémoire dans le cache. Chaque entrée va ainsi stocker :

- les index et décalages d'un accès mémoire dans la ligne de cache ;
- la destination de la lecture effectuée par le défaut de cache (généralement un registre) ;
- un format, qui donne des informations sur l'instruction à l'origine du défaut ;
- un bit empty, qui dit si l'entrée est vide ou non.

Avec cette organisation, le nombre de MSHR indique combien de lignes de cache peuvent être lues en même temps depuis la mémoire. Quant aux nombre d'entrées par MSHR, il détermine combien d'accès mémoires qui atterrissent dans la même ligne de cache peuvent avoir lieu en même temps, sans contraintes de recouvrement.

Généralement, plus on veut supporter de défauts de cache, plus le nombre de MSHR et d'entrées augmente. Mais au-delà d'un certain nombre d'entrées et de MSHR, les MSHR adressés implicitement et explicitement ont tendance à bouffer un peu trop de circuits. Utiliser une organisation un peu moins gourmande en circuits est donc une nécessité. Cette organisation plus économique se base sur des MSHR inversés. Ces MSHR ne contiennent qu'une seule entrée, en quelque sorte : au lieu d'utiliser n MSHR de m entrée chacun, on va utiliser $n \times m$ MSHR inversés. La différence, c'est que plusieurs MSHR peuvent contenir un tag identique, contrairement aux MSHR adressés implicitement et explicitement. Lorsqu'un défaut de cache a lieu, chaque MSHR est vérifié. Si jamais aucun MSHR ne contient de tag identique à celui utilisé par le défaut, un MSHR vide est choisi pour stocker ce défaut, et une requête de lecture en mémoire est lancée. Dans le cas contraire, un MSHR est réservé au défaut de cache, mais la requête n'est pas lancée. Quand la donnée est disponible, les MSHR correspondant à la ligne qui vient d'être chargée vont être utilisés un par un pour résoudre les défauts de cache en attente.

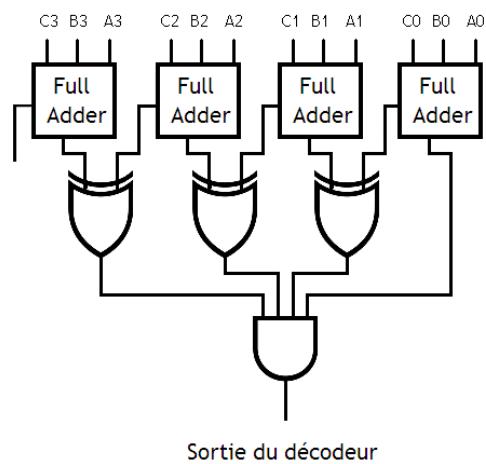
Certains chercheurs ont remarqué que pendant qu'une ligne de cache est en train de subir un défaut de cache, celle-ci reste inutilisée, et son contenu est destiné à être perdu une fois le défaut de cache résolu. Ils se sont dit que, plutôt que d'utiliser des MSHR séparés, il vaudrait mieux utiliser la ligne de cache pour stocker les informations sur les défaut de cache en attente dans cette ligne de cache. Pour éviter tout problème, il faut rajouter un bit dans les bits de contrôle de la ligne de cache, qui sert à indiquer que la ligne de cache est occupée : un défaut de cache a eu lieu dans cette ligne, et elle stocke donc des informations pour résoudre les défauts de cache.

Contourner le cache (cache bypassing)

Dans certains cas, il arrive que des données avec une faible localité soient chargées dans le cache inutilement. Il vaut mieux que ces données transittent directement entre le processeur et la mémoire, sans passer par l'intermédiaire du cache. Pour cela, le processeur peut fournir des instructions d'accès mémoire qui ne passent pas par le cache, à côté d'instructions normales. Mais il existe aussi des techniques matérielles : c'est le cache qui s'occupe de détecter les données utiles à l'exécution. Cette méthode repose sur l'identification des instructions à l'origine des défauts de cache, le processeur accédant directement à la RAM quand une telle instruction est détectée. Pour identifier ces instructions, le processeur dispose d'une mémoire, qui mémorise les program counters des instructions d'accès mémoire déjà rencontrées lors de l'exécution du programme. On appelle cette mémoire la **table d'historique des défauts de lecture** (load miss history table). A chaque nouvelle exécution d'une instruction d'accès mémoire, une entrée de cette mémoire est réservée pour cette instruction. Chaque adresse est couplée à un compteur de quelques bits : ce compteur est incrémenté à chaque succès de cache, et décrémenté à chaque défaut de cache. À chaque fois que l'on exécute une instruction, son program counter est envoyé en entrée de la table d'historique des défauts de lecture : si jamais l'instruction était déjà présente dans celle-ci, la valeur du compteur est récupérée, et le processeur vérifie celle-ci. Si elle tombe en dessous d'un certain seuil, c'est que l'instruction a commis trop de défauts de cache, et qu'il faut éviter de lui allouer du cache.

Caches adressés par somme et hashés

Pour rappel, certains modes d'adressage ont besoin de faire des calculs d'adresse, les plus communs consistant à ajouter un décalage à une adresse. Cette addition est parfois faite dans l'unité de calcul ou dans l'Address Generation Unit, mais processeurs effectuent ce calcul directement dans le cache en modifiant les décodeurs de celui-ci. Pour rappel, un décodeur normal est composé de comparateurs, qui vérifient si l'entrée est égale à une constante bien précise. Par contre, le décodeur nouvelle version est composé de comparateurs qui testent si la somme adresse + décalage est égale à une constante. Si on sait créer ces circuits de base, il suffira d'en mettre plusieurs, de les calibrer convenablement, et le tour est joué ! Or, vérifier si $A + B = K$ est équivalent à vérifier que $A + B - K = 0$. Il suffit donc d'utiliser un additionneur carry-save pour faire l'addition des trois termes vus plus haut, et le tour est joué ! Il ne reste plus qu'à dupliquer ces circuits, en modifiant l'additionneur (pour régler la constante), et le tour est joué ! Comme on le voit, l'addition disparaît et est remplacée par quelques portes logiques en plus. Autant dire que l'on gagne tout de même un petit peu en rapidité. Mais ne vous affolez pas : on gagne assez peu, juste de quoi gagner un cycle d'horloge sur un accès au cache L1/L2. Autant dire que ce n'est pas une technique miracle pour avoir un cache accessible rapidement.



Le préchargement

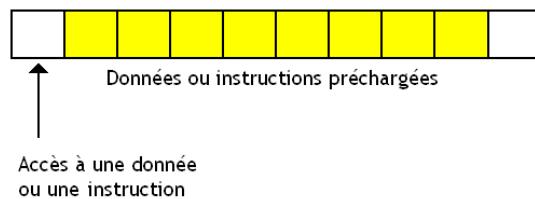
En raison de la localité spatiale, il est avantageux de précharger des données proches de celles chargées il y a peu. Ce **préchargement** (en anglais, *prefetching*) peut être effectué par le programmeur, à la condition que le processeur possède une instruction de préchargement. Mais celui-ci peut aussi être pris en charge directement par le processeur, sans implication du programmeur, ce qui a de nombreux avantages. Pour ce faire, le processeur doit contenir un circuit nommé prefetcher. Qui plus est, on peut utiliser à la fois des instructions de préchargement et un prefetcher intégré au processeur : les deux solutions ne sont pas incompatibles.

Préchargement des données

Certains prefetchers sont adaptés à l'utilisation de tableaux (des ensembles de données consécutives de même taille et de même type). Ils profitent du fait que ces tableaux sont souvent accédés case par case.

Prefetchers séquentiels

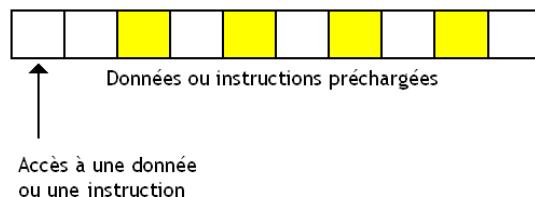
Les prefetchers séquentiels préchargent les données immédiatement consécutives de la donnée venant tout juste d'être lue ou écrite. Ils fonctionnent bien lorsqu'on accède à des données consécutives en mémoire, ce qui arrive souvent lors de parcours de tableaux. Dans le cas le plus simple, on précharge le bloc de mémoire qui suit immédiatement la dernière ligne chargée. L'adresse de ce bloc se calcule en additionnant la longueur d'une ligne de cache à la dernière adresse lue ou écrite. Ce qui peut être fait avec un seul bloc de mémoire peut aussi l'être avec plusieurs. Rien n'empêche de charger non pas un, mais deux ou trois blocs consécutifs dans notre mémoire cache. Mais attention : le nombre de blocs de mémoire chargés dans le cache est fixe. Cela fonctionne bien si l'on utilise des données avec une bonne localité spatiale.



Ces prefetchers ne fonctionnent que pour des accès à des adresses consécutives : si ce n'est pas le cas, l'utilisation d'un prefetcher séquentiel est contreproductive. Pour limiter la casse, les prefetchers sont capables de reconnaître les accès séquentiels et les accès problématiques. Cette détection peut se faire de deux façons. Avec la première, le prefetcher va calculer une moyenne du nombre de blocs préchargés qui ont été utiles, à partir des n derniers blocs préchargés. En clair, il va calculer le rapport entre le nombre de blocs qu'il a préchargés dans le cache et le nombre de ces blocs qui ont été accédés. Si jamais ce rapport diminue trop, cela signifie que l'on n'a pas affaire à des accès séquentiels : le prefetcher arrêtera temporairement de précharger. Autre solution : garder un historique des derniers accès mémoires et voir s'ils accèdent à des zones consécutives de la mémoire. Si ce n'est pas le cas, le prefetcher ne précharge pas. Le processeur peut même décider de désactiver temporairement le préchargeage si jamais le nombre de blocs préchargés utilement tombe trop près de zéro.

Accès par enjambées

Certains accès, les **accès par enjambées** (ou stride access), se font sur des données séparées par une distance constante k . Ils ont comme origine les parcours de tableaux multidimensionnels, et de tableaux de structures/objets. Avec ce genre d'accès, un prefetcher séquentiel charge des données inutiles, ce qui est synonyme de baisse de performances. Mais certains prefetchers gèrent de tels accès à la perfection. Cela ne rend pas ces accès aussi rapides que des accès à des blocs de mémoire consécutifs, vu qu'on gâche une ligne de cache pour n'en utiliser qu'une petite portion, mais cela aide tout de même beaucoup.

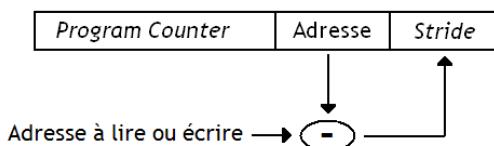


Ces prefetchers conservent un historique des accès mémoires effectués récemment dans un cache : la **table de prédiction de références** (reference prediction table). Chaque ligne de cache associe à une instruction toutes les informations nécessaires pour prédire quelle sera la prochaine adresse utilisée par celle-ci. Elle stocke notamment la dernière adresse lue/écrite par l'instruction, l'enjambée, ainsi que des bits qui indiquent la validité de ces informations. L'adresse de l'instruction est dans le tag de la ligne de cache.

L'algorithme de gestion des enjambées doit :

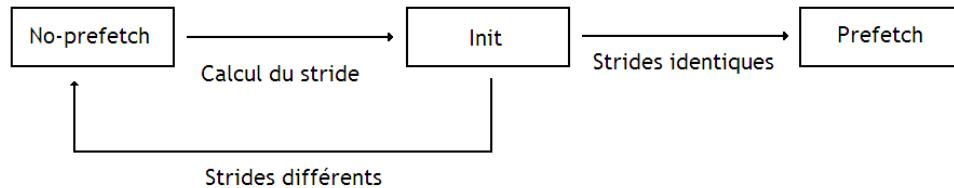
- déterminer la taille de l'enjambée ;
- détecter les enjambées ;
- précharger un ou plusieurs blocs.

Détecter les enjambées et déterminer leur taille peut se faire simultanément de différentes manières. La première considère que si une instruction effectue deux défauts de cache à la suite, elle effectue un accès par enjambées. Il s'agit là d'une approximation grossière, mais qui ne fonctionne pas trop mal. Avec cette méthode, une ligne de cache de la table de prédiction de référence peut avoir deux états : un où l'instruction n'effectue pas d'accès par enjambées, qui désactive le préchargement, ainsi qu'un second état pour les instructions qui effectuent des accès par enjambées qui lui l'active. La première fois qu'une instruction effectue un défaut de cache, une entrée lui est allouée dans la table de prédiction de références. La ligne est initialisée avec une enjambée inconnue en état "préchargement désactivé". Lors du second accès, l'enjambée est calculée et la ligne de cache est mise à jour en état "préchargement activé". Le calcul de l'enjambée s'effectue en soustrayant l'adresse précédente de l'adresse en cours.

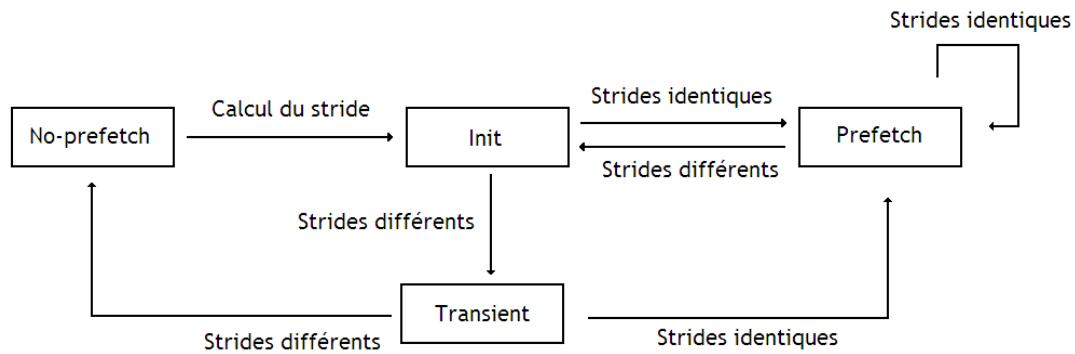


Néanmoins, cet algorithme a un petit problème : il arrive très souvent qu'il voie des accès par enjambées là où il n'y en a pas. Une solution à cela consiste à rajouter un état init. Lorsque l'instruction effectue son premier défaut de cache, l'entrée est initialisé dans l'état no prefetch. Lors du défaut de cache suivant, l'enjambée est calculée, mais le préchargement ne commence pas : l'entrée est placée dans l'état init. C'est lors d'un

troisième défaut de cache que l'enjambée est recalculée, et comparée avec l'enjambée calculée lors des deux précédents défauts. Si les deux correspondent, un accès par enjambées est détecté, et le préchargement commence. Sinon, l'instruction n'effectue pas d'accès par enjambées : on place l'entrée en état no prefetch.



On peut améliorer l'algorithme précédent pour recalculer l'enjambée à chaque accès mémoire de l'instruction, et vérifier si celui-ci a changé. Si un changement est détecté, la prédiction avec enjambée est certainement fausse et on ne précharge rien. Pour que cet algorithme fonctionne, on doit ajouter un quatrième état aux entrées : « transitoire » (transient), qui stoppe le préchargement et recalcule l'enjambée.

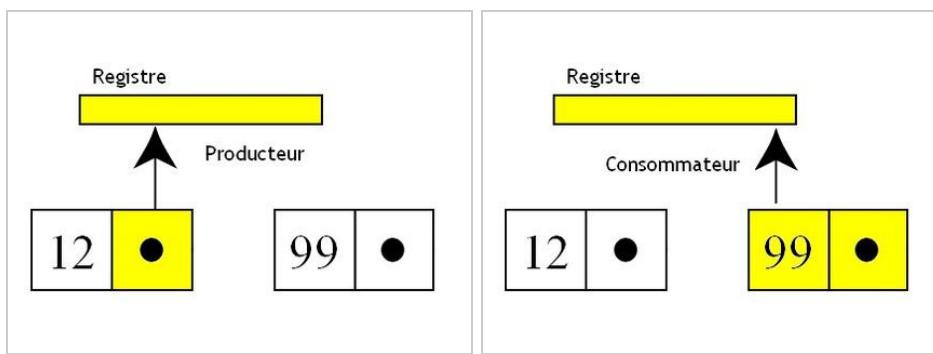


Pour prédire la prochaine adresse, il suffit d'ajouter la longueur d'enjambée à l'adresse à lire ou écrire. Cette technique peut être adaptée pour précharger non seulement la prochaine adresse, mais aussi les n adresses suivantes, la i ème adresse ayant pour valeur : adresse + $n \times$ enjambée. Évidemment, ces adresses à précharger ne peuvent pas être lues ou écrits simultanément depuis la mémoire. On doit donc les mettre en attente, le temps que la mémoire soit libre. Pour cela, on utilise un tampon de préchargement, qui stocke des requêtes de lecture ou d'écriture fournies par l'unité de préchargement.

Préchargement selon les dépendances

Certaines applications ont besoin de structures de données qui permettent de supprimer ou d'ajouter un élément rapidement. Pour cela, on doit utiliser des structures de données alternatives aux tableaux, dans lesquelles les données peuvent se retrouver à des endroits très éloignés en mémoire. Pour faire le lien entre les données, chacune d'entre elles sera stockée avec les adresses des données suivantes ou précédentes. On peut notamment citer les listes, les arbres et les graphes. Il va de soi que les méthodes précédentes fonctionnent mal avec ces structures de données, où les données ne sont pas placées à intervalle régulier en mémoire. Cela dit, ça ne veut pas dire qu'il n'existe pas de technique de préchargement adaptée pour ce genre de structures de données. En fait, il existe deux grandes techniques pour ce genre de structures de données, et quelques autres techniques un peu plus confidentielles.

La première de ces techniques a reçu le nom de **préchargement selon les dépendances** (dependence based prefetching). Elle ne donne de bons résultats que sur des listes. Prenons un exemple : une liste simplement chaînée, une structure où chaque donnée indique l'adresse de la suivante. Pour lire la donnée suivante, le processeur doit récupérer son adresse, qui est placée à côté de la donnée actuelle. Puis, il doit charger tout ou partie de la donnée suivante dans un registre. Comme vous le voyez, on se retrouve avec deux lectures : la première récupère l'adresse et l'autre l'utilise. Dans ce qui va suivre, je vais identifier ces deux instructions en parlant d'instruction productrice (celle qui charge l'adresse) et consommatrice (celle qui utilise l'adresse chargée).



Instruction productrice.

Instruction consommatrice.

Avec le préchargement selon les dépendances, le processeur mémorise si deux instructions ont une dépendance producteur-consommateur dans un cache : la **table de corrélations**. Celle-ci stocke les adresses du producteur et du consommateur. Reste que ces corrélations ne sortent pas de cuisse de Jupiter : elles sont détectées lors de l'exécution d'une instruction consommatrice. Pour toute lecture, le processeur vérifie si la donnée à lire a été chargée par une autre instruction : si c'est le cas, l'instruction est consommatrice. Reste à mémoriser les adresses lues et l'instruction de lecture associée. Pour cela, le processeur contient une table de correspondances entre la donnée lue et l'adresse de l'instruction (le program counter) : la **fenêtre de producteurs potentiels**. Lors de l'exécution d'une instruction, il vérifie si l'adresse à lire est dans la fenêtre de producteurs potentiels : si c'est le cas, c'est qu'une instruction productrice a chargé l'adresse, et que l'instruction en cours est consommatrice. L'adresse des instructions productrice et consommatrice sont alors stockées dans la table de corrélations. A chaque lecture, le processeur vérifie si l'instruction est productrice en regardant le contenu de la table de corrélations. Dès qu'une instruction détectée comme productrice a chargé son adresse, le processeur précharge les données de l'instruction consommatrice associée. Lorsqu'elle s'exécutera quelques cycles plus tard, la donnée aura déjà été lue depuis la mémoire.

Préchargement de Markov

Pour gérer des structures de données plus évoluées, comme des arbres ou des graphes, la technique vue plus haut ne marche pas très bien. Avec ces types de données, chaque donnée a plusieurs successeurs, ce qui fait qu'une instruction consommatrice ne va pas toujours consommer la même adresse (celle-ci dépendant du successeur). Pour gérer cette situation, on doit utiliser des prefetchers plus évolués, comme **des prefetchers de Markov**. Ces prefetchers de Markov fonctionnent comme les précédents, sauf que la table de corrélations permet de mémoriser plusieurs correspondances, plusieurs adresses de successeurs. Chaque adresse se voit attribuer une probabilité, qui indique si elle a plus de chance d'être la bonne donnée à précharger que ses voisines. À chaque lecture ou écriture, les probabilités sont mises à jour. Dans certains prefetchers, seule l'adresse de plus forte probabilité. Mais dans d'autres, toutes les adresses de destination sont préchargées. Vu que la mémoire ne peut précharger qu'une seule donnée à la fois, certaines adresses sont alors mises en attente dans une mémoire tampon de précharge. Lors du préchargement, le program counter de l'instruction qui initiera le préchargement sera envoyé à cette table. Cette table fournira plusieurs adresses, qui seront mises en attente dans le tampon de précharge avant leur préchargement. L'ordre d'envoi des requêtes de préchargement (le passage de la mémoire tampon au sous-système mémoire) est déterminé par les probabilités des différentes adresses : on précharge d'abord les adresses les plus probables.

Préchargement par distance

Le gain apporté par les prefetchers vus auparavant est appréciable, mais ceux-ci fonctionnent mal sur des accès cycliques ou répétitifs, certes rares dans le cas général, mais présents à foison dans certaines applications. Ils apparaissent surtout quand on parcourt plusieurs tableaux à la fois. Pour gérer au mieux ces accès, on a inventé des prefetchers plus évolués, capables de ce genre de prouesses.



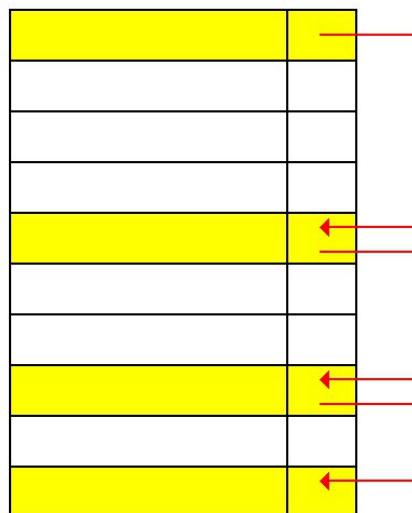
Le **préchargement par distance** (distance prefetching), une adaptation du prefetcher du Markov, est un de ces prefetchers. Celui-ci n'utilise pas les adresses, mais les différences entre adresses accédées de manière consécutive, qui sont appelées des deltas. Ces deltas se calculent tout simplement en soustrayant les deux adresses. Ainsi, si j'accède à une adresse A, suivie par une adresse B, le préchargement par distance calculera le delta B - A, et l'utilisera pour sélectionner une entrée dans la table de correspondances. La table de correspondances est toujours structurée autour d'entrées, qui stockent chacune plusieurs correspondances, sauf qu'elle stocke les deltas. Cette table permet de faire des prédictions du style : si le delta entre B et A est de 3, alors le delta entre la prochaine adresse sera soit 5, soit 6, soit 7. L'utilité du prefetcher de Markov, c'est que la même entrée peut servir pour des adresses différentes.

Tampon d'historique global

Les techniques vues plus haut utilisent toutes une sorte de table de correspondances. L'accès à la table s'effectue soit en envoyant le program counter de l'instruction en entrée (préchargement par enjambées), soit l'adresse lue, soit les différences entre adresses. Ce qui est envoyé en entrée sera appelé l'**index** de la table, dans la suite de cette partie. Cette table stocke une quantité limitée de données, tirées de l'historique des défauts de cache précédents. En somme, la table stocke, pour chaque index, un historique des défauts de cache associés à l'index. Dans les techniques vues précédemment, chaque table stocke un nombre fixe de défauts de cache par index : le one block lookahead stocke une adresse par instruction, le stride stocke une enjambée et une adresse pour chaque instruction, le préchargement de Markov stocke une ou plusieurs adresses par instruction, etc. Dit autrement, l'historique a une taille fixe.

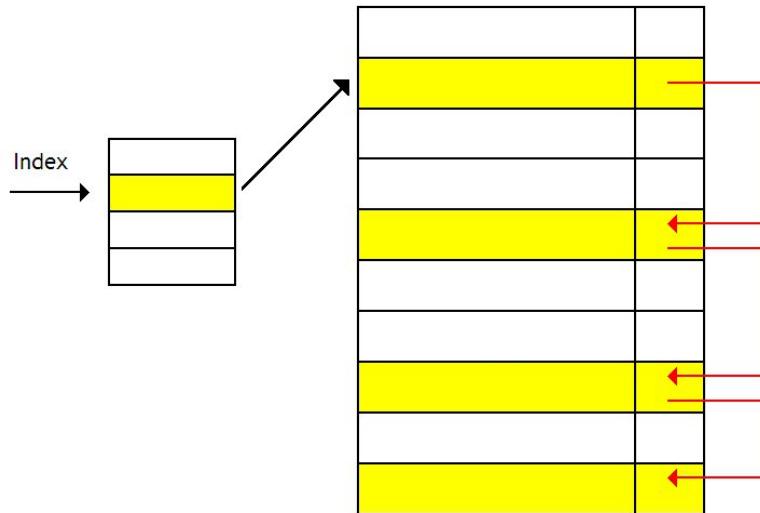
Vu que cette quantité est fixe, elle est souvent sous-utilisée. Par exemple, le préchargement de Markov limite le nombre d'adresses pour chaque instruction à 4, 5, 6 suivant le prefetcher. Certaines instructions n'utilisent jamais plus de deux entrées, tandis que le nombre de ces entrées n'est pas suffisant pour d'autres instructions plus rares. La quantité d'informations mémorisée pour chaque instruction est toujours la même, alors que les instructions n'ont pas les mêmes besoins : c'est loin d'être optimal. De plus, le nombre de défauts de cache par index limite le nombre d'instructions ou d'adresses qui peuvent être prédictes. De plus, il se peut que des données assez anciennes restent dans la table de prédition, et mènent à de mauvaises prédictions : pour prédire l'avenir, il faut des données à jour. Pour éviter ce genre de défauts, les chercheurs ont inventé des prefetchers qui utilisent un tampon d'historique global (global history buffer). Celui-ci permet d'implémenter plusieurs techniques de préchargement. Les techniques précédentes peuvent s'implémenter facilement sur ces prefetchers, mais sans les défauts cités au-dessus.

Ces prefetchers sont composés de deux sous-composants. Premièrement, on trouve une mémoire tampon de type FIFO qui mémorise les défauts de cache les plus récents : l'**historique global**. Pour chaque défaut de cache, la mémoire FIFO mémorise l'adresse lue ou écrite dans une entrée. Pour effectuer des prédictions crédibles, ces défauts de cache sont regroupés suivant divers critères : l'instruction à l'origine du défaut, par exemple. Pour cela, les entrées sont organisées en liste chaînée : chaque entrée pointe sur l'entrée suivante qui appartient au même groupe. On peut voir chacune de ces listes comme un historique dédié à un index : cela peut être l'ensemble des défauts de cache associés à une instruction, l'ensemble des défauts de cache qui suivront l'accès à une adresse donnée, etc. Généralement, les instructions sont triées à l'intérieur de chaque groupe dans l'ordre d'arrivée : l'entrée la plus récente contient le défaut de cache le plus récent du groupe. Ainsi, la taille de l'historique s'adapte dynamiquement suivant les besoins, contrairement aux prefetchers précédents où celui-ci tait de taille fixe.



Reste que le processeur doit savoir où est l'entrée qui correspond au début de chaque liste. Pour cela, on doit rajouter une **table de correspondances d'historiques**, qui permet de dire où se trouve l'historique associé à chaque index. Cette table de correspondances (index → historique par index) a bien sûr une taille finie. En somme, le nombre d'entrées de cette table limite le nombre d'index (souvent des instructions)

gérées en même temps. Mais par contre, pour chaque instruction, la taille de l'historique des défauts de cache est variable.



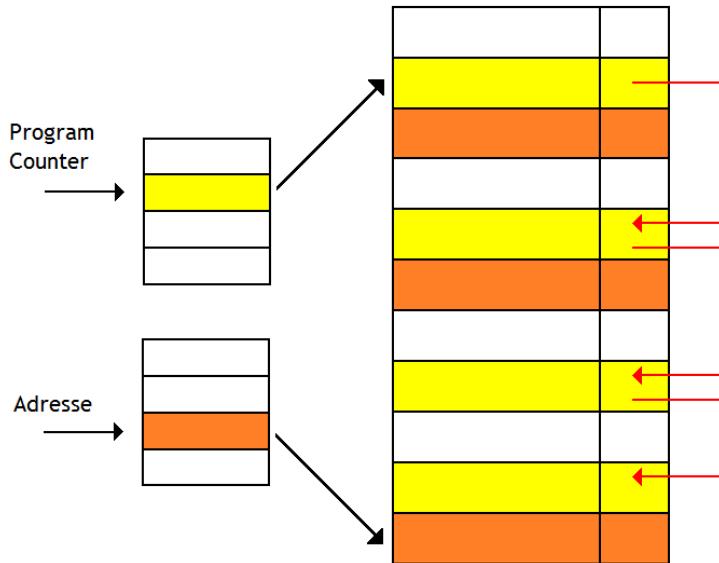
La table de correspondances et l'historique global sont couplés avec un **circuit de prédition**, qui peut utiliser chaque historique pour faire ces prédictions. Celui-ci peut aussi bien utiliser la totalité de l'historique global, que les historiques dédiés à un index. Faire une prévision est simple demande d'accéder à la table de correspondances avec l'index adéquat : l'adresse lue ou écrite, le program counter de l'instruction d'accès mémoire, la distance entre cette adresse et la précédente, etc. Cela va alors sélectionner une liste dans l'historique global, qui sera parcourue de proche en proche par le circuit de prévision, qui déterminera l'adresse à précharger en fonction de l'historique stocké dans la liste. Dans certains cas, l'historique global est aussi parcouru par le circuit de prévision, mais c'est plus rare.

Ce tampon d'historique global permet d'implémenter un algorithme de Markov assez simplement : il suffit que la table d'index mémorise une correspondance adresse → début de liste. Ainsi, pour chaque adresse, on associera la liste d'adresses suivantes possibles, classées suivant leur probabilité. L'adresse au début de la liste sera la plus probable, tandis que celle de la fin sera la moins probable. Même chose pour le préchargement par distance : il suffit que l'index soit la distance entre adresse précédemment accédée et adresse couramment accédée. Dans ce cas, la liste des entrées mémorisera la suite de distances qui correspondent. L'implémentation d'un préchargement par enjambées est aussi possible, mais assez complexe. Mais de nouveaux algorithmes sont aussi possibles.

Variante du tampon d'historique global

Des variantes du tampon d'historique global ont été inventées. On pourrait citer celle qui ajoute, en plus de l'historique global, des historiques locaux sous la forme de mémoires FIFO qui mémorisent les derniers accès effectués par une instruction. Plus précisément, si on dispose de n historiques locaux, chacun de ces n historiques mémorise l'historique des n dernières instructions d'accès mémoire les plus récentes. D'autres variantes, et notamment celle du **cache d'accès aux données**, ont ajouté une seconde table d'index :

- la première table d'index prend en entrée le program counter de l'instruction à l'origine du défaut de cache ou de l'accès mémoire ;
- la seconde prend en entrée l'adresse à lire ou écrire.

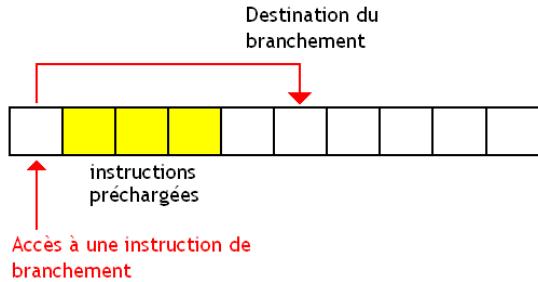


Préchargement d'instructions

Certains processeurs utilisent un préfetcher séparé pour les instructions, chose qui fonctionne à la perfection si le cache d'instruction est séparé des caches de données. Les techniques utilisées par les préfetchers d'instructions sont multiples et nombreux sont ceux qui réutilisent les techniques vues précédemment. Le préchargement séquentiel est notamment utilisé sur certains préfetchers relativement simples, de même que les préfetchers de Markov. Mais certaines techniques sont spécifiques au préchargement des instructions. Il faut dire que les branchements sont une spécificité du flux d'instruction, qui doit être prise en compte par le préfetcher pour obtenir un résultat optimal.

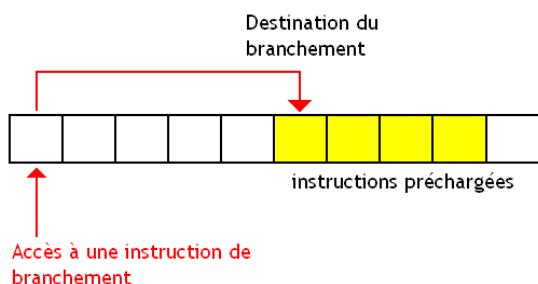
Préchargement séquentiel, le retour !

Le préchargement séquentiel est parfaitement adapté au préchargement des instructions d'un programme, vu que ses instructions sont placées les unes après les autres en mémoire. Sur certains processeurs, cette technique est utilisée non seulement pour charger des instructions dans le cache, mais aussi pour charger à l'avance certaines instructions dans le séquenceur. Sur ces processeurs, l'unité de chargement et de décodage sont séparées par une petite mémoire tapon de type FIFO : le **tampon d'instructions** (instruction buffer). En utilisant du préchargement séquentiel, on peut précharger des instructions dans le tampon d'instructions à l'avance, permettant ainsi de masquer certains accès au cache ou à la mémoire assez longs. Lorsqu'un branchement est décodé, ce tampon d'instructions est totalement vidé de son contenu. Mais un prefetcher purement séquentiel gère mal les branchements.



Target line prefetching

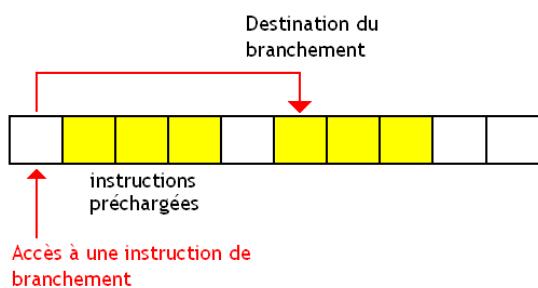
Il serait judicieux de trouver des moyens qui permettent de limiter la casse. Pour ce faire, il faudrait trouver un moyen de savoir où va nous faire atterrir notre branchement, de connaître l'adresse de destination. Néanmoins, le processeur peut supposer que l'adresse de destination est fixe : il suffit de s'en souvenir pour la prochaine fois. C'est ce qu'on appelle le **target line prefetching**.



Pour implémenter cette technique, le prefetcher incorpore un cache pour stocker les adresses de destination des branchements déjà rencontrés. Plus précisément, ce cache contient les correspondances entre une ligne de cache, et la ligne de cache à charger à la suite de celle-ci. Pour plus d'efficacité, certains processeurs ne stockent pas les correspondances entre lignes de cache consécutives. Si jamais deux lignes de cache sont consécutives, on fera alors face à un défaut de cache dans cette mémoire qui stocke nos correspondances : le prefetcher utilisera alors automatiquement un préchargement séquentiel. Ainsi, la table de correspondances est remplie uniquement avec des correspondances utiles.

Préchargement du mauvais chemin

On peut améliorer la technique vue juste avant pour lui permettre de s'adapter au mieux aux branchements conditionnels. En plus de charger les instructions correspondant à un branchement pris, on peut aussi charger les instructions situées juste après le branchement. Comme ça, si le branchement n'est pas pris, les instructions suivantes seront disponibles quand même. On appelle cette technique le **préchargement du mauvais chemin** (wrong path prefetching).



Pollution de cache

Le seul problème avec ces techniques de préchargement, c'est que le prefetcher peut se tromper et précharger des données inutilement dans le cache. On peut se retrouver avec quelques lignes de cache qui contiendront des données qui ne servent à rien, et qui auront éjecté des données potentiellement utiles du cache. C'est le phénomène de pollution de cache. Il va de soi que limiter au maximum cette pollution du cache est une nécessité si on veut tirer parti au maximum de notre mémoire cache, reste à savoir comment.

Dirty bit

Avec la première solution, la donnée chargée inutilement sera sélectionnée pour remplacement lors du prochain défaut de cache. Si le cache utilise un algorithme de sélection des lignes de cache de type LRU, on peut la mettre directement dans l'état « utilisée la moins récemment », ou « très peu utilisée ».

Duel d'ensembles

Des chercheurs ont inventé des techniques encore plus complexes, dont la plus connue est le duel d'ensembles (set dueling). Dans leurs travaux, ils utilisent un cache associatif à plusieurs voies. Ces voies sont réparties en deux groupes : statiques ou dynamiques. Les voies statiques ont une politique de remplacement fixée une fois pour toutes :

- dans certaines voies statiques, toute ligne chargée depuis la mémoire est considérée comme la plus récemment utilisée ;

- dans les autres voies statiques, toute ligne chargée depuis la mémoire est considérée comme la moins récemment utilisée.

Les voies restantes choisissent dynamiquement d'utiliser ou non l'optimisation vue plus haut (la ligne chargée est considérée comme la moins récemment utilisée) suivant les voies statiques qui ont le plus de défauts de cache : si les voies qui appliquent l'optimisation ont plus de défauts de cache que les autres, on ne l'utilise pas et inversement. Il suffit d'utiliser un simple compteur incrémenté ou décrémenté lors d'un défaut de cache dans une voie utilisant ou non l'optimisation.

Cache spécialisé pour le précharge

L'autre solution est de précharger les données non pas dans le cache, mais dans une mémoire séparée dédiée au précharge : un tampon de flux (stream buffer). Si jamais un défaut de cache a lieu, on regarde si la ligne de cache à lire ou écrire est dans le tampon de flux : on la rapatrie dans le cache si c'est le cas. Si ce n'est pas le cas, c'est que le tampon de flux contient des données préchargées à tort : le tampon de flux est totalement vidé, et on va chercher la donnée en mémoire. Dans le cas où le précharge utilisé est un simple précharge séquentiel, le tampon de flux est une simple mémoire FIFO.

Filtrage de cache

Une autre solution consiste à détecter les requêtes de précharge inutiles en sortie du prefetcher. Entre les circuits d'adressage de la mémoire (ou les niveaux de cache inférieurs), et le prefetcher, on ajoute un circuit de filtrage, qui détecte les requêtes de précharge visiblement inutiles et contreproductives. Les algorithmes utilisés par ce circuit de filtrage de cache varient considérablement suivant le processeur, et les travaux de recherche sur le sujet sont nombreux.

Quand précharger ?

Une problématique importante est de savoir quand précharger des données. Si on précharge des données trop tard ou trop tôt, le résultat n'est pas optimal. Pour résoudre ce problème, il existe diverses solutions :

- le précharge sur événement ;
- le précharge par anticipation du program counter ;
- le précharge par prédiction.

Précharge sur événement

Une solution consiste à précharger quand certains événements spéciaux ont lieu. Par exemple, on peut précharger à chaque défaut de cache, à chaque accès mémoire, lors de l'exécution d'un branchement (pour le précharge des instructions), etc. De nos jours, le précharge est initié par les accès mémoire, de diverses manières.

La première solution consiste à précharger le bloc suivant de manière systématique : un précharge a lieu lors de chaque accès mémoire.

Seconde solution : ne précharger que lors d'un défaut de cache. Ainsi, si j'ai un défaut de cache qui me force à charger le bloc B dans le cache, le prefetcher chargera le bloc immédiatement suivant avec.

Troisième solution : à chaque accès à un bloc de mémoire dans le cache, on charge le bloc de mémoire immédiatement suivant. Pour cela, on doit mémoriser quelle est la dernière ligne de cache qui a été accédée. Cela se fait en marquant chaque ligne de cache avec un bit spécial, qui indique si cette ligne a été accédée lors du dernier cycle d'horloge. Ce bit vaut 1 si c'est le cas, et vaut 0 sinon. Ce bit est automatiquement mis à zéro au bout d'un certain temps (typiquement au cycle d'horloge suivant). Le prefetcher se contentera alors de charger le bloc qui suit la ligne de cache dont le bit vaut 1.

Chargé lors d'un <i>cache miss</i>	1	 Lecture ou écriture
Bloc préchargé	0	
	0	
	0	
...	0	

Un peu plus tard...

Chargé lors d'un <i>cache miss</i>	0	 Lecture ou écriture
Bloc préchargé	1	
Bloc préchargé	0	
	0	
...	0	

Précharge par anticipation du program counter

Une seconde technique détermine les adresses à précharger en prenant de l'avance sur les instructions en cours d'exécution : quand le processeur est bloqué par un accès mémoire, l'unité de précharge anticipe les prochaines instructions chargées. Le processeur pourra, à partir de ces prédictions, déterminer les adresses lues ou écrites par ces instructions anticipées.

La première technique se base sur un lookahead program counter initialisé avec le program counter lors d'un défaut de cache. Il est incrémenté à chaque cycle (et les branchements sont prédictifs) : ce lookahead program counter est mis à jour comme si l'exécution du programme se poursuivait, mais le reste du processeur est mis en attente. Les adresses fournies à chaque cycle par le lookahead program counter sont alors envoyées aux unités de précharge pour qu'elles fassent leur travail. On peut aussi adapter cette technique pour que le lookahead program counter passe non d'une instruction à la prochaine à chaque cycle, mais d'un branchement au suivant.

On peut aussi citer le précharge anticipé (runahead prefetching). Cette technique est utile dans le cas où un processeur doit arrêter l'exécution de son programme parce que celui-ci attend une donnée en provenance de la mémoire. En clair, cette technique est utile si un défaut de cache a eu lieu et que le processeur n'est pas conçu pour pouvoir continuer ses calculs dans de telles conditions (pas de caches non bloquants, pas d'exécution dans le désordre, etc.).

Dans un cas pareil, le processeur est censé stopper l'exécution de son programme. Mais à la place, on va continuer l'exécution des instructions suivantes de façon spéculative : on les exécute, même si on n'est pas censé avoir ce droit. Si elles accèdent à la mémoire, on laisse ces accès

s'exécuter (sauf en écriture pour éviter de modifier des données alors qu'on n'aurait pas dû) : cela permettra d'effectuer des accès en avance et donc de précharger les données qu'elles manipulent. On continue ainsi jusqu'à ce que l'instruction qui a stoppé tout le processeur ait enfin reçu sa donnée.

Le seul truc, c'est que tout doit se passer comme si ces instructions exécutées en avance n'avaient jamais eu lieu. Dans le cas contraire, on a peut-être exécuté des instructions qu'on n'aurait peut-être pas dû, et cela peut avoir modifié des registres un peu trop tôt, ou mis à jour des bits du registre d'état qui n'auraient pas dû être modifiés ainsi. Il faut donc trouver un moyen de remettre le processeur tel qu'il était quand le défaut de cache a eu lieu. Pour cela, le processeur doit sauvegarder les registres du processeur avant d'exécuter spéculativement les instructions suivantes, et les restaurer une fois le tout terminé.

Qui plus est, il vaut mieux éviter que ces instructions exécutées en avance puissent modifier l'état de la mémoire : imaginez qu'une instruction modifie une ligne de cache alors qu'elle n'aurait pas dû le faire ! Pour cela, on interdit à ces instructions d'écrire dans la mémoire.

Les processeurs qui utilisent ce genre de techniques sont redoutablement rares à l'heure où j'écris ce tutoriel. On trouve pourtant quelques articles de recherche sur le sujet, et quelques universitaires travaillent dessus. Mais aucun processeur ne précharge ses données ainsi. Il y a bien le processeur Rock de la compagnie Sun, qui aurait pu faire l'affaire, mais celui-ci a été annulé au dernier moment.

Préchargement par prédiction

Certains prefetchers avancés essaient de déduire le moment adéquat pour précharger en se basant sur l'historique des accès précédents : ils en déduisent des statistiques qui permettent de savoir quand précharger. Par exemple, ils peuvent calculer le temps d'accès moyen entre un accès mémoire et un préchargement, et armer des chronomètres pour initialiser le préchargement en temps voulu.

Le pipeline

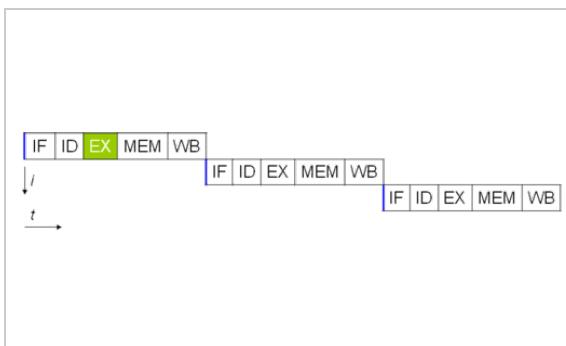
De manière générale, le temps d'exécution d'une instruction dépend du CPI, le nombre moyen de cycles d'horloge par instruction et de la durée P d'un cycle d'horloge. En conséquence, il existe deux solutions pour rendre les instructions plus rapides : diminuer le CPI en améliorant la conception du processeur et augmenter la fréquence. Avec le temps, il est devenu de plus en plus difficile de monter en fréquence, les contraintes de consommation énergétique se faisant de plus en plus lourdes. Il ne restait plus qu'une solution : diminuer le CPI. Les concepteurs de processeurs ont alors cherché à optimiser au mieux les instructions les plus utilisées et se sont plus ou moins heurtés à un mur. Il est devenu évident au fil du temps qu'il fallait réfléchir hors du cadre et trouver des solutions innovantes, ne ressemblant à rien de connu. Ils ont fini par trouver une solution assez incroyable : exécuter plusieurs instructions en même temps ! Pour cela, il a bien fallu trouver quelques solutions diverses et variées, dont le **pipeline** est la plus importante. Pour expliquer en quoi il consiste, il va falloir faire un petit rappel sur les différentes étapes d'une instruction.

Le pipeline : rien à voir avec un quelconque tuyau à pétrole !

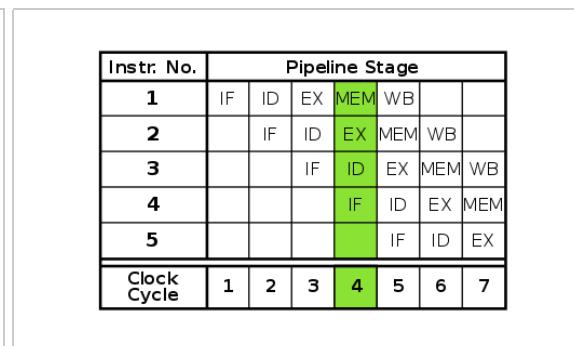
Ceux qui se souviennent du chapitre sur la micro-architecture d'un processeur savent qu'une instruction est exécutée en plusieurs étapes bien distinctes, dépendant du processeur, du mode d'adressage, ou des manipulations qu'elle doit effectuer. Dans ce qui va suivre, nous allons utiliser une organisation relativement générale, où chaque instruction passe par les étapes suivantes :

- PC : mise à jour du program counter ;
- chargement : on charge notre instruction depuis la mémoire ;
- décodage : décodage de l'instruction ;
- chargement d'opérandes : si besoin, nos opérandes sont lus depuis la mémoire ou les registres ;
- exécution: on exécute l'instruction ;
- accès mémoire : accès à la mémoire RAM ;
- enregistrement : si besoin, le résultat de l'instruction est enregistré en mémoire.

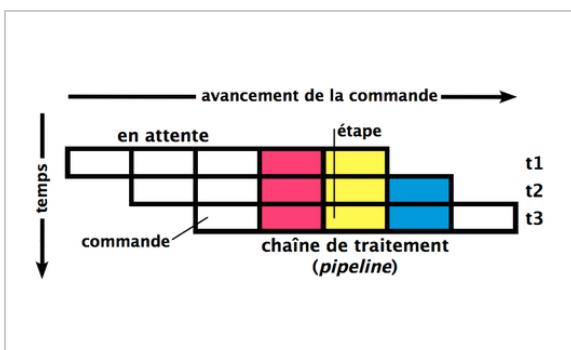
Sans pipeline, on doit attendre qu'une instruction soit finie pour exécuter la suivante. Avec un pipeline, on peut commencer à exécuter une nouvelle instruction sans attendre que la précédente soit terminée. Par exemple, on pourrait charger la prochaine instruction pendant que l'instruction en cours d'exécution en est à l'étape d'exécution. Après tout, ces deux étapes sont complètement indépendantes et utilisent des circuits séparés. Le principe du pipeline est simple : exécuter plusieurs instructions différentes, chacune étant à une étape différente des autres. Chaque instruction passe progressivement d'une étape à la suivante dans ce pipeline et on charge une nouvelle instruction par cycle dans le premier étage. Le nombre total d'étapes nécessaires pour effectuer une instruction (et donc le nombre d'étages du pipeline) est appelé la **profondeur du pipeline**. Il correspond au nombre maximal théorique d'instructions exécutées en même temps dans le pipeline.



Exécution de trois instructions sans pipeline.



Exécution de cinq instructions dans un pipeline de 5 étages.

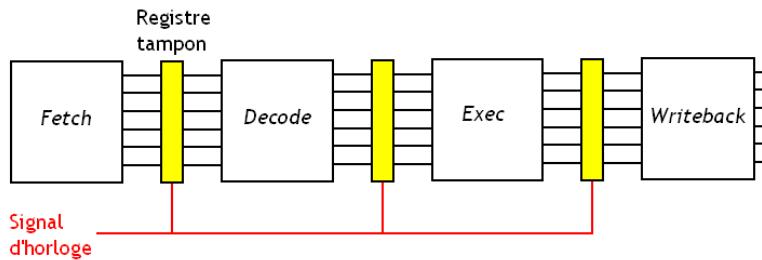


Pipeline : principe.

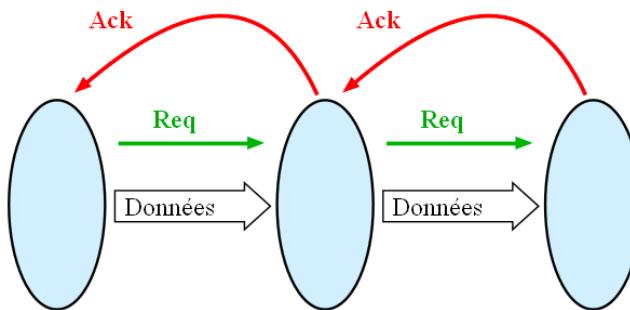
Isoler les étages du pipeline

Concevoir un processeur avec un pipeline nécessite quelques modifications de l'architecture de notre processeur. Tout d'abord, chaque étape d'une instruction doit s'exécuter indépendamment des autres, ce qui signifie utiliser des circuits indépendants pour chaque étape. Il est donc impossible de réutiliser un circuit dans plusieurs étapes, comme on le fait dans certains processeurs sans pipeline. Par exemple, sur un processeur sans pipeline, l'additionneur de l'ALU peut être utilisé pour mettre à jour le program counter lors du chargement, calculer des adresses lors d'un accès mémoire, les additions, etc. Mais sur un processeur doté d'un pipeline, on ne peut pas se le permettre, ce qui fait que chaque étape doit utiliser son propre additionneur. De même, l'étage de chargement peut entrer en conflit avec d'autres étages pour l'accès à la mémoire ou au cache, notamment pour les instructions d'accès mémoire. On peut résoudre ce conflit entre étage de chargement et étage d'accès mémoire en dupliquant le cache L1 en un cache d'instructions et un cache de données. Et ce principe est général : il est important de séparer les circuits en charge de chaque étape. Chaque circuit dédié à une étape est appelé un **étage du pipeline**.

De plus, la plupart des pipelines intercalent des registres entre les étages, pour les isoler.

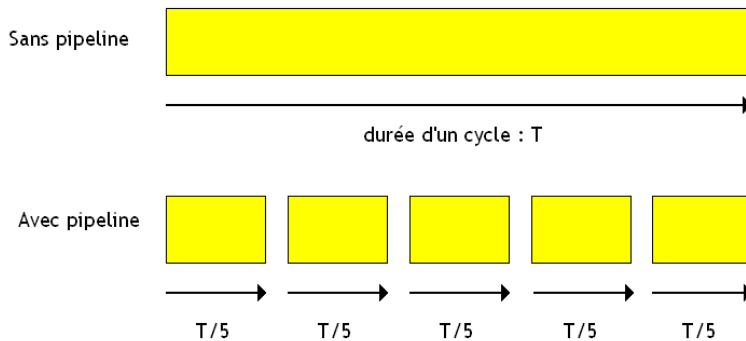


Ces transferts entre registres et étages peuvent être synchronisés par une horloge, mais ce n'est pas systématique. Si le pipeline est synchronisé sur l'horloge du processeur, on parle de **pipeline synchrone**. Chaque étage met un cycle d'horloge pour effectuer son travail, à savoir, lire le contenu du registre qui le relie à l'étape précédente et déduire le résultat à écrire dans le registre suivant. Ce sont ces pipelines que l'on trouve dans les processeurs Intel et AMD les plus récents. Sur d'autres pipelines, il n'y a pas d'horloge pour synchroniser les transferts entre étages, qui se font via un « bus » asynchrone. On parle de **pipeline asynchrone**.



Le paradoxe des pipelines

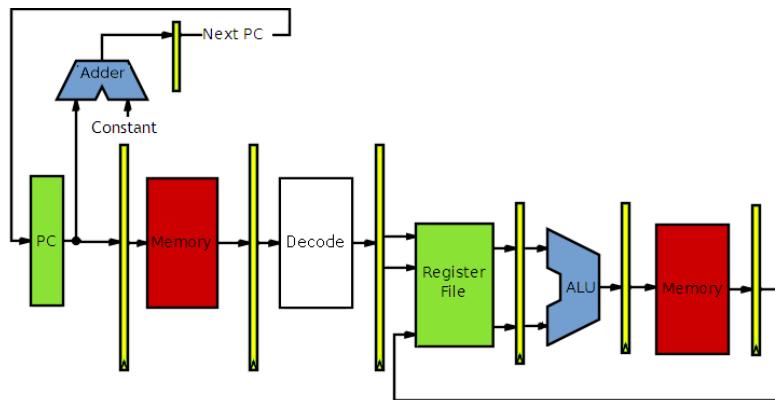
Revenons un peu sur les pipelines synchrones. Sur ceux-ci, un étage fait son travail en un cycle d'horloge. Sur un pipeline à N étages, une instruction met N cycles d'horloge à s'exécuter, contre un seul cycle sur un processeur sans aucun pipeline. Cela a tendance à annuler l'augmentation du CPI : à quoi bon exécuter N instructions simultanément, si chaque instruction prend N fois plus de temps ? Le temps d'exécution d'un paquet d'instructions n'est pas censé changer ! Sauf que ce raisonnement oublie un paramètre important : un étage de pipeline a un temps de propagation plus petit qu'un processeur complet; ce qui permet d'en augmenter la fréquence. Cela permet donc de multiplier la fréquence par un coefficient plus ou moins proportionnel aux nombres d'étages.



En réalité, certains étages possèdent un chemin critique plus long que d'autres. On est alors obligé de se caler sur l'étage le plus lent, ce qui réduit quelque peu le gain. La durée d'un cycle d'horloge devra être supérieure au temps de propagation de l'étage le plus fourni en portes logiques. Mais dans tous les cas, l'usage d'un pipeline permet au mieux de multiplier la fréquence par le nombre d'étages. Cela a poussé certains fabricants de processeurs à créer des processeurs ayant un nombre d'étages assez élevé pour les faire fonctionner à très haute fréquence. Par exemple, c'est ce qu'a fait Intel avec le Pentium 4, dont le pipeline faisait 20 étages pour les Pentium 4 basés sur l'architecture Willamette et Northwood, et 31 étages pour ceux basés sur l'architecture Prescott et Cedar Mill.

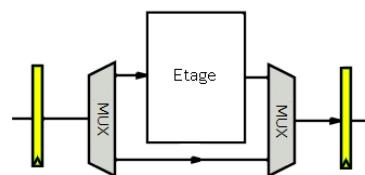
Pipeline de longueur fixe

Découper un processeur en pipeline peut se faire de différentes manières, le nombre et la fonction des étages variant fortement d'un processeur à l'autre. Nous allons prendre comme exemple le pipeline vu plus haut, composé de sept étages. L'étage de PC va gérer le program counter. L'étage de chargement va devoir utiliser l'interface de communication avec la mémoire. L'étage de décodage contiendra l'unité de décodage d'instruction. L'étage de lecture de registre contiendra le banc de registres. L'étage d'exécution contiendra l'ALU, l'étage d'accès mémoire aura besoin de l'interface avec la mémoire, et l'étage d'enregistrement aura besoin des ports d'écriture du banc de registres. Naivement, on peut être tenté de relier l'ensemble de cette façon.

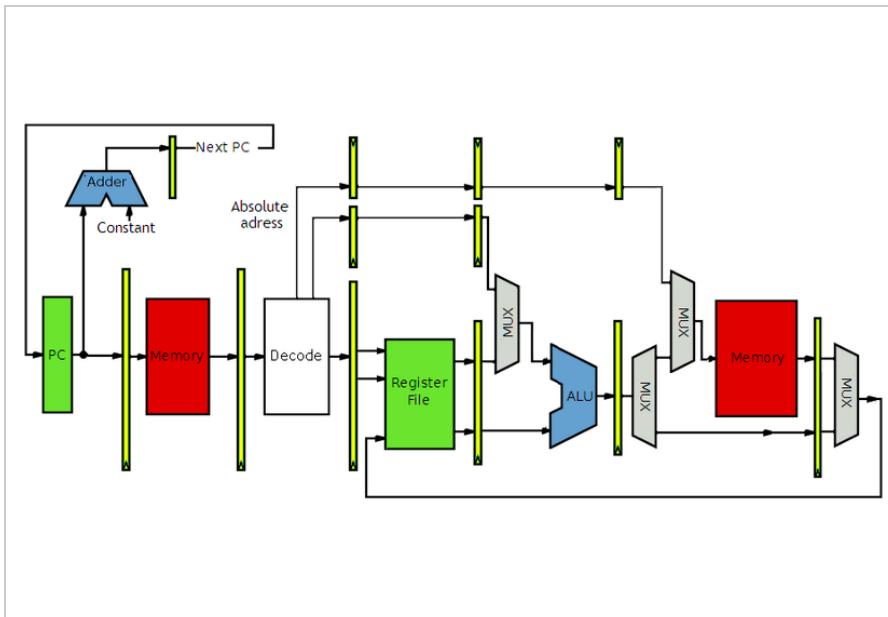


Chemin de données

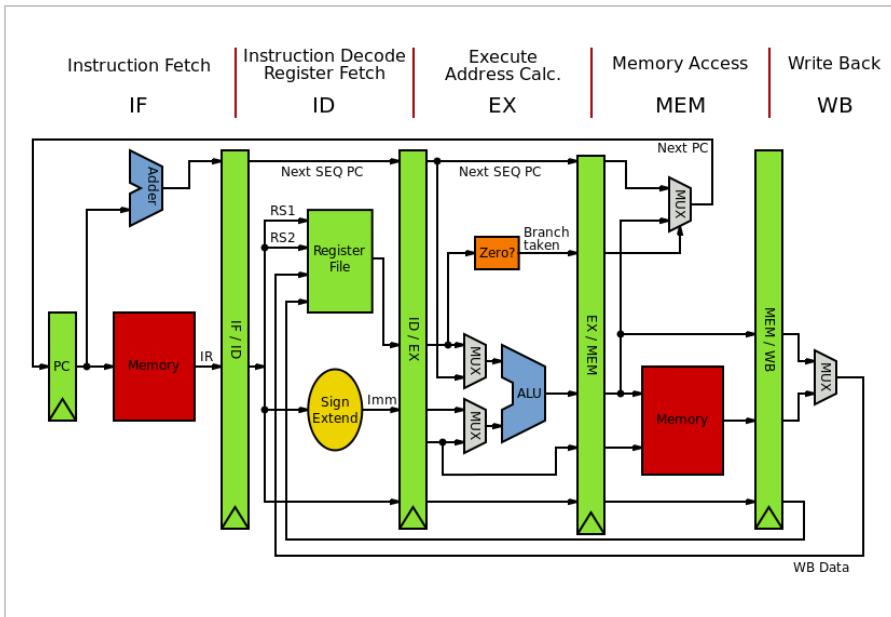
Mais toutes les instructions n'ont pas besoin d'accéder à la mémoire, tout comme certaines instructions n'ont pas à utiliser l'ALU ou lire des registres. Par exemple, certaines instructions n'ont pas besoin d'accéder à la RAM : on doit donc court-circuiter l'étage d'accès mémoire. De même, l'ALU aussi doit être court-circuitée pour les opérations qui ne font pas de calculs. Certains étages sont « facultatifs » : l'instruction doit passer par ces étages, mais ceux-ci ne doivent rien faire, être rendus inactifs. Pour inactiver ces circuits, il suffit juste que ceux-ci puissent effectuer une instruction NOP, qui ne fait que recopier l'entrée sur la sortie. Pour les circuits qui ne s'inactivent pas facilement, on peut les court-circuiter en utilisant diverses techniques, la plus simple d'entre elle consistant à utiliser des multiplexeurs.



La lecture dans les registres peut être court-circuitée lors de l'utilisation de certains modes d'adressage. C'est notamment le cas lors de l'usage du mode d'adressage absolu. Pour le gérer, on doit envoyer l'adresse, fournie par l'unité de décodage, sur l'entrée d'adresse de l'interface de communication avec la mémoire. Le principe est le même avec le mode d'adressage immédiat, sauf que l'on envoie une constante sur une entrée de l'ALU. On peut aller relativement loin comme cela. La seconde illustration montre ce que peut donner un pipeline MIPS à 5 étages qui gère les modes d'adressage les plus courants.



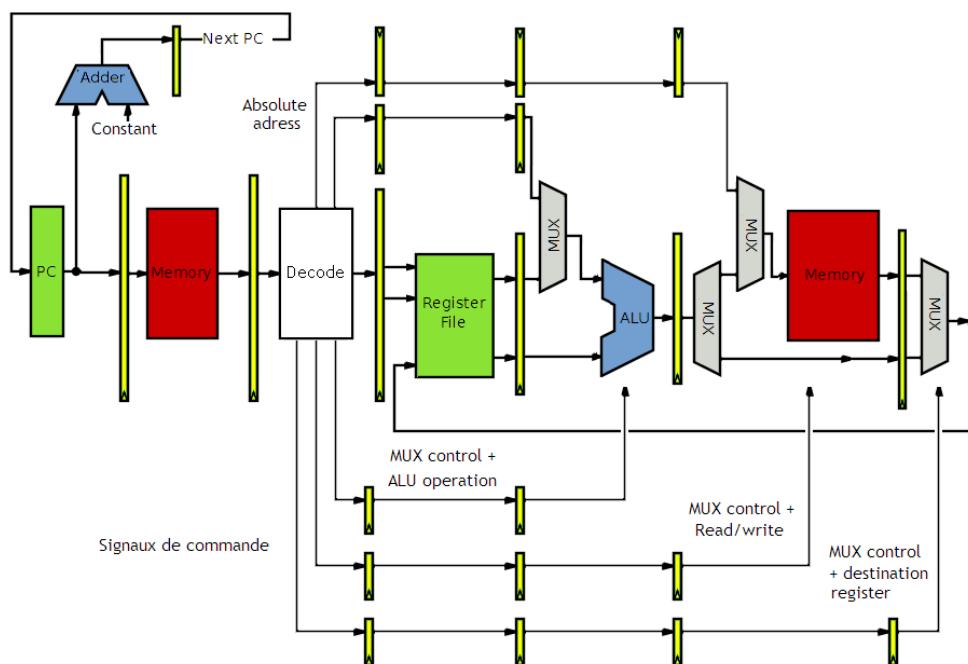
Pipeline à 7 étages, avec mode d'adressage immédiat et absolu gérés.



MIPS Architecture (Pipelined)

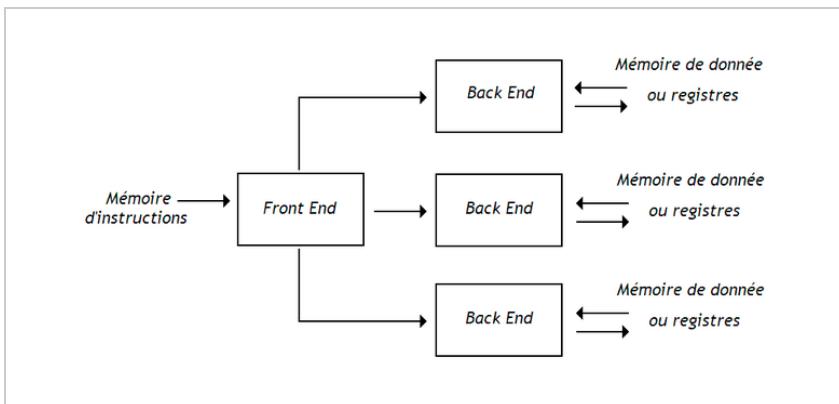
Signaux de commande

Les signaux de commande qui servent à configurer le chemin de données sont générés par l'unité de décodage, dans la second étage du pipeline. Comment faire pour que ces signaux de commande traversent le pipeline ? Relier directement les sorties de l'unité de décodage aux circuits incriminés ne marcherait pas : les signaux de commande arriveraient immédiatement aux circuits, alors que l'instruction n'a pas encore atteint ces étages ! La réponse consiste à faire passer ces signaux de commande d'un étage à l'autre en utilisant des registres.

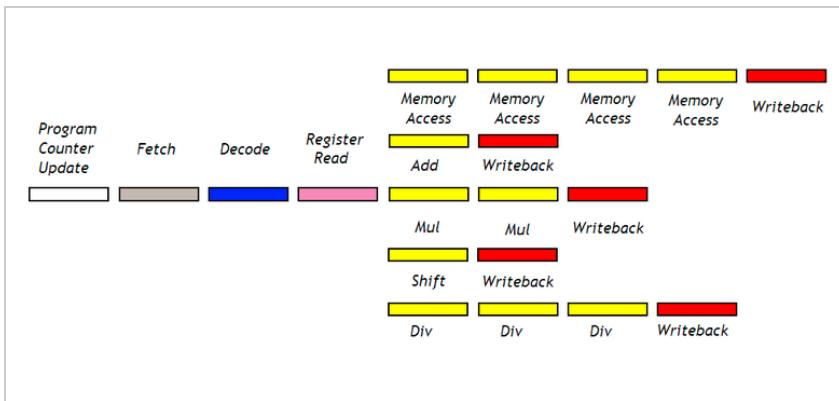


Pipeline de longueur variable

Avec le pipeline vu plus haut, toutes les instructions se voient attribuer le même nombre d'étages, les étages inutiles étant court-circuités ou inactivés. Sur un processeur non pipeliné, on aurait pu éviter ces étapes inutiles en faisant varier le nombre de micro-opérations par instruction, certaines instructions pouvant prendre 7 cycles, d'autres 9, d'autres 25, etc. Il est possible de faire la même chose sur les processeurs pipelinés, dans une certaine limite, en utilisant plusieurs pipelines de longueurs différentes. Avec cette technique, le pipeline du processeur est décomposé en plusieurs parties. La première partie, l'**amont** (front end), prend en charge les étages communs à toutes les instructions : la mise à jour du program counter, le chargement, l'étage de décodage, etc. Cet amont est suivi par le chemin de données, découpé en plusieurs unités adaptées à un certain type d'instructions, chacune formant ce qu'on appelle un **aval** (back end). Un aval est spécialisé dans une classe d'instructions : un pour les instructions arithmétiques et logiques, un autre pour les opérations d'accès mémoire, un autre pour les instructions d'échange de données entre registres, et ainsi de suite. La longueur de l'aval peut varier suivant le type d'instructions.



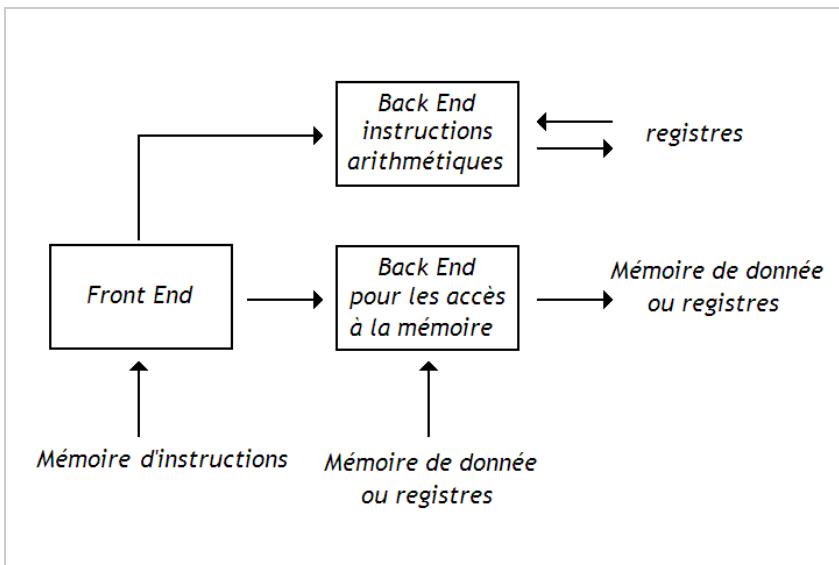
Pipeline avec un aval et un amont (back-end et front-end).



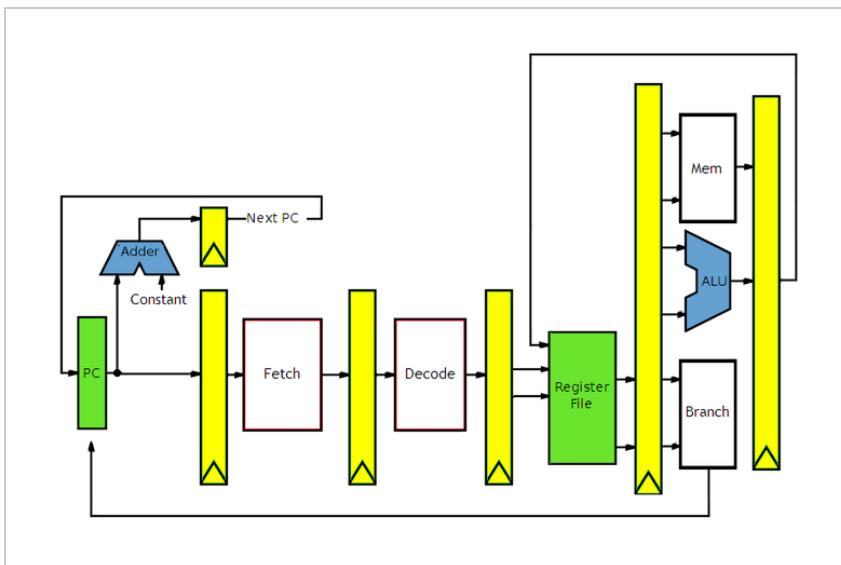
Pipeline avec un nombre variable d'étages par instructions.

Processeur load-store

Les processeurs load-store peuvent se contenter de deux avals : un pour les accès mémoire, et un pour les calculs. Il suffit alors de les placer en parallèle et de rediriger l'instruction dans l'unité qui la prend en charge. Certains processeurs utilisent un aval pour les lectures et un autre pour les écritures. De même, il est possible de scinder l'aval pour les instructions en plusieurs avals séparés pour l'addition, la multiplication, la division, etc. On peut aussi utiliser une unité de calcul séparée pour les instructions de tests et branchements.



Pipeline à deux aval de type load-store.



Pipeline avec un aval pour les instructions arithmétiques, un aval pour les accès mémoire et un aval pour les branchements.

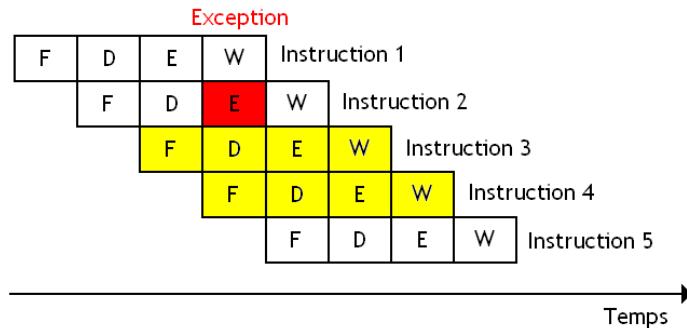
Mais la séparation des avals n'est pas optimale pour les instructions qui se pipelinent mal, auxquelles il est difficile de leur créer un aval dédié. C'est notamment le cas de la division, dont les unités de calcul ne sont jamais totalement pipelinées, voire pas du tout. Il n'est ainsi pas rare d'avoir à gérer des unités de calcul dont chaque étage peut prendre deux à trois cycles pour s'exécuter : il faut attendre un certain nombre de cycles avant d'envoyer une nouvelle instruction dans l'aval.

Processeurs non load-store

Les modes d'adressage complexes se marient mal avec un pipeline, notamment pour les modes d'adresses qui permettent plusieurs accès mémoire par instructions. La seule solution est de découper ces instructions machines en sous-instructions qui seront alors chargées dans notre pipeline. Ces instructions complexes sont transformées par le décodeur d'instructions en une suite de micro-instructions directement exécutables par le pipeline. Il va de soi que cette organisation complique pas mal le fonctionnement du séquenceur.

Interruptions et pipeline

Sur un processeur purement séquentiel, ces interruptions et exceptions ne posent aucun problème. Mais si on rajoute un pipeline, les choses changent : cela pose des soucis si une instruction génère une exception ou interruption dans le pipeline. Avant que l'exception n'ait été détectée, le processeur a chargé des instructions dans le pipeline alors qu'elles n'auraient pas dû l'être. En effet, ces instructions sont placées après l'instruction à l'origine de l'exception dans l'ordre du programme. Logiquement, elles n'auraient pas dû être exécutées, vu que l'exception est censée avoir fait brancher notre processeur autre part.



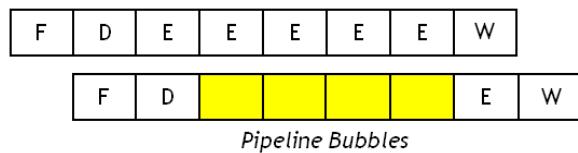
Une solution consiste à placer des instructions inutiles après une instruction susceptible de générer une exception matérielle ou une interruption. Dans l'exemple, l'exception est prise en compte avec deux cycles d'horloge de retard. Il suffit donc de placer deux instructions inutiles juste après celle pouvant causer une exception. Le seul problème, c'est que le nombre d'instructions inutiles à ajouter dépend du pipeline du processeur, ce qui peut poser des problèmes de compatibilité. Au lieu d'insérer ces instructions directement dans le programme, les concepteurs de processeur ont décidé de faire en sorte que le processeur se charge lui-même de gérer exceptions et interruptions correctement : le processeur annule les instructions chargées à tort et remet le pipeline en ordre.

Achèvement dans l'ordre

Ces techniques de gestion des exceptions sont spéculatives : le processeur va spéculer que les instructions qu'il vient de charger ne lèvent aucune exception. Le processeur continuera à exécuter ses instructions, ce qui évite des bulles de pipeline inutiles si la spéulation tombe juste. Mais si jamais cette prédiction se révèle fausse, il va devoir annuler les instructions exécutées suite à cette erreur de spéulation. Au final, ce pari est souvent gagnant : les exceptions et interruptions sont très rares, même si beaucoup d'instructions peuvent en lever.

Ordre des écritures

Pour implémenter cette technique, le processeur garantit que les instructions chargées ne feront aucun mal si jamais la spéulation est fausse : leurs écritures en RAM, dans les registres, ou dans le program counter doivent pouvoir être empêchées et annulées. Pour régler ce problème, on doit garantir que ces écritures se fassent dans l'ordre du programme. Avec un pipeline de longueur fixe, il n'y a rien à faire, contrairement à ceux de longueur variable. Prenons cet exemple : je charge deux instructions l'une à la suite de l'autre dans mon pipeline, la première prend 8 cycles pour s'exécuter, tandis que la seconde en prend 4. On a beau avoir démarré les instructions dans l'ordre, la première enregistre son résultat avant la seconde : si la première instruction lève une exception, les résultats de la seconde auront déjà été enregistrés dans les registres. La solution consiste à ajouter des bulles de pipeline pour retarder certaines instructions. Si une instruction est trop rapide et risque d'écrire son résultat avant ses prédecesseurs, il suffit simplement de la retarder avec le bon nombre de cycles.



Pour ce faire, on utilise un circuit spécial : le registre de décalage de résultat (result shift register), un registre à décalage qui contient autant de bits qu'il y a d'étages dans notre pipeline. Chaque bit est attribué à un étage et signifie que l'étage en question est utilisé par une instruction. À chaque cycle d'horloge, ce registre est décalé d'un cran vers la droite, pour simuler la progression des instructions dans le pipeline. Lorsque l'unité de décodage démarre une instruction, elle vérifie le nombre de cycles que va prendre l'instruction pour s'exécuter. Pour une instruction de n cycles, elle va vérifier le n-ième bit de ce registre. S'il est à 1, une autre instruction est déjà en cours pour cet étage, et l'instruction est mise en attente. Si ce bit est à 0, l'unité va le placer à 1, ainsi que tous les bits précédents : l'instruction s'exécutera.

Récupération après spéulation

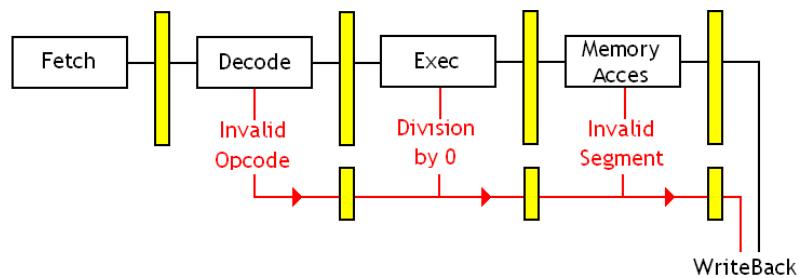
Les exceptions sont détectées dans le pipeline, quand elles sont levées par un circuit. Mais elles ne sont prises en compte qu'au moment d'enregistrer les données en mémoire, dans l'étage d'enregistrement. Pour comprendre pourquoi, imaginez que dans l'exemple du dessus, les deux instructions lèvent une exception à des étages différents. Quelle exception traiter en premier ? Il va de soi qu'on doit traiter ces exceptions dans l'ordre du programme, donc c'est celle de la première instruction qui doit être traitée. Traiter les exceptions à la fin du pipeline permet de traiter les exceptions dans leur ordre d'occurrence dans le programme.

Annulation des écritures fautives

Pour interdire les modifications des registres et de la mémoire en cas d'exception, on doit rajouter un étage dans le pipeline, qui sera chargé d'enregistrer les données dans les registres et la mémoire : si une exception a lieu, il suffit de ne pas enregistrer les résultats des instructions suivantes dans les registres, jusqu'à ce que toutes les instructions fautives aient quitté le pipeline.

Prise en compte des exceptions

Tout étage fournit à chaque cycle un indicateur d'exception, un groupe de quelques bits qui indiquent si une exception a eu lieu et laquelle le cas échéant. Ces bits sont propagés dans le pipeline, et passent à l'étage suivant à chaque cycle. Une fois arrivé à l'étage d'enregistrement, un circuit combinatoire vérifie ces bits (pour voir si une exception a été levée), et autorise ou interdit l'écriture dans les registres ou la mémoire en cas d'exception.



Achèvement dans le désordre

Il existe d'autres solutions pour maintenir l'ordre des écritures, sans avoir besoin de registre à décalage. Toutes ces techniques, hormis la sauvegarde de registres, demandent d'ajouter un étage de pipeline pour remettre les écritures dans l'ordre du programme. Celui-ci est inséré entre l'étage d'exécution et celui d'enregistrement. Voici la liste de ces techniques :

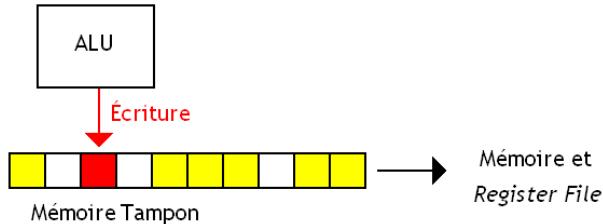
- la sauvegarde de registres ;
- un tampon de réordonnancement ;
- un tampon d'historique ;
- un banc de registres futurs ;
- autre chose.

Sauvegarde de registres

La sauvegarde de registres (register checkpointing) consiste à faire des sauvegardes régulières des registres, et récupérer cette sauvegarde lors d'une exception. Mais sauvegarder les registres du processeur prend du temps, ce qui fait que cette solution n'est que rarement utilisée.

Tampon de réordonnancement

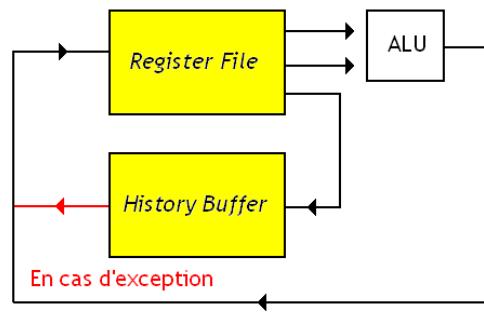
Une autre solution consiste à exécuter nos instructions sans se préoccuper de l'ordre des écritures, avant de remettre celles-ci dans le bon ordre. Pour remettre en ordre ces écritures, les résultats des instructions seront mis en attente dans une FIFO, avant d'être autorisés à être enregistrés dans les registres une fois que cela ne pose pas de problèmes. Cette mémoire tampon s'appelle le **tampon de réordonnancement** (re-order buffer ou ROB). Un résultat est enregistré dans un registre lorsque les instructions précédentes (dans l'ordre du programme) ont toutes écrit leurs résultats dans les registres. Seule l'instruction la plus ancienne peut quitter le ROB et enregistrer son résultat : les autres instructions doivent attendre. Lorsqu'une instruction vient d'être décodée, celle-ci est ajoutée dans le ROB à la suite des autres : les instructions étant décodées dans l'ordre du programme, l'ajout des instructions dans le ROB se fera dans l'ordre du programme, automatiquement. Si une exception a lieu, le ROB se débarrasse des instructions qui suivent l'instruction fautive (celle qui a déclenché l'interruption ou la mauvaise prédiction de branchement) : ces résultats ne seront pas enregistrés dans les registres architecturaux. Quand le ROB est plein, le processeur va bloquer les étages de chargement, décodage, etc. Cela évite de charger des instructions dans celui-ci alors qu'il est plein. Sur les processeurs utilisant un séquenceur microcodé, la fusion de plusieurs instructions machines en une seule micro-opération diminue le nombre d'instructions à stocker dans le ROB, qui stocke les micro-opérations.



Ce ROB est composé de plusieurs **entrées**, des espèces de blocs dans lesquels il va stocker des informations sur les résultats à écrire dans les registres ou la mémoire. Le nombre d'entrées est fixé, câblé une fois pour toutes. Chaque entrée contient l'adresse de l'instruction, obtenue en récupérant le contenu du program counter, pour savoir à quelle instruction reprendre en cas d'erreur de spéculation. Elle contient aussi un bit Exception pour préciser si l'instruction a levé une exception ou non. Lorsqu'un résultat quitte le ROB, pour être enregistré dans les registres, ce bit est vérifié pour savoir s'il faut ou non vider le ROB. Chaque entrée contient aussi le résultat à écrire dans les registres. Néanmoins, il faut prendre garde à un détail : certaines instructions ne renvoient pas de résultat, comme c'est le cas des branchements. La logique voudrait que ces instructions ne prennent pas d'entrée dans le ROB. Mais n'oubliez pas qu'on détermine à quelle adresse reprendre en se basant sur le program counter de l'instruction qui quitte le ROB : ne pas allouer d'entrées dans le ROB à ces instructions risque de faire reprendre le processeur quelques instruction à côté. Pour éviter cela, on ajoute quand même ces instructions dans le ROB, et on rajoute un champ qui stocke le type de l'instruction, afin que le ROB puisse savoir s'il s'agit d'une instruction qui a un résultat ou pas. On peut aussi utiliser cette indication pour savoir si le résultat doit être stocké dans un registre ou dans la mémoire. De plus chaque entrée du ROB contient le nom du registre de destination du résultat, histoire de savoir où l'enregistrer. Enfin, chaque entrée du ROB contient un bit de présence qui est mis à 1 quand le résultat de l'instruction est écrit dans l'entrée, qui sert à indiquer que le résultat a bien été calculé. Suivant le processeur, le ROB peut contenir d'autres informations.

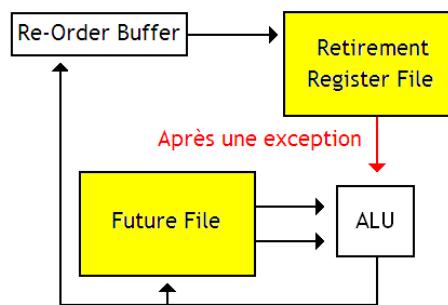
Tampon d'historique

Une autre solution laisse les instructions écrire dans les registres dans l'ordre qu'elles veulent, mais conserve des informations pour remettre les écritures dans l'ordre, pour retrouver les valeurs antérieures. Ces informations sont stockées dans ce qu'on appelle le **tampon d'historique** (history buffer ou HB). Comme pour le ROB, le HB est une mémoire FIFO dont chaque mot mémoire est une entrée qui mémorise les informations dédiées à une instruction. Lorsqu'une instruction s'exécute, elle va souvent modifier le contenu d'un registre, écrasant l'ancienne valeur. Le HB sauvegarde une copie de cette ancienne valeur, pour la restaurer en cas d'exception. Lorsqu'une instruction située dans l'entrée la plus ancienne a levé une exception, il faut annuler toutes les modifications faites par les instructions suivantes. Pour cela, on utilise les informations stockées dans le HB pour remettre les registres à leur ancienne valeur. Plus précisément, on vide le HB dans l'ordre inverse d'ajout des instructions, en allant de la plus récente à la plus ancienne, jusqu'à vider totalement le HB. Une fois le tout terminé, on retrouve bien nos registres tels qu'ils étaient avant l'exécution de l'exception.



Banc de registres futurs

Le HB possède un défaut : remettre les registres à l'état normal prend du temps. Pour éviter cela, on peut utiliser deux bancs de registres. Le premier est mis à jour comme si les exceptions n'existaient pas, et conserve un état spéculatif : c'est le **banc de registres futurs** (future file ou FF). L'autre stocke les données valides en cas d'exception : c'est le **banc de registres de retrait** (retirement register file ou RRF). Le FF est systématiquement utilisé pour les lectures et écritures, sauf en cas d'exception : il laisse alors la main au RRF. Le RRF est couplé à un ROB ou un HB, histoire de conserver un état valide en cas d'exception.



Dépendances de contrôle

Les branchements ont des effets similaires aux exceptions et interruptions. Lorsqu'on charge un branchement dans le pipeline, l'adresse à laquelle brancher sera connue après quelques cycles et des instructions seront chargées durant ce temps. Pour éviter tout problème, on doit faire en sorte que ces instructions ne modifient pas les registres du processeur ou qu'elles ne lancent pas d'écritures en mémoire. La solution la plus simple consiste à inclure des instructions qui ne font rien à la suite du branchement : c'est ce qu'on appelle un délai de branchement. Le processeur chargera ces instructions, jusqu'à ce que l'adresse à laquelle brancher soit connue. Cette technique a les mêmes inconvénients que ceux vus dans le chapitre précédents. Pour éviter cela, on peut réutiliser les techniques vues dans les chapitres précédents, avec quelques améliorations.

Instructions à prédictat

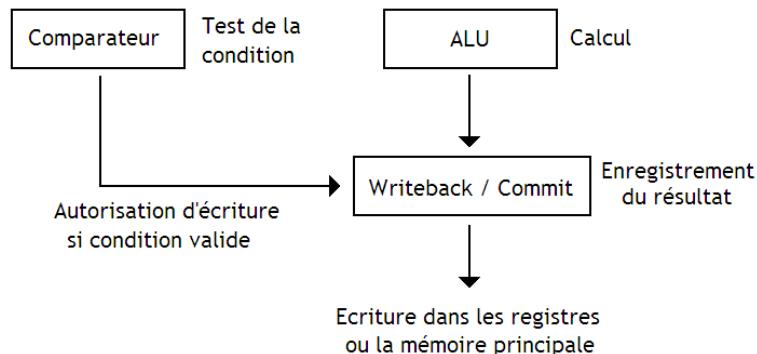
Une première solution est de remplacer les branchements par d'autres instructions. Cela marche bien pour certains calculs assez communs comme les calculs de valeur absolue, le calcul du maximum de deux nombres, etc. Pour cela, les concepteurs de processeurs ont créé des **instructions à prédictat**. Ces instructions à prédictat sont des instructions normales, avec une différence : elles ne font quelque chose que si une condition est respectée, et se comportent comme un NOP (une instruction qui ne fait rien) sinon. Elles permettent de créer de petites structures de contrôle sans branchement. Avec ces instructions, il suffit d'utiliser une instruction de test et de placer les instructions à exécuter ou ne pas exécuter (en fonction du résultat de l'instruction de test) immédiatement à la suite pour créer un SI...ALORS. Évidemment, ces instructions ne sont utiles que pour des structures de contrôle contenant peu d'instructions.

Certains processeurs fournissent des instructions à prédictats en plus d'instructions normales : on dit qu'ils implémentent la **prédition partielle**. Certains d'entre eux ne disposent que d'une seule instruction à prédictat : CMOV, qui copie un registre dans un autre si une condition est remplie. Sur d'autres processeurs, toutes les instructions sont des instructions à prédictat : on parle de **prédition totale**.

Implémenter ces instructions est simple :

- ces instructions font un calcul : elles utilisent donc l'ALU ;
- elles agissent en fonction d'une condition, qu'il faut calculer avec un comparateur ;
- elles ne doivent rien faire si cette condition est fausse, ce qui demande d'annuler l'instruction dans le pipeline.

La première implémentation possible consiste à ne pas envoyer l'instruction aux ALU si la condition n'est pas valide. Le résultat de la comparaison doit être connu avant l'envoi aux ALU du calcul, mais ce résultat est rarement disponible rapidement : le calcul doit être mis en attente tant que le résultat de la comparaison n'est pas disponible. Cette solution est peu pratique, ce qui fait que la seconde méthode est toujours privilégiée. La seconde méthode empêche l'enregistrement du résultat dans les registres ou la mémoire : la comparaison est faite en parallèle du calcul.

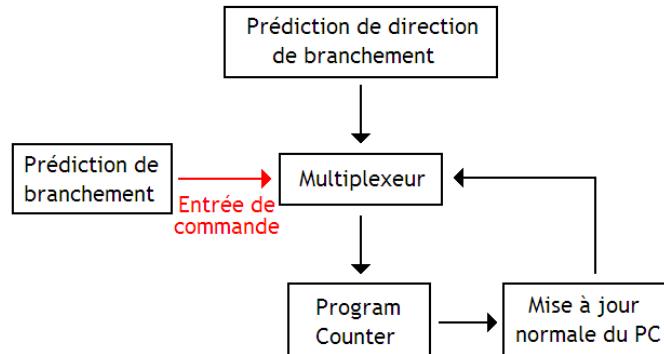


Prédiction de branchement

Pour éviter les délais de branchement, les concepteurs de processeurs ont inventé l'exécution spéculative de branchement, qui consiste à deviner l'adresse de destination du branchement, et l'exécuter avant que celle-ci ne soit connue. Cela demande de résoudre trois problèmes :

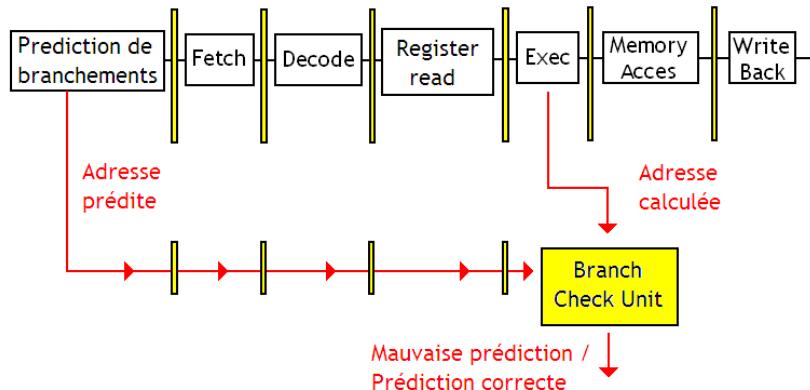
- savoir si un branchement sera exécuté ou non : c'est la **prédiction de branchement** ;
- dans le cas où un branchement serait exécuté, il faut aussi savoir quelle est l'adresse de destination : c'est la **prédiction de direction de branchement** ;
- reconnaître les branchements.

Pour résoudre le premier problème, notre processeur contient un circuit qui va déterminer si notre branchement sera pris (on devra brancher vers l'adresse de destination) ou non pris (on poursuit l'exécution de notre programme immédiatement après le branchement) : c'est l'unité de prédiction de branchement. La prédiction de direction de branchement fait face à un autre problème : il faut déterminer l'adresse de destination de notre branchement. Cette prédiction de direction de branchement est déléguée à un circuit spécialisé : l'unité de prédiction de direction de branchement. Pour faire simple, c'est l'unité de prédiction de branchement qui va autoriser l'unité de prédiction de destination de branchement à modifier le program counter. Dans certains processeurs, les deux unités sont regroupées dans le même circuit.



Erreurs de prédiction

Bien évidemment, une erreur est toujours possible : le processeur peut se tromper en faisant ses prédictions. Dans ce cas, les instructions préchargées ne sont pas les bonnes : c'est une erreur de prédition. Pour corriger les erreurs de préditions, il faut d'abord les détecter. Pour cela, on doit rajouter un circuit dans le processeur : l'unité de vérification de branchement (branch verification unit). Cette unité va juste comparer l'adresse de destination prédictive, et celle calculée en exécutant le branchement : un simple comparateur suffit. Pour cela, l'adresse prédictive doit être propagée dans le pipeline jusqu'à ce que l'adresse de destination du branchement soit calculée.



Une fois la mauvaise prédition détectée, il faut corriger le program counter immédiatement. En effet, si il y a eu erreur de prédition, le program counter n'a pas été mis à jour correctement, et il faut faire reprendre le processeur au bon endroit. Toutefois, cela ne doit être fait que s'il y a erreur de prédition. S'il n'y a pas eu d'erreur de prédition, le program counter a été mis à jour correctement une fois par cycle : il pointe donc plusieurs instructions après l'adresse de destination du branchement. Remettre l'adresse calculée dans le program counter ferait reprendre le processeur en arrière, et lui ferait répéter certaines instructions, ce qui n'est pas prévu.

Pour implémenter cette correction du program counter, il suffit d'utiliser un circuit qui met à jour le program counter seulement s'il y a eu erreur de prédition, et qui laisse l'unité de mise jour fonctionner sinon. La gestion des mauvaises prédictions dépend fortement du processeur. Certains processeurs peuvent reprendre l'exécution du programme immédiatement après avoir détecté la mauvaise prédition (ils sont capables de supprimer les instructions qui n'auraient pas dû être exécutées), tandis que d'autres ne peuvent le faire immédiatement et doivent attendre quelques cycles.

Dans le meilleur des cas, le processeur peut corriger directement le program counter. Mais dans le pire des cas, on est donc obligé d'attendre que les instructions fautives terminent avant de corriger le program counter : il faut que le pipeline soit vidé de ces instructions poubelles. Tant que ces instructions ne sont pas sorties du pipeline, on doit attendre sans charger d'instructions dans le pipeline. Le temps d'attente nécessaire pour vider le pipeline est égal à son nombre d'étages. En effet, la dernière instruction à être chargée dans le pipeline le sera durant l'étape à laquelle on détecte l'erreur de prédition : il faudra attendre que cette instruction quitte le pipeline, et donc qu'elle traverse tous les étages.

De plus, il faut remettre le pipeline dans l'état qu'il avait avant le chargement du branchement. Tout se passe lors de la dernière étape. Et pour cela, on réutilise les techniques vues dans le chapitre précédent pour la gestion des exceptions et interruptions. En utilisant une technique du nom de minimal control dependency, seules les instructions qui dépendent du résultat du branchement sont supprimées du pipeline en cas de mauvaise prédition, les autres n'étant pas annulées. L'utilisation d'un cache de boucle peut permettre de diminuer l'effet des mauvaises prédictions de branchement.

Reconnaissance des branchements

Pour prédire des branchements, le processeur doit faire la différence entre branchements et autres instructions directement lors de l'étage de chargement. Or, un processeur « normal », sans prédition de branchement, ne peut faire cette différence qu'à l'étage de décodage, et pas avant. Les chercheurs ont donc du trouver une solution.

La première solution se base sur les techniques de prédécodage vues dans le chapitre sur le cache. Pour rappel, ce prédécodage consiste à décoder partiellement les instructions lors de leur chargement dans le cache d'instructions et à mémoriser des informations utiles dans la ligne de cache. Dans le cas des branchements, les circuits de prédécodage peuvent identifier les branchements et mémoriser cette information dans la ligne de cache.

Une autre solution consiste à mémoriser les branchements déjà rencontrés dans un cache intégré dans l'unité de chargement ou de prédition de branchement. Ce cache mémorise l'adresse (le program counter) des branchements déjà rencontrés : il est appelé le tampon d'adresse de branchement (branch address buffer). À chaque cycle d'horloge, l'unité de chargement envoie le program counter en entrée du cache. Si il n'y a pas de défaut de cache, l'instruction à charger est un branchement déjà rencontré : on peut alors effectuer la prédition de branchement. Dans le cas contraire, la prédition de branchement n'est pas utilisée, et l'instruction est chargée normalement.

Prédiction de direction de branchement

La prédition de direction de branchement permet de déterminer l'adresse de destination d'un branchement. Il faut faire la différence entre un branchement direct, pour lequel l'adresse de destination est toujours la même, ou un branchement indirect, pour lequel l'adresse de destination est une variable et peut donc changer durant l'exécution du programme. Les branchements directs sont plus facilement prévisibles : l'adresse vers laquelle il faut brancher est toujours la même. Pour les branchements indirects, vu que cette adresse change, la prédire celle-ci est particulièrement compliqué (quand c'est possible).

Lorsqu'un branchement est exécuté, on peut se souvenir de l'adresse de destination et la réutiliser lors d'exécutions ultérieures du branchement. Cette technique marche à la perfection pour les branchements directs, vu que l'adresse de destination est toujours la même. Pour se souvenir de l'adresse de destination, on a besoin d'un cache qui mémorise les correspondances entre l'adresse du branchement, et l'adresse de destination : ce cache est appelé le tampon de destination de branchement (branch target buffer). Ce cache est une amélioration du tampon d'adresse de branchement vu plus haut. En effet, le tampon de destination de branchement doit identifier les branchements déjà rencontrés, et mémoriser leur adresse de destination : on doit ajouter, pour chaque adresse dans le tampon d'adresse de branchement, l'adresse de destination du branchement. Évidemment, la capacité limitée du cache de branchement fait que d'anciennes correspondances sont souvent éliminées pour laisser la place à de nouvelles : le cache utilise un algorithme de remplacement de type LRU. En conséquence, certains branchements peuvent donner des erreurs de prédition si leur correspondance a été éliminée du cache. Pour limiter la casse, il suffit de ne pas mémoriser les correspondances pour les branchements non pris, qui sont proprement inutiles : en cas de non-prédiction, le program counter est incrémenté normalement, pointant automatiquement sur la bonne instruction.

D'autres processeurs mémorisent les correspondances entre branchement et adresse de destination dans les bits de contrôle du cache d'instructions. Cela demande de mémoriser trois paramètres : l'adresse du branchement dans la ligne de cache (stockée dans le branch bloc index), l'adresse de la ligne de cache de destination, la position de l'instruction de destination dans la ligne de cache de destination. Lors de l'initialisation de la ligne de cache, on considère qu'il n'y a pas de branchement dans la ligne de cache. Les informations de prédition de branchement sont alors mises à jour progressivement, au fur et à mesure de l'exécution de branchements. Le processeur doit en même temps

charger l'instruction de destination correcte dans le cache, si un défaut de cache a lieu : il faut donc utiliser un cache multiport. L'avantage de cette technique, c'est que l'on peut mémoriser une information par ligne de cache, comparé à une instruction par entrée dans un tampon de destination de branchement : le nombre d'entrées est plus faible que le nombre de lignes de cache. Mais cela ralentit fortement l'accès au cache, et fait sérieusement grossir celui-ci : les bits de contrôle ajoutés ne sont pas gratuits. En pratique, on n'utilise pas cette technique, sauf sur quelques processeurs (un des processeurs Alpha utilisait cette méthode).

Passons maintenant aux branchements indirects. Dans le cas le plus simple, le processeur considère qu'un branchement indirect se comporte comme un branchement direct : le branchement va brancher vers l'adresse de destination utilisée la dernière fois qu'on a exécuté le branchement. Il suffit d'utiliser un tampon de destination de branchement normal, à condition de le mettre à jour à chaque exécution du branchement. A chaque fois que le branchement change d'adresse de destination, on se retrouve avec une mauvaise prédiction. Tous les processeurs moins récents que le Pentium M prédisent les branchements indirects de cette façon. Certains processeurs haute performance sont capables de prédire l'adresse de destination d'un branchement indirect avec un tampon de destination de branchement amélioré. Ce tampon de destination de branchement amélioré mémorise plusieurs adresses de destination pour un seul branchement et des informations qui lui permettent de déduire plus ou moins efficacement quelle adresse de destination est la bonne. Mais même malgré ces techniques avancées de prédiction, les branchements indirects et appels de sous-programmes indirects sont souvent très mal prédits.

Certains processeurs peuvent prévoir l'adresse à laquelle il faudra reprendre lorsqu'un sous-programme a fini de s'exécuter, cette adresse de retour étant stockée sur la pile, ou dans des registres spéciaux du processeur dans certains cas particuliers. Ils possèdent un circuit spécialisé capable de prédire cette adresse : la prédiction de retour de fonction (return function predictor). Lorsqu'une fonction est appelée, ce circuit stocke l'adresse de retour d'une fonction dans des registres internes au processeur organisés en pile. Avec cette organisation des registres en forme de pile, on sait d'avance que l'adresse de retour du sous-programme en cours d'exécution est au sommet de cette pile.

Prédiction de branchement

Maintenant, voyons comment notre processeur fait pour prédire si un branchement est pris ou non. Si on prédit qu'un branchement est non pris, on continue l'exécution à partir de l'instruction qui suit le branchement. A l'inverse si le branchement est prédit comme étant pris, le processeur devra recourir à l'unité de prédiction de direction de branchement.

Avec la **prédiction statique**, on considère que le branchement est soit toujours pris, soit jamais pris. L'idée est de faire une distinction entre les branchements inconditionnels, toujours pris, et les branchements conditionnels qui sont soit pris, soit non pris. Ainsi, on peut donner un premier algorithme de prédiction dynamique : on suppose que les branchements inconditionnels sont toujours pris alors que les branchements conditionnels ne sont jamais pris (ce qui est une approximation). Cette méthode est particulièrement inefficace pour les branchements de boucle, où la condition est toujours vraie, sauf en sortie de boucle ! Il a donc fallu affiner légèrement l'algorithme de prédiction statique.

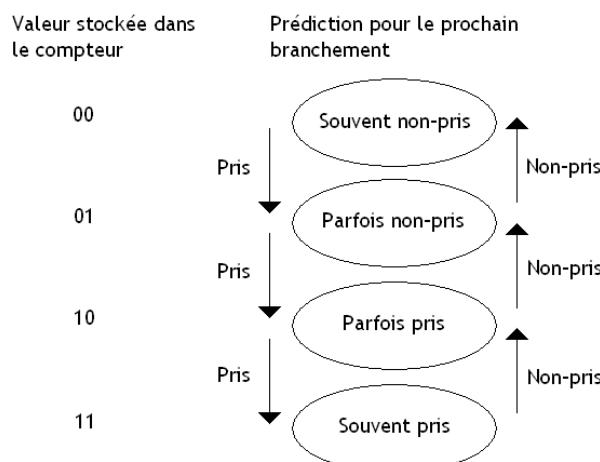
Une autre manière d'implémenter la prédiction statique de branchement est de faire une différence entre les branchements conditionnels ascendants et les branchements conditionnels descendants. Un branchement conditionnel ascendant est un branchement qui demande au processeur de reprendre plus loin dans la mémoire : l'adresse de destination est supérieure à l'adresse du branchement. Un branchement conditionnel descendant a une adresse de destination inférieure à l'adresse du branchement : le branchement demande au processeur de reprendre plus loin dans la mémoire. Les branchements ascendants sont rarement pris (ils servent dans les conditions de type SI...ALORS), contrairement aux branchements descendants (qui servent à fabriquer des boucles). On peut ainsi modifier l'algorithme de prédiction statique comme suit :

- les branchements inconditionnels sont toujours pris ;
- les branchements descendants sont toujours pris ;
- les branchements ascendants ne sont jamais pris.

Sur certains processeurs, certains bits de l'opcode d'un branchement précisent si le branchement est majoritairement pris ou non pris : ces bits spéciaux permettent d'influencer les règles de prédiction statique et de passer outre les réglages par défaut. Ces bits sont appelés des **suggestions de branchement** (branch hint).

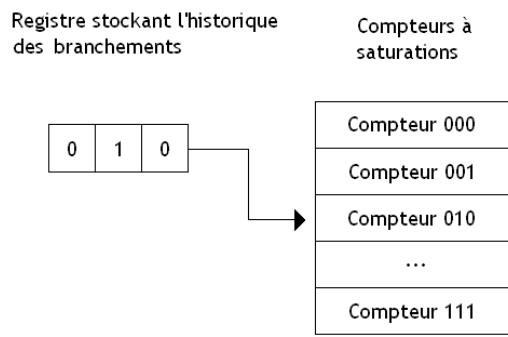
Avec la **prédiction dynamique**, on mémorise à chaque exécution du branchement si celui-ci est pris ou pas, et on effectue une moyenne statistique sur toutes les exécutions précédentes du branchement. Si la probabilité d'être pris est supérieure à 50 %, le branchement est considéré comme pris. Dans le cas contraire, il est considéré comme non pris. L'idée, c'est d'utiliser un compteur qui mémorise le nombre de fois qu'un branchement est pris ou non pris. Ce compteur est initialisé de manière à avoir une probabilité proche de 50 %. Par la suite, si ce branchement est pris, le compteur sera augmenté de 1. Dans le cas contraire, le compteur sera diminué de 1. Pour décider si le branchement sera pris ou pas, on regarde le bit de poids fort du compteur : le branchement est considéré comme pris si ce bit de poids fort est à 1, et non pris s'il vaut 0. Certains processeurs utilisent un seul **compteur à saturation**, qui fait la moyenne pour tous les branchements. D'autres utilisent un compteur par branchement dans le tampon de destination de branchement. Pour la culture générale, il faut savoir que le compteur à saturation du Pentium 1 était légèrement bogue. :-°

Compteur à saturation de deux bits



Le compteur à saturation fonctionne mal pour un branchement exécuté comme suit : pris, non pris, pris, non pris, et ainsi de suite. Même chose pour un branchement qui ferait : pris, non pris, non pris, pris, non pris, non pris, non pris, etc. Une solution pour régler ce problème est de se souvenir des 2, 3, 4 (ou plus suivant les modèles) exécutions précédentes du branchement. Pour cela, on va utiliser un registre à décalage. À chaque fois qu'un branchement s'exécute, on décale le contenu du registre et on fait rentrer un 1 si le branchement est pris, et un 0 sinon. Pour chaque valeur possible contenue dans le registre, on a un compteur à saturation. Le registre est donc couplé à plusieurs compteurs à saturations : pour un registre de n bits (qui se souvient donc des n derniers branchements exécutés), on aura besoin de 2^n compteurs à saturation. Chaque

compteur mémorise le nombre de fois qu'un branchement a été pris à chaque fois que son historique est identique à celui mémorisé dans le registre. Par exemple, si le registre contient 010, le compteur associé à cette valeur (qui est donc numéroté 010), sert à dire : à chaque fois que je me suis retrouvé dans une situation telle que le branchement a été non pris, puis pris, puis non pris, le branchement a été majoritairement pris ou non pris.



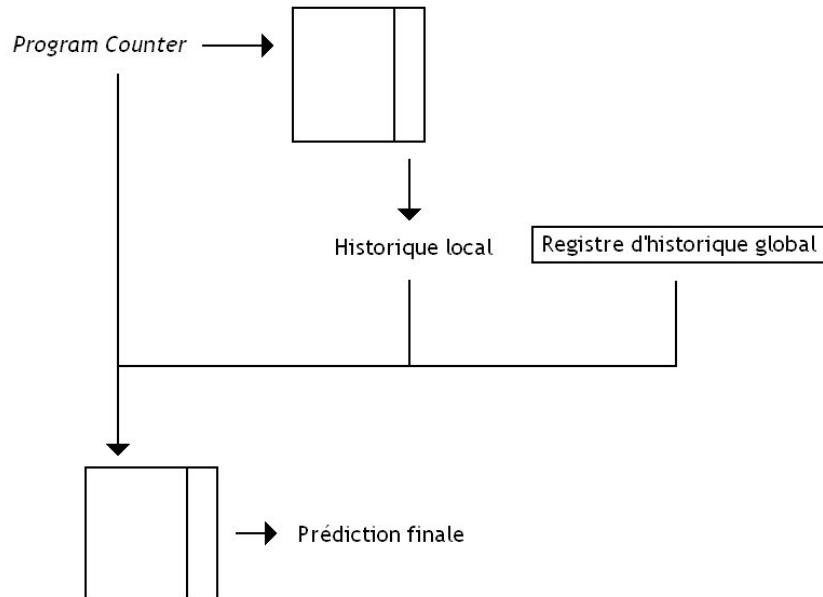
Certains processeurs se payent le luxe d'avoir un registre (et le jeu de compteurs à saturation qui vont avec) différent pour chaque branchement qui est placé dans le tampon de destination de branchement : on dit qu'ils font de la prédiction locale. D'autres, utilisent un registre unique pour tous les branchements : ceux-ci font de la prédiction globale. L'avantage de ces unités de prédiction, c'est que d'éventuelles corrélations entre branchements sont prises en compte. Mais cela a un revers : si deux branchements différents passent dans l'unité de prédiction de branchement avec le même historique, alors ils modifieront le même compteur à saturation. En clair, ces deux branchements vont se marcher dessus sans vergogne, chacun interférant sur les prédiction de l'autre !

C'est la raison d'être des unités de prédiction « **gshare** » et « **gselect** » : limiter le plus possible ces interférences. Pour cela, on va effectuer une petite opération entre l'historique et certains bits de l'adresse de ces branchements (leur program counter) pour trouver quel compteur utiliser. Avec les unités de prédiction « **gshare** », cette opération est un simple XOR entre le registre d'historique et les bits de poids faible de l'adresse du branchement. Le résultat de ce XOR donne le numéro du compteur à utiliser. Avec les unités de prédiction « **gselect** », cette opération consiste simplement à mettre côté à côté ces bits et l'historique pour obtenir le numéro du compteur à saturation à utiliser.

<gallery widths=400px heights=400px Prédiction « gshare ».png|Prédiction « gshare ». Prédiction « gselect ».png|Prédiction « gselect ».>

D'autres unités de prédiction mélangent prédiction globale et locale. Par exemple, on peut citer la prédiction mixte (alloyed predictor). Avec celle-ci, la table des compteurs à saturation est adressée avec la concaténation :

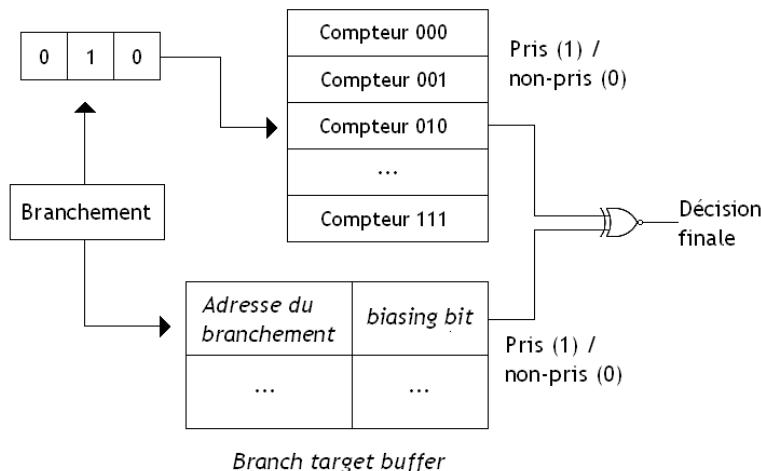
- de bits du program counter ;
- du registre d'historique global ;
- d'un historique local, obtenu avec l'aide d'une table de correspondances branchement → historique local.



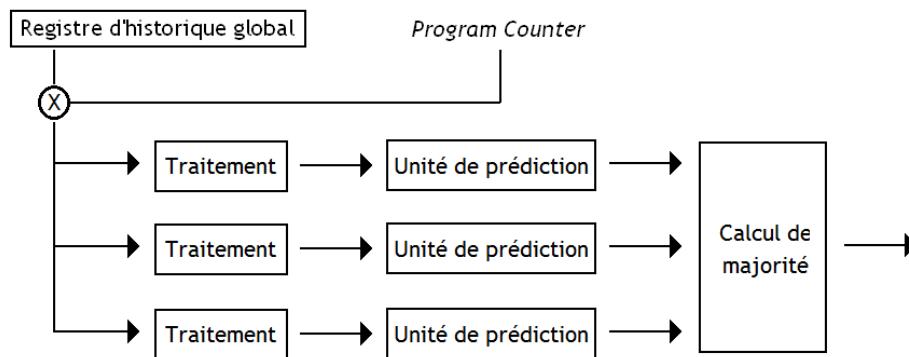
Certaines unités de prédiction de branchement sont capables de prédire à la perfection les branchements de boucles FOR, qui utilisent un compteur : ces branchements sont pris $n - 1$ fois, la dernière exécution étant non prise. La méthode de prédiction utilise un simple compteur, incrémenté à chaque exécution du branchement. Tant que la valeur stockée dans le compteur est différente du nombre n , on considère que le branchement est pris, tandis que si le contenu de ce compteur vaut n , le branchement n'est pas pris. Lorsqu'une boucle est exécutée la première fois, ce nombre n est détecté, et stocké dans le tampon de destination de branchement.

Il est possible de combiner plusieurs unités de prédiction de branchement différentes et de combiner leurs résultats en une seule prédiction. De nombreuses unités de prédiction de branchement se basent sur ce principe. Dans le cas le plus simple, on utilise plusieurs unités de prédiction de branchement, et on garde le résultat majoritaire : si une majorité d'unités pense que le branchement est pris, alors on le considère comme pris, et comme non pris dans le cas contraire.

La prédiction par consensus (agree predictor) utilise une prédiction dynamique à deux niveaux, couplée avec un compteur à saturation. Le décision finale vérifie que ces deux circuits sont d'accord : lorsque les deux unités de branchement ne sont pas d'accord, c'est qu'il y a sûrement eu interférence. Cette décision se fait donc en utilisant une porte XOR suivie d'une porte NON entre les sorties de ces deux unités de prédiction de branchement.



Cette technique est aussi utilisée sur l'unité de prédiction de branchement « e-gskew ». Sur celle-ci, trois unités de prédiction de branchement sont utilisées, avec un vote majoritaire. Pour les trois unités, on commence par effectuer un XOR entre certains bits du program counter, et le registre d'historique global. Simplement, le résultat subit une modification qui dépend de l'unité : la fonction de hachage qui associe une entrée à un branchement dépend de l'unité.



Une autre technique remplace la porte à majorité par un additionneur. Cette technique demande de lire directement le contenu des compteurs de l'unité de prédiction, et de les additionner entre eux. On déduit la prédiction à partir de cette somme : prendre le bit de poids fort suffit.

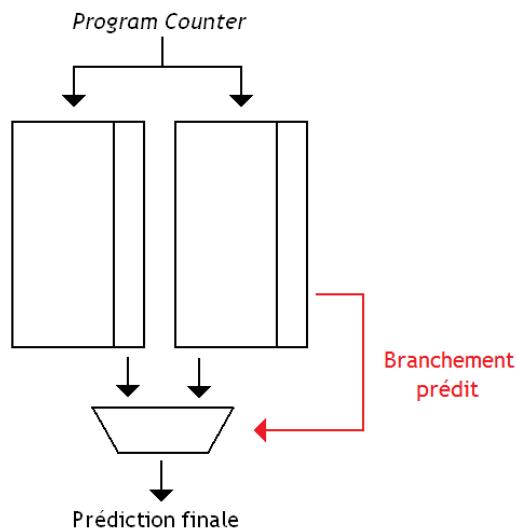
On peut aussi utiliser des historiques de taille différente, qui sont combinés avec le program counter pour identifier les branchements dans chaque unité de prédiction. L'ensemble des tailles d'historique donne quelque chose du type 1, 2, 4, 8, 16, etc. On obtient alors la prédiction à longueur d'historique géométrique (geometric history length predictor).

Une autre technique consiste à prioriser les unités de prédiction de branchement. Par défaut, on utilise une première unité de branchement, relativement simple (un simple compteur à saturation par branchement) : cette unité a la priorité la plus faible, et est utilisée en dernier recours. À côté, on trouve une seconde unité de branchement, qui utilise un autre algorithme. A chaque cycle, la seconde unité reçoit une requête de prédiction de branchement :

- si sa table de correspondances ne permet pas de donner de prédiction pour ce branchement, alors on utilise la prédiction de l'autre unité ;
- dans le cas contraire, la prédiction disponible est utilisée en lieu et place de celle de l'unité simple.

Pour implémenter le tout, rien de plus simple. Il suffit :

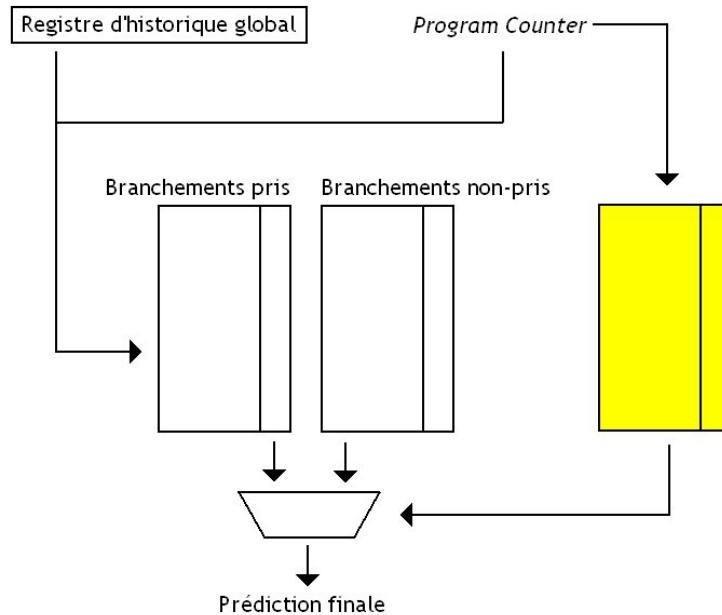
- de placer un multiplexeur qui effectue le choix de la prédiction ;
- d'ajouter une sortie de 1 bit à la seconde unité, qui indique si l'unité peut prédire le branchement (succès de cache dans la table de correspondances) ;
- de commander le multiplexeur avec la sortie de 1 bit.



D'autres unités de prédiction de branchements sont composées de deux unités de prédiction séparées et choisissent laquelle croire avec un compteur à saturation : ce compteur indiquera si la première unité de prédiction a plus souvent raison que l'autre unité. Le résultat de ce compteur à saturation est alors utilisé pour commander un multiplexeur, qui effectuera le choix entre les deux prédictions. C'est cette technique qui est utilisée dans l'unité de prédiction bimodale (bimode predictor). Celle-ci est composée de deux tables :

- une dédiée aux branchements qui sont très souvent pris ;
- une autre pour les branchements très peu pris.

Chaque table attribue un compteur à saturation pour chaque branchements. Les branchements sont identifiés par quelques bits du program counter qui sont XORés avec un registre d'historique global. Le résultat de ce XOR est envoyé aux deux tables, qui vont alors fournir chacune une prédiction. Ces deux prédictions seront envoyées à un multiplexeur commandé par une unité de sélection, qui se chargera de déduire quelle unité a raison : cette unité de sélection est composé d'une table de correspondances program counter → compteur à saturation. On peut encore améliorer l'unité de sélection de prédiction en n'utilisant pas des compteurs à saturation, mais une unité de prédiction dynamique à deux niveaux, ou toute autre unité vue auparavant.



Certains processeurs utilisent plusieurs unités de prédiction de branchements qui fonctionnent en parallèle. Certains d'entre eux utilisent deux unités de prédiction de branchements : une très rapide mais ne pouvant prédire que les branchements « simples à prédire », et une autre plus lente qui prend le relais quand la première unité de prédiction de branchements n'arrive pas à prédire un branchements.

Processseurs à chemins multiples

Pour limiter la casse en cas de mauvaise prédiction, certains processeurs chargent des instructions en provenance des deux chemins (celui du branchements pris, et celui du branchements non pris) dans une mémoire cache, qui permet de récupérer rapidement les instructions correctes en cas de mauvaise prédiction de branchements. Certains processeurs vont plus loin et permettent non seulement de charger les deux chemins d'exécution, mais peuvent aussi exécuter les deux possibilités séparément : c'est ce qu'on appelle l'**exécution stricte** (eager execution). Bien sûr, on n'est pas limité à un seul branchements mais on peut poursuivre un peu plus loin. On peut remarquer que cette technique ne peut pas se passer de prédiction de branchements : pour charger la suite d'instructions correspondant à un branchements pris, il faut savoir quelle est la destination du branchements. Les techniques utilisées pour annuler les instructions du chemin non-pris sont les mêmes que celles utilisées pour la prédiction de branchements.

Avec cette technique, on est limité par le nombre d'unités de calculs, de registres, etc. Pour limiter la casse, on peut coupler exécution stricte et prédiction de branchements. L'idée est de ne pas exécuter les chemins qui ont une probabilité trop faible d'être pris : si un chemin a une probabilité inférieure à un certain seuil, on ne charge pas ses instructions. On parle d'**exécution stricte disjointe** (disjoint eager execution). C'est la

technique qui est censée donner les meilleurs résultats d'un point de vue théorique. À chaque fois qu'un nouveau branchement est rencontré, le processeur refait les calculs de probabilité. Cela signifie que d'anciens branchements qui n'avaient pas été exécutés car ils avaient une probabilité trop faible peuvent être exécuté si des branchements avec des probabilités encore plus faibles sont rencontrés en cours de route.

Implémenter cette technique demande de dupliquer les circuits de chargement (Program Counter, prédition de branchement), pour charger des instructions provenant de plusieurs « chemins d'exécution ». Pour identifier les instructions à annuler, chaque instruction se voit attribuer un numéro par l'unité de chargement, qui indique à quel chemin elle appartient. Chaque chemin peut être dans quatre états, états qui doivent être mémorisés avec le program counter qui correspond :

- invalide (pas de second chemin) ;
- en cours de chargement ;
- mis en pause suite à un défaut de cache ;
- chemin stoppé car il s'est séparé en deux autres chemins, à cause d'un branchement.

Pour l'accès au cache d'instructions, l'implémentation varie suivant le type de cache utilisé. Certains processeurs utilisent un cache à un seul port avec un circuit d'arbitrage. La politique d'arbitrage peut être plus ou moins complexe. Dans le cas le plus simple, chaque chemin charge ses instructions au tour par tour. Des politiques plus complexes sont possibles : on peut notamment privilégier le chemin qui a la plus forte probabilité d'être pris, pas exemple. D'autres processeurs préfèrent utiliser un cache multiport, capable d'alimenter plusieurs chemins à la fois.

Dépendances de données

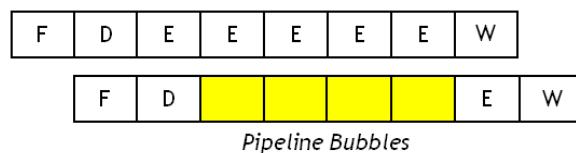
Une limite à une utilisation efficiente du pipeline tient dans l'existence de dépendances entre instructions. Deux instructions ont une dépendance quand elles manipulent la même ressource : le même registre, la même unité de calcul, la même adresse mémoire. Il existe divers types de dépendances, appelées dépendances structurales, de contrôle et de données. Dans ce chapitre, nous allons nous concentrer sur les **dépendances de données**. Deux instructions ont une dépendance de données quand elles accèdent (en lecture ou écriture) au même registre ou à la même adresse mémoire. Différents cas se présentent alors :

- **Read after read** : Nos deux instructions doivent lire la même donnée, mais pas en même temps ! Dans ce cas, on peut mettre les deux instructions dans n'importe quel ordre, cela ne pose aucun problème.
- **Read after write** : La première instruction va écrire son résultat dans un registre ou dans la RAM, et un peu plus tard, la seconde va lire ce résultat et effectuer une opération dessus. La seconde instruction va donc manipuler le résultat de la première. Ces dépendances apparaissent quand une instruction a besoin du résultat de la précédente alors que celui-ci n'est disponible qu'après avoir été enregistré dans un registre, soit après l'étape d'enregistrement. Si on ne fait rien, la seconde instruction ne lira pas le résultat de la première, mais l'ancienne valeur, encore présente dans le registre.
- **Write after read** : La première instruction va lire un registre ou le contenu d'une adresse en RAM, et la seconde va écrire son résultat au même endroit un peu plus tard. Dans ce cas, on doit aussi exécuter la première instruction avant la seconde. Les dépendances WAR n'apparaissent que sur les pipelines où l'écriture des résultats a lieu assez tôt (vers le début du pipeline), et les lectures assez tard (vers la fin du pipeline).
- **Write after write** : Nos deux instructions effectuent des écritures au même endroit : registre ou adresse mémoire. Dans ce cas aussi, on doit conserver l'ordre des instructions et ne pas réordonner, pour les mêmes raisons que les deux dépendances précédentes. Les dépendances WAW n'apparaissent que si le pipeline autorise les instructions sur plusieurs cycles d'horloge ou les écritures qui prennent plusieurs étages.

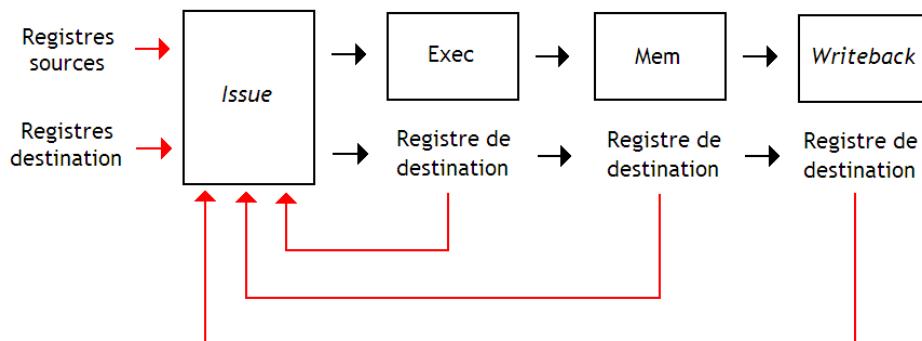
Si deux instructions ont une dépendance de données, la première doit avoir terminé son accès mémoire avant que l'autre ne commence le sien. Et cette contrainte n'est pas forcément respectée avec un pipeline, dont le principe même est de démarrer l'exécution d'une instruction sans attendre que la précédente soit terminée. Dans ces conditions, l'ordre de démarrage des instructions est respecté, mais pas l'ordre des accès mémoire. Pour éviter tout problème avec ces dépendances, on est obligé d'insérer des instructions qui ne font rien entre les deux instructions dépendantes. Mais insérer ces instructions nécessite de connaître le fonctionnement du pipeline en détail : niveau portabilité, c'est pas la joie !

Bulle de pipeline

Il est possible de déléguer cet ajout de NOP au processeur, à ses unités de décodage. Si une dépendance de données est détectée, l'unité de décodage d'instruction met l'instruction en attente tant que la dépendance n'est pas résolue. Durant ce temps d'attente, on insère des vides dans le pipeline : certains étages seront inoccupés et n'auront rien à faire. Ces vides sont appelés des calages (stall), ou **bulles de pipeline** (pipeline bubble). Lors de cette attente, les étages qui précèdent l'unité de décodage sont bloqués en empêchant l'horloge d'arriver aux étages antérieurs au décodage.



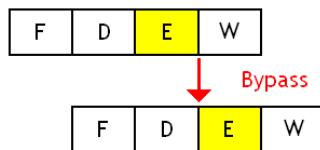
C'est un nouvel étage, l'**étage d'émission** (issue), qui détecte les dépendances et rajoute des calages si besoin. Pour détecter les dépendances, il va comparer les registres utilisés par l'instruction à émettre et ceux utilisés par les instructions dans le pipeline : il n'y a pas de dépendance si ces registres sont différents, alors qu'il y a dépendance dans le cas contraire. L'unité d'émission est donc un paquet de comparateurs reliés par des portes OU. En sortie, elle fournit un signal STALL, qui indique s'il faut caler ou non. L'unité d'émission doit connaître les registres de destination des instructions dans le pipeline, ainsi que les registres utilisés par l'instruction à émettre. Obtenir les registres de destination des instructions dans le pipeline peut se faire de deux manières. La première méthode utilise le fait que les noms de registres sont propagés dans le pipeline, comme tous les signaux de commande. Dans ces conditions, rien n'empêche de relier les registres tampons chargés de la propagation à l'unité d'émission.



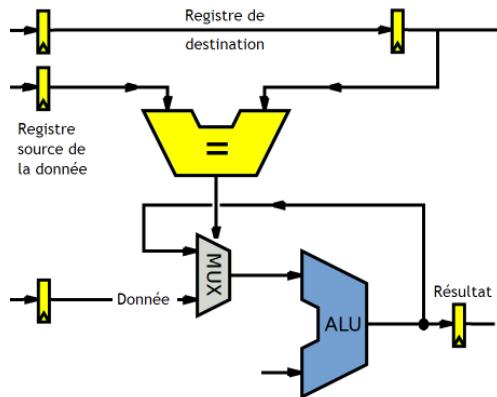
Avec la seconde possibilité, l'unité d'émission mémorise ces registres dans une petite mémoire : le **scoreboard**. Cette mémoire est une mémoire dont les mots mémoire font un bit : chaque adresse correspond à un nom de registre, et le bit correspondant à cet adresse indique si le registre est réservé par une instruction en cours d'exécution dans le pipeline. Lors de l'émission, le scoreboard est adressé avec les noms des registres source et destination utilisés dans l'instruction, pour vérifier les bits associés. Ce scoreboard est mis à jour lorsqu'une instruction écrit son résultat dans un registre, à l'étape d'enregistrement : le scoreboard met le bit correspondant à zéro.

Contournement et réacheminement

Pour diminuer l'effet des dépendances RAW, on peut faire en sorte que le résultat d'une instruction soit rapidement disponible, avant d'être enregistré dans les registres. Il s'agit de la technique du contournement (**bypass**). Cette méthode est spéculative : rien n'indique que ce résultat doit être enregistré dans les registres, vu qu'une prédition de branchement peut venir mettre son grain de sel.



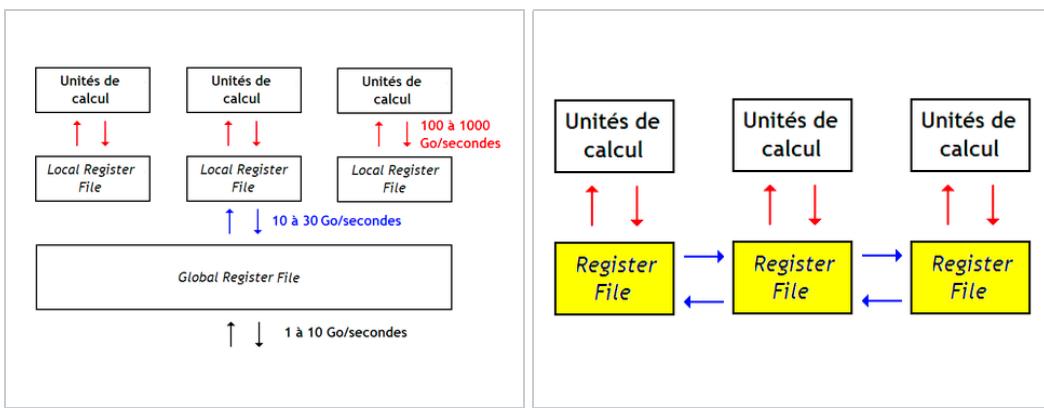
Implémenter la technique du contournement demande de relier la sortie de l'unité de calcul sur son entrée en cas de dépendances, et à la déconnecter sinon : cela se fait avec un multiplexeur. Pour détecter les dépendances, il faut comparer le registre destination avec le registre source en entrée : si ce registre est identique, on devra faire commuter le multiplexeur pour relier la sortie de l'unité de calcul.



Pour améliorer un peu les performances du système de contournement, certains processeurs ajoutent un petit cache en sortie des unités de calcul : le **cache de contournement** (bypass cache). Celui-ci mémorise les n derniers résultats produits par l'unité de calcul : le tag de chaque ligne de ce cache est le nom du registre du résultat.

Sur les processeurs ayant plusieurs d'unités de calculs, chaque sortie d'une unité de calcul doit être reliée aux entrées de toutes les autres, avec les comparateurs qui vont avec ! Pour limiter la casse, on ne relie pas toutes les unités de calcul ensemble. À la place, on préfère regrouper ces unités de calcul dans différents blocs séparés qu'on appelle des agglomérats (**cluster**) : le contournement est alors rendu possible entre les unités d'un même agglomérat, mais pas entre agglomérats différents. Cette agglomération peut prendre en compte les interconnexions entre unités de calcul et registres : les registres doivent alors être agglomérés. Et cela peut se faire de plusieurs façons différentes.

Une première solution, déjà vue dans les chapitres sur la micro-architecture d'un processeur, consiste à découper le banc de registres en plusieurs bancs de registres plus petits. Il faut juste prévoir un réseau d'interconnexions pour échanger des données entre bancs de registres. Dans la plupart des cas, cette séparation est invisible du point de vue de l'assembleur et du langage machine. Le processeur se charge de transférer les données entre bancs de registres suivant les besoins. Sur d'autres processeurs, les transferts de données se font via une instruction spéciale, souvent appelée COPY. Sur d'autres processeurs, on utilise une **hiérarchie de registres** : d'un coté, une couche de bancs de registres reliés aux ALU, de l'autre, un banc de registres qui sert à échanger des données entre les bancs de registres.



Hiérarchie de registres.

Banc de registres distribué.

Certains chercheurs ont adapté cette hiérarchie de bancs de registres de façon à ce que les bancs de registres reliés aux unités de calcul se comportent comme des caches. Suivant l'implémentation, les écritures et lecture en mémoire peuvent lire ou écrire dans tous les niveaux de cache, ou uniquement dans le niveau de banc de registres le plus proche de la mémoire. Il faut noter que le préchargement est possible entre bancs de registres. Dans d'autres travaux, on préfère y stocker les résultats qui ne sont pas utilisés après un contournement : ces valeurs sont écrites dans tous les niveaux de la hiérarchie des registres, tandis que les valeurs contournées sont écrites uniquement dans les registres des niveaux inférieurs.

Sur de nombreux processeurs, un branchement est exécuté par une unité de calcul spécialisée. Or les registres à lire pour déterminer l'adresse de destination du branchement ne sont pas forcément dans le même agglomérat que cette unité de calcul. Pour éviter cela, certains processeurs disposent d'une unité de calcul des branchements dans chaque agglomérat. Dans les cas où plusieurs unités veulent modifier le program counter en même temps, un système de contrôle général décide quelle unité a la priorité sur les autres. Mais d'autres processeurs fonctionnent autrement : seul un agglomérat possède une unité de branchement, qui peut recevoir des résultats de tests de toutes les autres unités de calcul, quel que soit l'agglomérat.

Dépendances structurelles

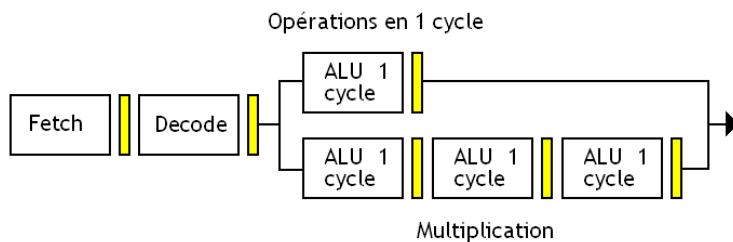
Il se peut qu'un circuit du processeur doive être manipulé par plusieurs instructions à la fois. Par exemple, deux instructions peuvent ainsi vouloir utiliser l'unité de calcul ou accéder à la mémoire simultanément. Dans ce cas, il est difficile à un circuit de traiter deux instructions à la fois s'il n'est pas conçu pour. Le processeur fait alors face à une **dépendance structurelle**. Celle-ci se résout en arbitrant l'accès au circuit fautif. Une des instructions est mise en attente durant quelques cycles et attend son tour tant que la première n'a pas fini d'utiliser le circuit. Pour cela, on peut placer des instructions inutiles dans notre programme, afin de décaler nos instructions dans le pipeline. Mais ce n'est pas une solution très élégante, sans compter qu'elle ne marche pas pour les instructions de durée variable. Mais le processeur peut gérer une telle situation de lui-même, s'il est conçu pour.

Pipeline sans boucle

Nous avons vu il y a quelques chapitres que les étages sont censés être indépendants, chacun possédant ses propres circuits. Dans ces conditions, les dépendances structurelles ne sont pas censées exister. Mais il n'est pas rare que les concepteurs de processeurs laissent des dépendances structurelles pour économiser des transistors. C'est notamment le cas pour ce qui est de la gestion des **instructions multi-cycles** (qui prennent plusieurs cycles pour s'exécuter). Dans les grandes lignes, il existe trois grandes solutions pour résoudre ces dépendances structurelles : la duplication de circuit, l'échelonnement et l'insertion de bulles de pipeline.

Échelonnement

La première solution consiste à pipeliner, échelonner les instructions. Si jamais vous avez une opération qui prend cinq cycles, pourquoi ne pas lui fournir un seul circuit et l'échelonner en cinq étages ? Pour certains circuits, c'est possible : on peut totalement échelonner une unité de multiplication, par exemple. En faisant ainsi, il est possible de démarrer une nouvelle multiplication à chaque cycle d'horloge dans la même ALU, éliminant ainsi toute dépendance structurelle. Cependant, certaines instructions s'échellonnent mal, notamment les divisions : chaque étage peut prendre 10 à 20 cycles pour s'exécuter. Il faut alors trouver une autre solution pour ces instructions.



Duplication de circuits

La seconde solution consiste à dupliquer les unités de calcul. Ainsi, si une instruction bloque une ALU durant plusieurs cycles, on peut exécuter d'autres instructions de calcul dans une autre ALU. En théorie, on peut supprimer toute dépendance structurelle totalement en utilisant autant d'unités de calcul qu'une instruction met de cycles pour s'exécuter. Prenons un exemple : toutes mes instructions arithmétiques prennent un cycle, à part la multiplication qui prend cinq cycles et la division qui prend quarante cycles. Je peux supprimer toute dépendance structurelle en utilisant une ALU pour les opérations en un cycle, cinq ALU capables de faire une multiplication, et quarante ALU dédiées aux divisions.

Les statistiques montrent qu'il est rare que deux opérations nécessitant plusieurs cycles d'horloge se suivent dans un programme, ce qui fait que les unités de calcul sont rarement présentes en un nombre important d'exemplaires. La plupart des processeurs se limitent à utiliser un aval spécialisé pour les opérations en plusieurs cycles, à côté d'une unité de calcul dédiée aux opérations en un seul cycle.

Bulles de pipeline

La dernière technique consiste à laisser la gestion des dépendances au séquenceur. Sur toutes les instructions qui bataillent pour accéder au circuit, une instruction a accès au circuit, les autres étant mises en attente. Pour cela, les dépendances structurelles entre instructions sont vérifiées à l'émission.

Une première solution consiste à déterminer les paires d'instructions consécutives qui peuvent donner lieu à des dépendances structurelles. Par exemple, si une instruction de multiplication de cinq cycles est dans le pipeline, je sais que je ne peux pas démarrer une nouvelle multiplication immédiatement après, vu qu'il n'y a qu'un seul multiplicateur. Déetecter les dépendances structurelles consiste à comparer l'opcode de l'instruction à émettre, avec les opcodes des instructions en cours d'exécution dans le pipeline. Si jamais l'opcode de l'instruction à émettre et une opcode dans le pipeline forment une paire consécutive d'opérations interdite, il y a dépendance structurelle. En conséquence, l'instruction à émettre est bloquée, et l'unité d'émission insère des NOP dans le pipeline. Si jamais l'unité de calcul est échelonnée, on peut adapter ce principe en retirant certains étages du processus de comparaison.

Une autre solution est de connaître, pour chaque unité, le temps nécessaire avant de pouvoir lancer une nouvelle instruction dans l'aval. Les instructions ayant des durées fixées, quelques compteurs dans le séquenceur permettent de dire dans combien de temps telle ou telle unité de calcul sera libre. Seules les instructions d'accès mémoire, seules à avoir des temps de latence variables, sont gérées différemment. Pour les accès mémoire, c'est l'unité de gestion mémoire qui indique si elle est libre : elle génère un bit MEM_FREE qui dira si la mémoire est accessible ou non. L'unité d'émission aura juste à vérifier ce signal pour savoir si elle peut émettre une nouvelle instruction d'accès mémoire.

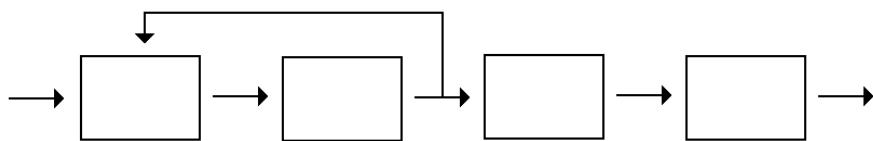
Pipeline avec boucles

Certains pipelines sont un peu plus compliqués, et contiennent des boucles : certains étages peuvent envoyer leur résultat à des étages précédents. Alors certes, les pipelines de ce type sont très rares, mais ils existent. Avec des boucles, la gestion des dépendances structurelles devient beaucoup plus compliquée, et diverses solutions existent. Toutes ces solutions consistent à envoyer des bulles de pipeline au bon moment.

Registre de décalage de contrôle

Ces solutions se basent sur des tables de réservation, des tables dans lesquelles on indique quel étage est occupé par une instruction, et à quel moment.

Par exemple, le pipeline suivant...



... aura la table de réservation suivante. Supposons que je lance une instruction dans le pipeline à un temps t_0 . Par la suite, k cycles plus tard, je veux lancer la même instruction dans le pipeline. Pour détecter les collisions, il me suffit de prendre la table de l'instruction en deux exemplaires : un pour l'instruction lancée à t_0 et un autre décalé de k cycles, qui correspond à l'instruction qu'on veut émettre.

Étage 1	Étage 2	Étage 3	Étage 4
X	-	-	-
-	X	-	-
X	-	-	-
-	X	-	-
-	-	X	-
-	-	-	X

Nous allons supposer que l'on lance deux instructions à la suite dans le pipeline. Dans la table de réservation, on voit bien qu'il n'y a pas de conflit : les instructions occupent des étages différents à chaque instant.

Étage 1	Étage 2	Étage 3	Étage 4
X	-	-	-
x	X	-	-
X	x	-	-
x	X	-	-
-	x	X	-
-	-	x	X
-	-	-	x

Maintenant, supposons que les instructions soient lancées à deux cycles d'intervalle. La table de réservation est alors celle-ci. On le voit clairement : certains étages sont occupés par plusieurs instructions, ce qui n'est pas possible. En clair : il n'est pas possible de démarrer une nouvelle instruction deux cycles après une autre.

Étage 1	Étage 2	Étage 3	Étage 4
X	-	-	-
-	X	-	-
X x	-	-	-
-	X x	-	-
x	-	X	-
-	x	-	X
-	-	x	-
-	-	-	x

Fort heureusement, le processeur n'a pas à calculer lui-même la possibilité de conflits en utilisant des tables de réservation. Il peut précalculer les latences autorisées et interdites entre deux instructions. Le résultat de ce précalcul est ce qu'on appelle un **vecteur de collision**. Pour un pipeline de n étages, ce vecteur de collision est un simple ensemble de n bits, numérotés de 1 à n , le bit numéro x indiquant si il y a conflit entre deux instructions émises à x cycles d'intervalle. Par exemple, le bit 1 indiquera si on peut démarrer deux instructions consécutives, le bit 2 indiquera si on peut lancer une nouvelle instruction deux cycles plus tard, etc. Ce vecteur de collision est précalculé pour chaque aval du processeur, et est ensuite utilisé par l'unité d'émission pour déterminer si une nouvelle instruction peut être émise. Avec une méthode naïve, l'unité d'émission va regarder combien de cycles se sont écoulés depuis la dernière instruction émise dans l'aval : ce nombre de cycles sera noté n dans ce qui suit. Ensuite, il lui reste à regarder le n -ième bit du vecteur de collision : la nouvelle instruction doit être mise en attente s'il vaut zéro, et peut être émise s'il vaut 1.

Mais on peut légèrement optimiser le tout. Lors de l'émission d'une nouvelle instruction, un registre à décalage est initialisé avec le vecteur de collision. À chaque cycle, ce registre à décalage est décalé d'un cran vers la gauche. Ainsi, si ce registre a été initialisé il y a n cycles, alors c'est le n -ième bit qui sortira de ce registre. Le bit qui sortira de ce registre sera donc le bit qui indique si on peut émettre une nouvelle instruction. Malheureusement, ce registre seul ne marche pas : si on initialise le registre à décalage avec le vecteur de collision de l'instruction émise, alors son ancien contenu est perdu. Conséquence : il ne prend en compte que les dépendances avec la dernière instruction à avoir été émise, et pas celles émises avant. Pour résoudre ce problème, il faut mixer les vecteurs de collision de toutes les instructions précédentes. Pour cela, on doit rajouter un circuit qui mixe convenablement le vecteur de collision avec le contenu du registre à décalage. La solution consiste donc à faire un OU entre le vecteur de collision à ajouter dans le registre, et le contenu de ce registre.

Diagramme d'état

En utilisant la technique du dessus, on n'obtient pas un résultat optimal. Dans certains cas, démarrer une instruction dès que possible peut ne pas être la meilleure solution, contrairement au fait d'attendre quelques cycles avant l'issue. La raison est basée sur des mathématiques assez complexes, aussi je me permets de la passer sous silence. Il est toutefois possible d'obtenir un résultat meilleur en adaptant le circuit d'émission. Ce circuit d'émission est un circuit séquentiel, qui peut se décrire intégralement avec un graphe de transitions. Pour créer ce graphe, il faut commencer par créer un nœud contenant le vecteur de collision initial. Ce nœud représente le contenu du registre à décalage quand on vient tout juste de démarrer une instruction dans un pipeline / aval totalement vide. À partir de ce vecteur de collision, on pourra émettre des instructions après un certain nombre de cycles. Par exemple, on suppose que les latences autorisées sont 2, 3, 7, 8+ (8 et supérieur) : dans ce cas, on pourra émettre une nouvelle instruction 2, 3, 7, 8 cycles après une autre. Une fois ces instructions émises, le registre à décalage aura été modifié. Dans ce cas, on crée un nœud pour chaque latence autorisée, et on met le contenu du registre à décalage obtenu dans ce nœud. On numérote la flèche avec le nombre de cycles qui séparent les deux instructions. Et on recommence à partir de chacun des nœuds obtenu à cette étape. On poursuit ainsi de suite jusqu'à retomber sur un nœud déjà présent dans le graphe. Au bout d'un moment, on obtient alors le graphe de transition total. Pour savoir comment gérer au mieux ces transitions, il faut trouver les boucles dans ce graphe qui partent du nœud de pipeline vide et qui retournent à ce nœud. La boucle idéale est celle dont la somme des numéros des flèches est la plus petite possible.

Aval multifonction

Certains avals sont capables d'effectuer plusieurs opérations différentes : par exemple, on peut avoir un aval qui peut faire soit des multiplications, soit des divisions. Dans ces conditions, les techniques vues au-dessus doivent être améliorées un petit peu. Par exemple, la technique avec un registre à décalage et des portes OU peut être adaptée : il suffit de rajouter plusieurs registres à décalage, et quelques vecteurs de collision. Reprenons l'exemple avec un aval qui peut faire soit des multiplications, soit des divisions, on devrait utiliser deux registres à décalage : un pour autoriser l'émission des divisions, et un autre pour les multiplications. Techniquement, il faut autant de registres à décalages que l'on a d'instruction. Il faudrait aussi utiliser autant de vecteurs de collision qu'il y a de paires d'instructions possible (on lance une instruction après l'autre, ce qui donne une paire). Pour l'exemple avec multiplication et division, on aurait :

- un des vecteurs de collision serait celui obtenu en superposant la table de réservation d'une addition suivie d'une multiplication ;
- un autre serait obtenu en superposant la table de réservation d'une multiplication suivie d'une addition ;
- un autre serait obtenu en superposant la table de réservation d'une multiplication suivie d'une multiplication ;
- un autre serait obtenu en superposant la table de réservation d'une addition suivie d'une addition.

Exécution dans le désordre

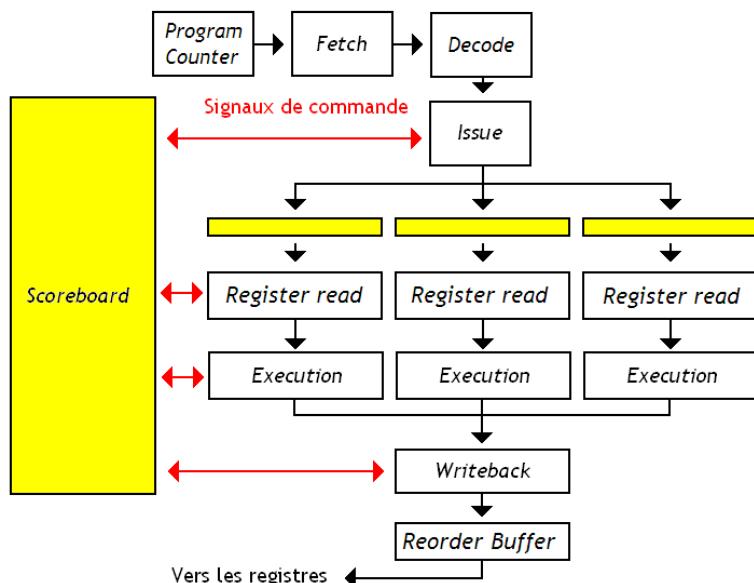
On peut diminuer l'influence des bulles de pipeline en changeant l'ordre des instructions du programme : l'idée est de remplir les bulles de pipeline avec des instructions indépendantes. On profite du fait que l'ordre induit par les dépendances de données est moins strict que l'ordre imposé par le program counter, même si ces deux ordres donnent le même résultat. Si on exclut l'intervention du compilateur, qui peut réordonner les instructions pour profiter au mieux du pipeline, les moyens pour combler les vides sont au nombre de deux.

- Avec les **architectures dataflow**, le compilateur ajoute des informations sur les dépendances de données dans chaque instruction, à côté de l'opcode. Le processeur ne détecte pas les dépendances à l'exécution : c'est le compilateur qui s'en charge. Ce dernier mémorise les informations sur les dépendances dans le programme lui-même (le plus souvent à l'intérieur des instructions). Le processeur utilise alors les informations fournies par le compilateur pour répartir les calculs sur les unités de calcul de manière optimale.
- Avec la **exécution dans le désordre**, si une instruction attend que ses opérandes soient disponibles, on peut remplir le temps d'attente par une instruction indépendante. Le processeur peut virer certaines dépendances qui apparaissent à l'exécution (branchements), ce qui est impossible pour le compilateur. Cette technique utilise plusieurs back ends, voire plusieurs unités de calcul. En effet, l'exécution dans le désordre revient à faire ce qui suit, chose qui est impossible si les deux instructions utilisent la même unité de calcul.

Il existe différents types d'exécution dans le désordre : le scoreboarding et l'algorithme de Tomasulo. Ces deux techniques ont une caractéristique commune : les instructions sont décodées dans l'ordre du programme, mais exécutées dans un ordre différent. Qui plus est, on ne décode qu'une instruction à la fois. On verra plus tard que certains processeurs arrivent à passer outre ces limitations, mais nous n'en sommes pas encore là !

Scoreboarding

Avec le **scoreboarding**, la gestion et la détection des dépendances sont effectuées par le scoreboard, lui-même secondé par d'autres circuits annexes, notamment l'étape d'émission. Celle-ci gère les dépendances structurelles et WAW, mais pas les dépendances RAW. Celles-ci sont prises en charge en mettant en attente les instructions dont les opérandes ne sont pas disponibles, dans une mémoire tampon. Celle-ci est située entre l'étage d'émission et les unités de calcul, et est gérée par le scoreboard. Une fois ses opérandes prêts, l'instruction est envoyée à une unité de calcul et est exécutée, sans se soucier de l'ordre des instructions dans le programme. Toutes les instructions ne prennent pas le même temps pour s'exécuter, et le scoreboard doit en tenir compte pour déterminer la disponibilité des opérandes (et donc gérer les dépendances RAW). Quand le résultat de notre instruction est calculé, le scoreboard est prévenu par l'unité de calcul, ce qui lui permet de détecter la disponibilité des opérandes. Reste à gérer les dépendances WAR. Pour cela, il faut mettre l'instruction en attente tant que les lectures dans le registre de destination ne sont pas terminées. Pour cela, les résultats sont placés dans une file d'attente, et s'enregistrent dans le banc de registres quand c'est possible. C'est le scoreboard qui autorise ou non cette écriture.



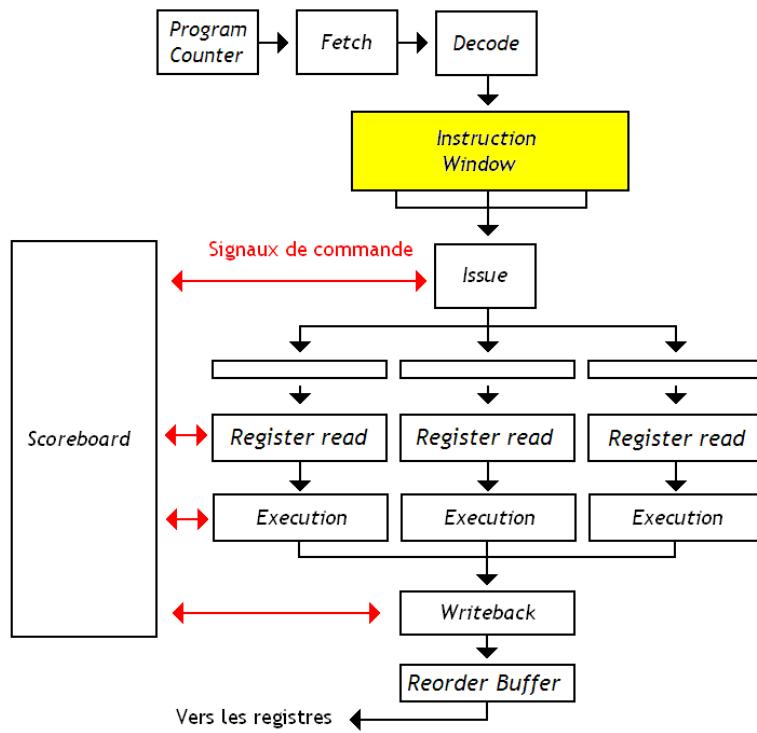
Fenêtres d'instruction

Le scoreboard est capable de fournir des gains proches de 10 à 30 % suivant les programmes. Mais celui-ci a toutefois un léger défaut : si deux instructions consécutives ont une dépendance structurelle ou WAW, l'instruction attend dans l'étage d'émission. Au lieu de bloquer le pipeline à l'étape d'émission en cas de dépendances, pourquoi ne pas utiliser des instructions indépendantes dans la suite du programme ? Pour que cela fonctionne, on est obligé de faire en sorte que l'étage d'émission mette en attente les instructions, sans bloquer le pipeline (chose que fait le scoreboard). Généralement, cette mise en attente se fait en utilisant une ou plusieurs mémoires spécialisées, chaque mémoire étant composée d'entrées dédiées à une instruction chacune.

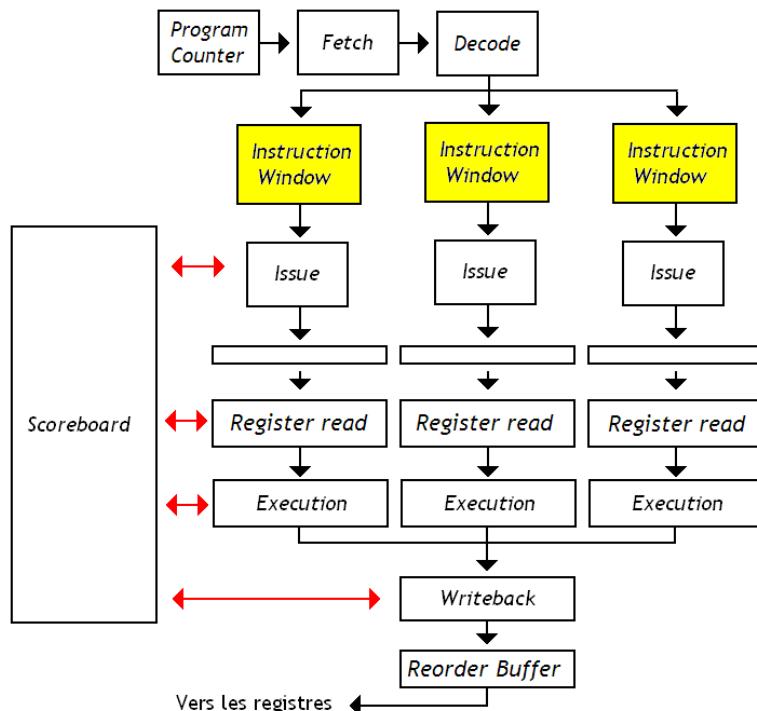
Fenêtre d'instruction

Le cas le plus simple n'utilise qu'une seule mémoire tampon : la **fenêtre d'instruction**. L'unité d'émission va alors :

- détecter les dépendances ;
- mettre en attente les instructions qui doivent l'être ;
- répartir les instructions sur les unités de calcul.



Sur certains processeurs, la répartition des instructions sur les unités de calcul est prise en charge par un circuit séparé de l'étage d'émission, les deux étant séparés par une mémoire tampon. On doit ainsi utiliser plusieurs fenêtres d'instruction, souvent une par unité de calcul. Ainsi, l'émission se contente de répartir les instructions sur les unités de calcul, la détection des dépendances étant prise en charge par les fenêtres d'instruction. Ces fenêtres d'instruction spécialisées sont parfois appelées des **stations de réservation**. Le premier avantage des stations de réservation est qu'elles sont plus petites qu'une grosse fenêtre d'instruction. L'autre avantage, c'est qu'il est possible de démarrer l'exécution de plusieurs instructions simultanément : une instruction par station de réservation, contre une par fenêtre d'instruction. Mais les stations de réservation sont souvent sous-utilisées, partiellement remplies, contrairement aux fenêtres d'instruction. Dans ce qui va suivre, j'utiliserai le terme « fenêtre d'instruction » pour parler des stations de réservation, ainsi que de la fenêtre d'instruction.



Compactage

À chaque cycle, les instructions décodées vont être ajoutées dans la fenêtre d'instruction, dans des entrées vides. Vu que les instructions quittent celle-ci dans le désordre, ces vides sont dispersés dans la fenêtre d'instruction, ce qui pose problème pour déterminer où placer les nouvelles instructions. La solution la plus triviale consiste à conserver une liste des vides, mise à jour à chaque insertion ou émission d'instruction. Une autre solution consiste à éliminer les vides en compactant la fenêtre d'instruction à chaque cycle d'horloge. Des circuits se chargent de détecter les vides et de regrouper les instructions en un unique bloc. Il faut signaler que certaines processeurs arrivent à se passer de cette étape de compactage, mais au prix de fenêtres d'instruction nettement plus complexes.

Autre problème : quand il faut choisir quelle instruction émettre, il y a toujours plusieurs candidats. Si on choisit mal, des instructions restent en

attente trop longtemps parce que d'autres instructions plus jeunes leur passent devant. Pour éviter cela, les instructions les plus vieilles, les plus anciennes, sont prioritaires. Pour cela, on peut utiliser une FIFO un peu spéciale pour la fenêtre d'instruction. Si les ajouts d'instruction se font dans l'ordre, les instructions ne quittent pas forcément la fenêtre d'instruction dans l'ordre imposé par une FIFO : les instructions restent triées dans leur ordre d'ajout, même s'il y a des vides entre elles. Dans ces conditions, il est préférable que le compactage conserve l'ordre FIFO des instructions. Dans ces conditions, l'instruction la plus ancienne est celle qui est située à l'adresse la plus faible : le circuit de sélection peut donc être fabriqué avec des encodeurs, et est relativement simple.

Un problème de latence...

Idéalement, on voudrait démarrer une nouvelle instruction sur l'unité de calcul dès qu'elle est libre. Cependant, une unité d'émission naïve attend que les opérandes soient disponibles avant de démarrer une nouvelle instruction dans cette ALU. Mais entre le moment où cette nouvelle instruction quitte la fenêtre d'instruction et le moment où elle arrive dans l'unité de calcul, plusieurs cycles se sont écoulés. Pour donner un exemple, sur le Pentium 4, on trouve 6 étages entre la fenêtre d'instruction et l'entrée de l'ALU. Les instructions ne démarrent donc pas aussi vite qu'elles le pourraient, du moins si elles attendent la disponibilité de leurs opérandes.

Émission en avance

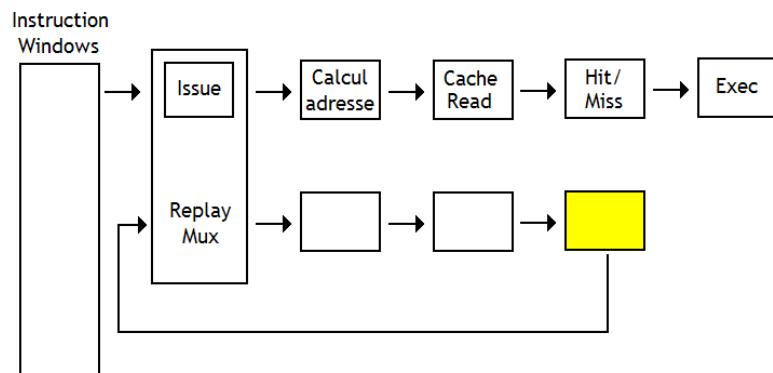
La solution consiste à démarrer des instructions en avance, quelques cycles avant que les opérandes soient tous disponibles. Le nombre de cycles d'avance est facile à connaître : c'est le nombre d'étages entre l'unité de calcul et la fenêtre d'instruction. Le cas le plus simple est celui des instructions qui ont une durée fixe : on gère la situation en ajoutant quelque circuits dans l'unité d'émission, ou en se basant sur un scoreboard. Sur certains processeurs on mémorise le temps d'exécution de l'instruction dans la fenêtre d'instruction : chaque entrée se voit ajouter un champ de latence d'instruction qui est mis à jour à chaque cycle d'horloge (un simple compteur suffit). Malheureusement, cette méthode ne fonctionne que pour les instructions de durée fixe.

Prédiction de latence

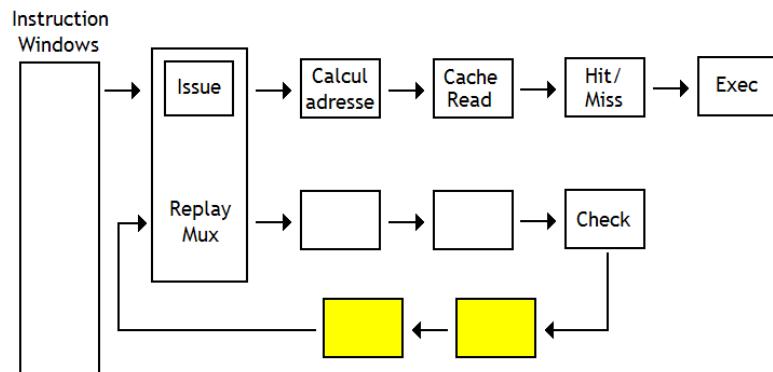
Pour gérer le cas des instructions à durée variable, le processeur peut tenter de prédire combien de temps va durer l'instruction, et agir en conséquence : il suffit de vider le pipeline si la prédiction se révèle être fausse ! Certains processeurs utilisent ainsi une unité de prédiction de latence mémoire, qui va prédire si la donnée est dans le cache ou la mémoire, et éventuellement dans quel cache elle est : cache L1, L2, etc. Cette prédiction se base sur des techniques directement inspirées des techniques de prédiction de branchement, comme des compteurs à saturation, une prédiction statique, etc. La latence prédite de l'instruction est stockée dans le fameux champ de latence d'instruction mentionné plus haut, quelques bits associés permettant de faire la différence entre durée prédite et connue avec certitude.

Pipeline à répétition

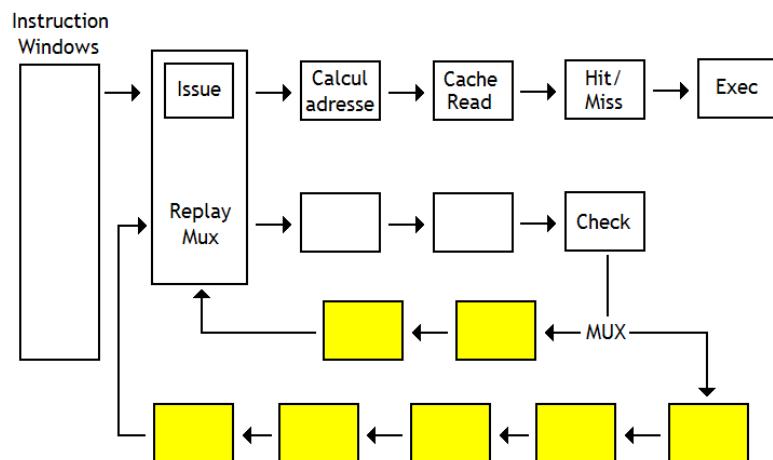
Certains processeurs sont beaucoup plus agressifs dans leurs prédictions, au point de postuler qu'aucune instruction ne fait de défaut de cache. Évidemment, ces processeurs devraient en théorie vider leurs pipelines assez souvent, mais ils utilisent une technique élégante pour gérer ces ratés de prédiction : ils utilisent ce qu'on appelle des **pipelines à répétition** (replay pipeline). Sur ces pipelines, on lance une instruction sans savoir quelle est la latence et on réexécute celle-ci en boucle tant que son résultat n'est pas valide. Pour réexécuter une instruction en boucle, le pipeline se voit ajouter une sorte de boucle, en plus du pipeline normal. Les instructions vont se propager à la fois dans la boucle et dans le pipeline normal. Les étages de la boucle servent juste à propager les signaux de commande de l'instruction, sans rien faire de spécial. Dans le pipeline qui exécute l'instruction, ces signaux de commande sont consommés au fur et à mesure, ce qui fait qu'à la fin du pipeline, il ne reste plus rien de l'instruction originale. D'où la présence de la boucle, qui sert à conserver les signaux de commande. L'étage final de la boucle vérifie que l'instruction n'a pas été émise trop tôt avec un scoreboard, et il regarde si l'instruction a donné lieu à un défaut de cache ou non. Si l'instruction a donné un bon résultat, une nouvelle instruction est envoyée dans le pipeline. Dans le cas contraire, l'instruction refera encore un tour dans le pipeline. Dans ce cas, l'unité de vérification va devoir envoyer un signal à l'unité d'émission pour lui dire « réserve un cycle pour l'instruction que je vais faire boucler ».



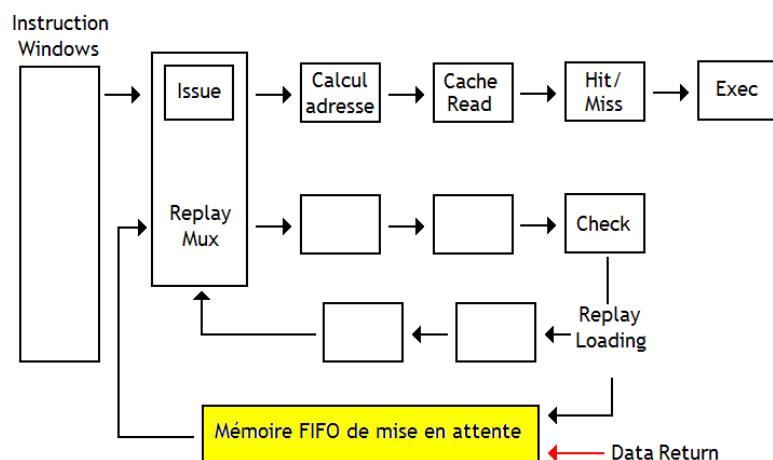
Toutefois, ce pipeline est loin d'être optimal avec des hiérarchies de cache. Prenons un exemple : un succès de cache dans le cache L1 dure 3 cycles d'horloge, un succès de cache dans le L2 dure 8 cycles, et un défaut de cache 12 cycles. Regardons ce qui se passe avec une instruction qui fait un défaut de cache dans le L1, et un succès de cache dans le L2. La boucle de 3 cycles utilisée pour le L1 ne permettra pas de gérer efficacement la latence de 8 cycles du L2 : l'instruction devra faire trois tours, soit 9 cycles d'attente, contre 8 idéalement. La solution consiste à retarder le second tour de boucle de quelques cycles, en rajoutant des étages vides dans la boucle. Dans notre exemple, il faut retarder de deux cycles : 8 cycles, dont trois en moins pour le premier tour, et trois en moins pour le trajet fenêtre d'instruction → unité de calcul.



Le même principe peut s'appliquer pour gérer les latences avec des niveaux de cache supérieurs : il faut alors utiliser plusieurs boucles de tailles différentes. Ce principe peut aussi s'appliquer dans d'autres cas assez spécifiques, dans lesquels l'instruction a été émise trop tôt, sans que cela fasse intervenir les mémoires caches. Les étages situés avant l'étage de vérification seront partagés, mais un multiplexeur se chargera de choisir vers quelle boucle envoyer l'instruction, suivant le cache dans lequel on fait défaut. Dans ces conditions, il arrive que plusieurs boucles veulent faire rentrer une instruction dans le pipeline en même temps. Pour cela, une seule boucle pourra réémettre son instruction, les autres étant mises en attente. Divers mécanismes d'arbitrage, de choix de la boucle sélectionnée pour l'émission, sont possible : privilégier la boucle dont l'instruction est la plus ancienne (et donc la boucle la plus longue) est la technique la plus fréquente. Mais dans certains cas, mettre une boucle en attente peut bloquer tous les étages précédents, ce qui peut bloquer l'émission de la nouvelle instruction : le processeur se retrouve définitivement bloqué. Dans ce cas, le processeur doit disposer d'un système de détection de ces blocages, ainsi que d'un moyen pour s'en sortir et revenir à la normale (en vidant le pipeline, par exemple).



Pour gérer au mieux les accès à la mémoire RAM, on remplace la boucle dédiée à la latency mémoire par une FIFO, dans laquelle les instructions sont accumulées en attendant le retour de la donnée en mémoire. Quand la donnée est disponible, lue ou écrite en mémoire, un signal est envoyé à cette mémoire, et l'instruction est envoyée directement dans le pipeline. Là encore, il faut gérer les conflits d'accès à l'émission entre les différentes boucles et la file d'attente de répétition, qui peuvent vouloir émettre une instruction en même temps.



On peut aussi adapter le pipeline à répétition pour qu'il gère certaines exceptions : certaines exceptions sont en effet récupérables, et disparaissent si on réexécute l'instruction. Ces exceptions peuvent provenir d'une erreur de prédiction de dépendance entre instructions (on a émis une instruction sans savoir si ses opérandes étaient prêts), ou d'autres problèmes dans le genre. Si jamais une exception récupérable a eu lieu, l'instruction doit être réexécutée, et donc réémise. Elle doit refaire une boucle dans le pipeline. Seul problème : ces exceptions se détectent à des étages très différents dans le pipeline. Dans ce cas, on peut adapter le pipeline pour que chaque exception soit vérifiée et éventuellement réémise dès que possible. On doit donc ajouter plusieurs étages de vérification, ainsi que de nombreuses boucles de réémission.

Éclaireurs matériels

Il existe d'autres techniques d'exécution dans le désordre, qui se basent sur un autre principe, relativement simple : exécuter les instructions indépendantes d'une lecture pendant que celle-ci attend la mémoire ou le cache. Mais ces architectures sont moins efficaces que les architectures à exécution dans le désordre, vu qu'elles ne peuvent que diminuer l'impact des accès mémoire, pas plus. Parmi ces techniques, les architectures déconnectées seront vues dans un chapitre à part, tandis que les éclaireurs matériels (hardware scouts) vont être abordés maintenant.

File d'attente différée

Lors d'une lecture, le processeur met en attente les instructions dépendantes de la donnée lue, mais continue d'exécuter les instructions indépendantes de celle-ci. Les instructions sont mises en attente dans une mémoire tampon, qui s'appelle la **file d'attente différée** (deferred queue). Celle-ci est souvent une simple mémoire FIFO, mais peut être plus complexe et former une véritable fenêtre d'instruction à retardement. Quand la donnée lue est disponible, écrite dans le registre de destination, les instructions mises en attente sont ré-exécutées.

Le processeur a juste à détecter les instructions dépendantes d'une lecture. Cette détection s'effectue en marquant le registre de destination de la lecture comme « invalide », du moins tant que la lecture n'a pas écrit son résultat dedans. Ensuite, chaque instruction qui a un registre invalide comme opérande sera détectée comme à mettre en attente : son registre de destination sera aussi marqué invalide. Pour marquer les registres comme valides ou invalides, on utilise un bit par registre, qui indique si le registre attribué contient une donnée valide ou non. L'ensemble de ces bits est regroupé dans un registre spécial, invisible pour le programmeur. Pour déterminer le bit de validité d'une instruction, il suffit de deux choses :

- soit il s'agit d'une lecture, et elle est marquée automatiquement comme invalide ;
- soit une instruction écrit une donnée valide dans ce registre : le registre en question est alors automatiquement marqué comme valide ;
- soit on effectue un OU entre les bits des registres d'opérandes : si un seul des registres opérande est invalide, le résultat l'est aussi.

Points de contrôle

Cependant, il arrive qu'une instruction indépendante de la lecture écrase un résultat calculé par les instructions dépendantes. Dans ce cas, l'exécution normale du programme voudrait que le registre contienne la version écrite par l'instruction indépendante, plus récente. Mais le fait de ré-exécuter les instructions dépendantes de la lecture après les instructions indépendante viole totalement ce principe ! Pour éviter tout problème, on est obligé de sauvegarder le contenu des registres à chaque lecture : le processeur peut ainsi revenir à la normale en cas de problème en remettant les registres à leur état antérieur. Et ces sauvegardes doivent être faites à chaque fois qu'on commence à spéculer, c'est-à-dire pour chaque accès mémoire : le processeur doit donc être capable de gérer un grand nombre de sauvegardes.

Fenêtres d'instruction et stations de réservation

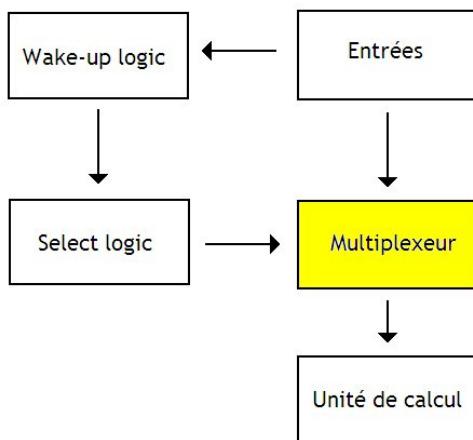
Dans le chapitre précédent, nous avons vu les principes qui se cachent derrière l'exécution dans le désordre. Nous avons notamment vu que les processeurs modernes utilisent des fenêtres d'instruction, des tampons dans lesquels les instructions à exécuter sont mises en attente. Dans certains processeurs, ces fenêtres d'instruction sont découpées en plusieurs stations de réservation indépendantes. Il est maintenant temps de voir en quoi sont faites ces fenêtres d'instruction et ces stations de réservation. Nous allons aussi voir quelle est la logique d'émission associée.

Généralités

Les fenêtres d'instruction sont composées d'**entrées**, des blocs de mémoire qui servent à stocker une instruction. Une instruction réserve une entrée après son décodage et la libère, soit quand l'instruction est exécutée, soit quand elle termine son exécution (tout dépend du processeur). Le rôle du planificateur (scheduler) est de vérifier le contenu des entrées pour savoir quelle instruction émettre. Pour faciliter la détection des dépendances, chaque station de réservation indique le registre de destination du résultat et les registres de chaque opérande. Chaque entrée possède un bit empty qui indique si elle est vide, cette information étant utile pour réserver des entrées. De plus, chaque opérande est couplée à un bit de disponibilité, qui indique si elle est disponible. Quand un opérande est écrit dans l'entrée, le bit de présence correspondant est mis à jour.



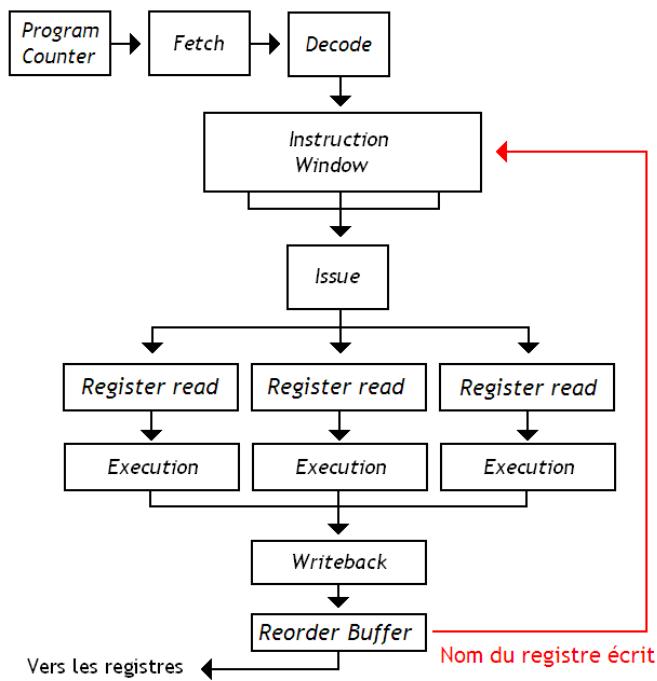
Seul un nombre limité d'entrées peut communiquer avec le chemin de données à un moment donné. Dans le cas qui va suivre, on ne peut émettre qu'une instruction à la fois : à chaque cycle, une entrée est sélectionnée et est connectée à l'entrée de l'ALU ou des stations de réservation. Cependant, seules les entrées dont tous les opérandes sont prêts peuvent émettre leurs instructions. Le processeur doit contenir un circuit qui détecte les entrées prêtes : la **logique d'éveil** (wake-up logic). Ce circuit vérifie si l'entrée n'est pas vide (le bit empty), et si tous les opérandes sont prêts (les bits de validité). Détecter la disponibilité d'une opérande, pour mettre à jour les bits de validité, demande de surveiller les écritures dans les registres. Une fois que la logique d'éveil a fait son travail, une entrée est choisie, ce choix doit être le plus pertinent possible, pour des raisons de performances. Ce rôle est dévolu à un circuit qu'on appelle la **logique de sélection**, ou select logic.



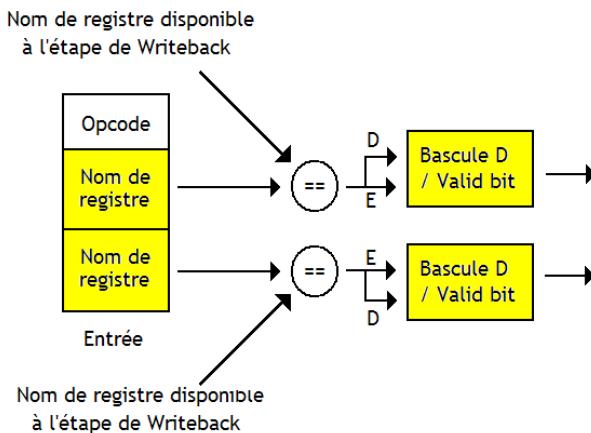
CAM-based

Certaines fenêtres d'instruction se basent sur des mémoires associatives, des mémoires abordées il y a quelques chapitres. Pour rappel, ces mémoires servent à accélérer la recherche de données dans un ensemble. Or, une fenêtre d'instruction vérifie régulièrement si chaque entrée est prête : il s'agit d'un processus de recherche, ce qui fait que les mémoires associatives sont donc tout indiquées.

Comme vous le savez, les signaux de commande d'une instruction sont propagés avec celle-ci dans le pipeline, le nom du registre de destination ne faisant pas exception. Ce nom de registre est alors envoyé à la fenêtre d'instruction lors de l'écriture du résultat dans les registres. S'il y a correspondance, l'opérande est disponible. On peut adapter cette méthode pour tenir compte du contournement assez simplement.



En conséquence, le circuit de détection des dépendances est constitué d'un grand nombre de comparateurs : un par champ « nom de registre » dans chaque entrée. Si l'opérande est disponible, le bit de validité associé doit être positionné à 1 (il est remis à 0 lorsque l'instruction quitte l'entrée).



Le problème avec des fenêtres d'instruction de ce type est leur forte consommation énergétique, ainsi que le grand nombre de portes logiques utilisées, qui deviennent prohibitif pour des fenêtres d'instruction un peu grosses. Pour résoudre ce problème, certains ont optimisé les comparateurs. D'autres ont tenté de profiter du fait que la majorité des bits dans les entrées sont composés de zéros. Bref, les optimisations purement matérielles sont légion.

Optimisation sur le nombre d'opérandes

Certains chercheurs ont remarqué que le nombre d'opérandes varie suivant l'instruction : certaines n'ont besoin que d'un seul opérande. De plus, il arrive qu'une opération tout juste décodée ait déjà un ou plusieurs de ses opérands de prêts. Cette constatation permet d'éviter d'utiliser des comparateurs pour les opérandes déjà prêts. Par exemple, on peut parfaitement utiliser trois fenêtres d'instruction :

- une pour les instructions dont tous les opérandes sont prêts, mais qui sont quand même mises en attente, sans comparateur ;
- une pour les instructions dont un seul opérande manque, qui n'utilise qu'un comparateur par entrée ;
- une pour les instructions dont deux opérandes manquent à l'appel, qui utilise deux comparateurs par entrée.

C'est beaucoup plus économique que d'utiliser une seule grosse fenêtre d'instruction qui contiendrait autant d'entrées que les trois précédentes réunies, avec deux comparateurs par entrée. Certains sont même allés plus loin, et ont proposé de supprimer la fenêtre d'instruction avec deux opérandes par entrée. Les instructions dont deux opérandes sont inconnus au décodage sont stockées dans la fenêtre d'instruction pour instructions avec un comparateur par entrée. Un circuit de prédiction se charge alors de prédire l'opérande manquant, cette prédiction étant vérifiée plus loin dans le pipeline.

Optimisations sur la détection des opérandes prêts

D'autres chercheurs ont conservé une fenêtre d'instruction unique, avec autant de comparateurs par entrée qu'il y a d'opérandes possibles par instruction. Simplement, le processus de détection des opérandes prêts est légèrement ralenti : on ne vérifie qu'un opérande par cycle. Pour vérifier les deux opérandes d'une entrée, on doit attendre deux cycles.

Optimisation sur la segmentation de la fenêtre d'instruction

Enfin, certains chercheurs ont proposé des fenêtres d'instruction segmentées. Les instructions circulent à chaque cycle d'un segment vers le suivant : en conséquence, certains segments conservent les instructions les plus anciennes, un autre les instructions les plus jeunes, etc. Seul le

denier segment, celui qui contient les instructions les plus vieilles, peut émettre une instruction : la détection des opérandes se fait seulement dans le dernier segment, qui contient les instructions les plus anciennes. On économise ainsi beaucoup de comparateurs.

Optimisation du pipeline et spéculation

Certains chercheurs ont tenté de pipeliner l'étape de sélection des opérandes, ainsi que l'étage d'arbitrage. Mais en faisant cela, il faut plusieurs cycles pour détecter qu'une instruction a ses opérandes prêts, ce qui pose problème face à de nombreuses dépendances RAW. Pour éviter de trop perdre en performances, certains chercheurs ont décidé d'utiliser des techniques pour prédir quelles seront les instructions dont les opérandes seront bientôt prêts. Si détecter qu'une instruction est prête prend n cycles, le processeur devra tenter de prédir la future disponibilité des opérandes n cycles en avance pour obtenir des performances optimales.

Tables indexées

Certaines optimisations cherchent à supprimer les comparateurs présents dans chaque entrée en remplaçant la fenêtre d'instruction par une mémoire RAM, dont chaque mot mémoire correspondrait à une entrée.

Recherche directe par étiquette

La première de ces techniques, la **recherche directe par étiquette** (direct tag search) fut créée par Weiss et son collègue Smith. De base, chaque instruction produit un résultat, qui devra être rapatrié dans une ou plusieurs entrées de la fenêtre d'instruction. L'optimisation de la recherche directe par étiquette fonctionne dans le cas où le résultat de l'instruction n'est utilisé que par une seule entrée, et pas plusieurs.

Le principe est simple : chaque champ « opérande » d'une entrée est adressable via le nom de registre de cette opérande. Pour faire le lien entre entrée et nom de registre, la recherche directe par étiquette ajoute une table de correspondances matérielle, une mémoire RAM qui mémorise les adresses des champs « opérande » des entrées. Les adresses envoyées dans cette mémoire sont les noms de registres des résultats. Lors de l'émission, la table de correspondances est mise à jour. Les numéros des champs « opérande » réservés lors de l'émission sont mémorisés dans les mots mémoire qui correspondent aux deux registres sources. Toutefois, une petite vérification est faite lors de l'émission : si il y a déjà une correspondance dans la table pour un registre source, alors une autre instruction compte lire ce registre. Pour éviter d'écraser les données de l'instruction précédente dans la table de correspondances, l'émission de l'instruction est bloquée.

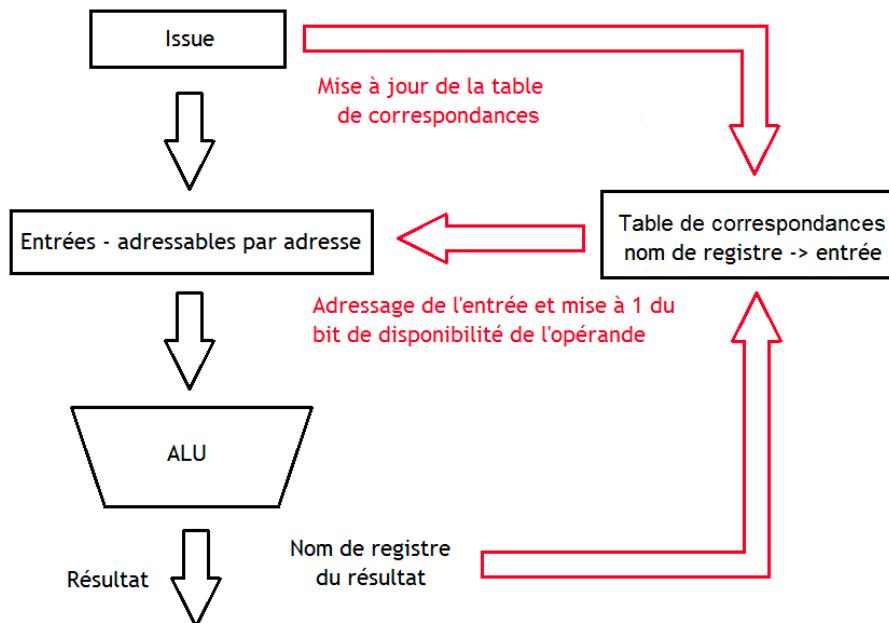


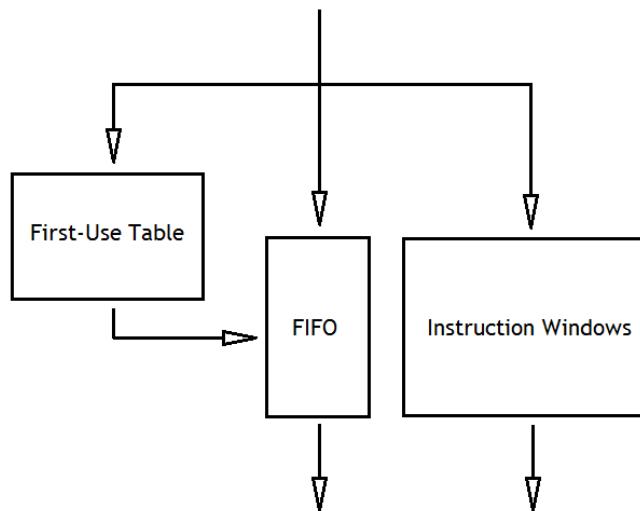
Table de première utilisation

Dans leur papier de recherche nommé « A Low-Complexity Issue Logic », Ramon Canal et Antonio González ont trouvé une solution assez similaire. Leur algorithme se base sur trois structures matérielles :

- une mémoire RAM nommée la table de première utilisation (first-use table) ;
- une mémoire de type FIFO, qui sert de file d'attente pour les instructions ;
- une fenêtre d'instruction normale.

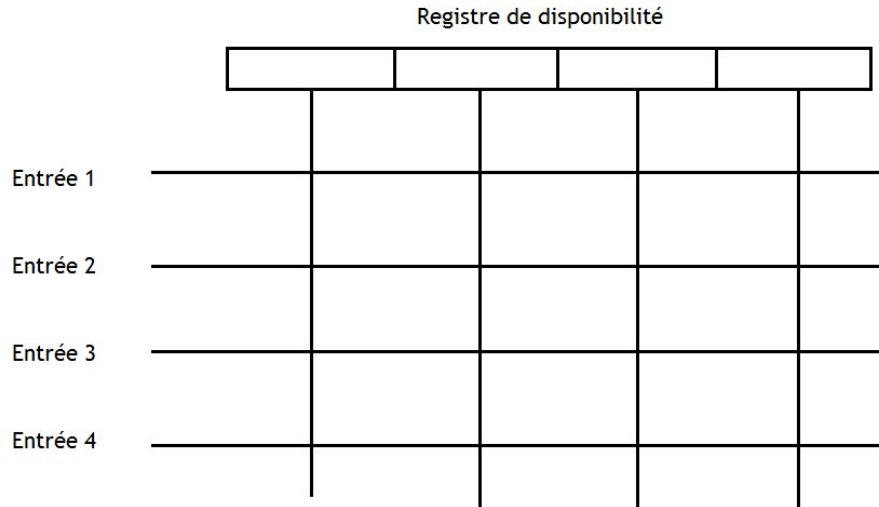
Le fonctionnement de cet algorithme peut se résumer en trois cas.

- Si une instruction a tous ses opérandes disponibles, celle-ci est placée dans la mémoire FIFO.
- Deuxième cas : l'instruction lit une valeur/donnée déjà lue/utilisée par une précédente instruction. Dans ce cas, l'instruction est placée dans la fenêtre d'instruction.
- Dans le dernier cas, l'instruction va lire une donnée qui n'a jamais été lue auparavant : le résultat à lire a été écrit dans un registre, mais pas encore été utilisé. Dans ce cas, l'instruction est mémorisée dans la table de première utilisation. Le registre lu en question servira d'adresse, et déterminera l'adresse de l'instruction dans la table de première utilisation. L'opcode de l'instruction sera stocké dans cette table de première utilisation avec le numéro de registre de l'opérande restant. Quand l'instruction qui produit la donnée à lire terminera, la donnée sera disponible. Le numéro du registre de destination sera alors envoyé en entrée d'adresse de la table de première utilisation : l'instruction qui consomme le résultat sera alors directement accessible. L'entrée sera alors mise à jour, et le registre de l'opérande restant est analysé. Si celui-ci ne pointe pas vers une instruction, alors l'instruction mise à jour a tous ses opérandes disponibles, elle est déplacée dans la mémoire FIFO.

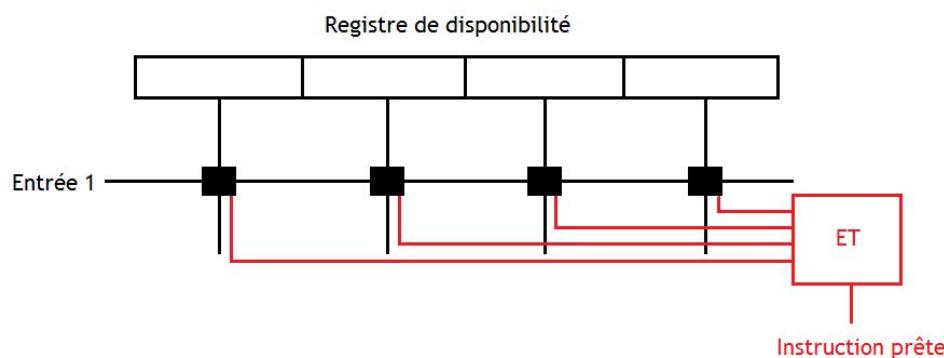


Éveil par matrice de bits

Une dernière solution se passe totalement de comparateurs complexes dans la fenêtre d'instruction. Le processeur mémorise juste les registres qui contiennent une donnée valide dans un registre de disponibilité d'opérandes, avec un bit par registre qui indique si celui-ci est valide ou non. Cette technique détecte la disponibilité des opérandes avec l'aide d'une **matrice de bits**, où chaque ligne correspond à une entrée et chaque colonne à un registre. À chaque croisement entre une ligne et une colonne, on trouve un bit. Si le bit de la ligne n et de la colonne m est à 1, cela veut dire : l'instruction dans l'entrée n a besoin de lire le registre m. S'il est à 0, alors cela veut dire que l'instruction stockée dans la ligne n n'a pas besoin de la donnée dans le registre m.

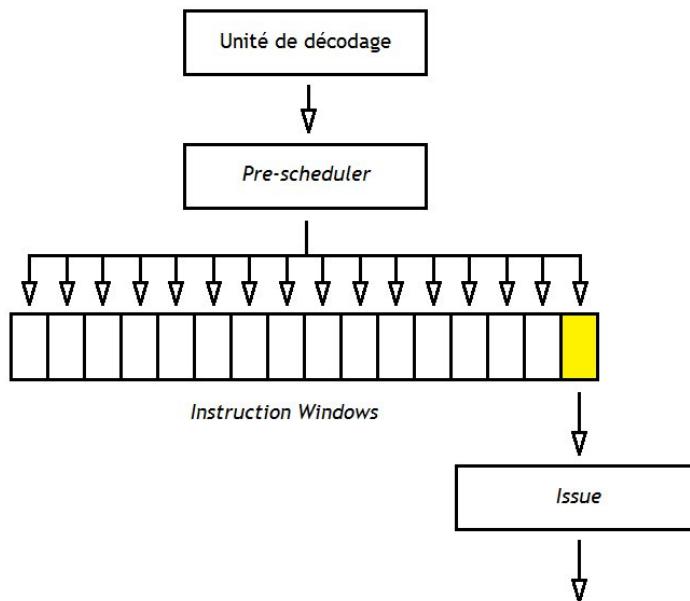


Lorsqu'une instruction réserve une entrée, elle initialise la ligne en fonction des registres qu'elle souhaite lire. Pour vérifier si une instruction a ses opérandes prêts, il suffit de comparer ce registre de disponibilité à la ligne qui correspond à l'instruction. L'instruction est prêt à s'exécuter quand les deux sont égaux. Cela demande donc d'ajouter, à chaque intersection ligne-colonne, un comparateur.



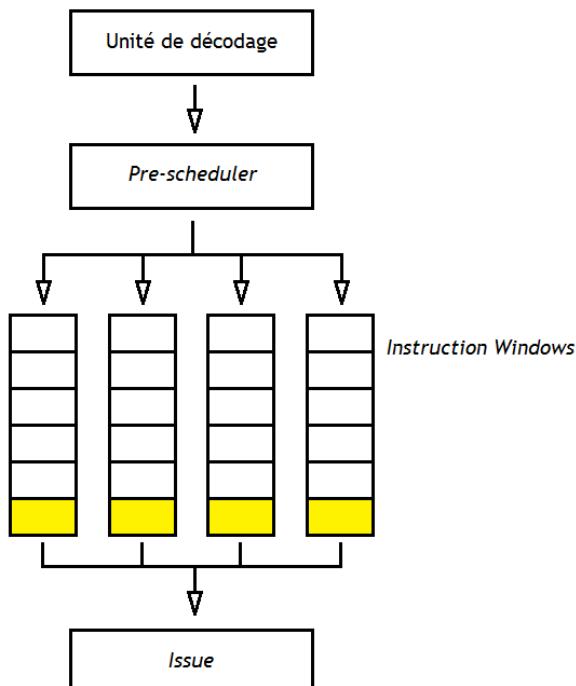
Préplanification

Avec la **préplanification** (prescheduling), la fenêtre d'instruction est composée de mémoires tampons dans lesquelles les instructions sont triées dans l'ordre d'émission. Il n'y a pas de logique d'émission proprement dite, celle-ci se bornant à vérifier si l'instruction située au début de la fenêtre d'instruction peut s'exécuter. Par contre, le préplanificateur détecte les dépendances et en déduit où insérer les instructions dans le tampon d'émission, à l'endroit le plus adéquat.



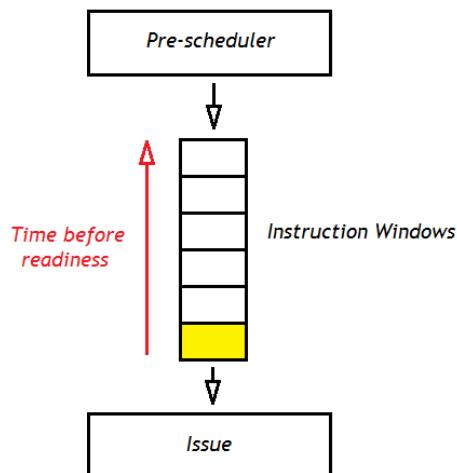
FIFO multiples

La technique de préplanification la plus simple consiste à scinder la fenêtre d'instruction en plusieurs FIFO. Quand une instruction a une dépendance avec une instruction présente dans une FIFO, elle est ajoutée dans celle-ci. Ainsi, le préplanificateur se charge de détecter les chaînes d'instructions dépendantes, et place les instructions d'une même chaîne dans une seule FIFO. Si jamais l'instruction située au tout début de la FIFO n'a pas ses opérandes disponibles, alors la FIFO est bloquée : l'instruction attend que ses opérandes soient disponibles, bloquant toutes les instructions précédentes. Mais les autres FIFO ne sont pas bloquées, laissant de la marge de manœuvre.



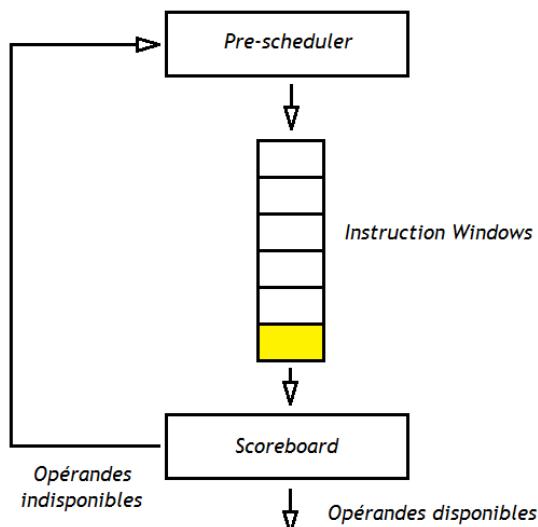
Tampon trié

Une autre technique, celle du tampon trié, trie les instructions selon le temps d'attente avant leur exécution : les instructions qui sont censées s'exécuter bientôt étant au tout début de cette mémoire, tandis que les instructions qui s'exécuteront dans un long moment sont situées vers la fin. Des techniques similaires se basent sur le même principe, avec un ordre des instructions différent, tel les relations de dépendance entre instructions. Comme précédemment, si jamais l'instruction au bout de la mémoire, celle prêtée à être émise n'a pas ses opérandes prêts, elle est mise en attente et bloque toute la fenêtre d'instruction. Le temps avant exécution dépend du temps de calcul de ces opérandes et du temps avant exécution des instructions qui produisent ces opérandes. Évidemment, celui-ci n'est pas toujours connu, notamment pour les instructions qui accèdent à la mémoire, ou des instructions dépendantes de celles-ci. Pour résoudre ce genre de problèmes, le préplanificateur doit faire des prédictions.



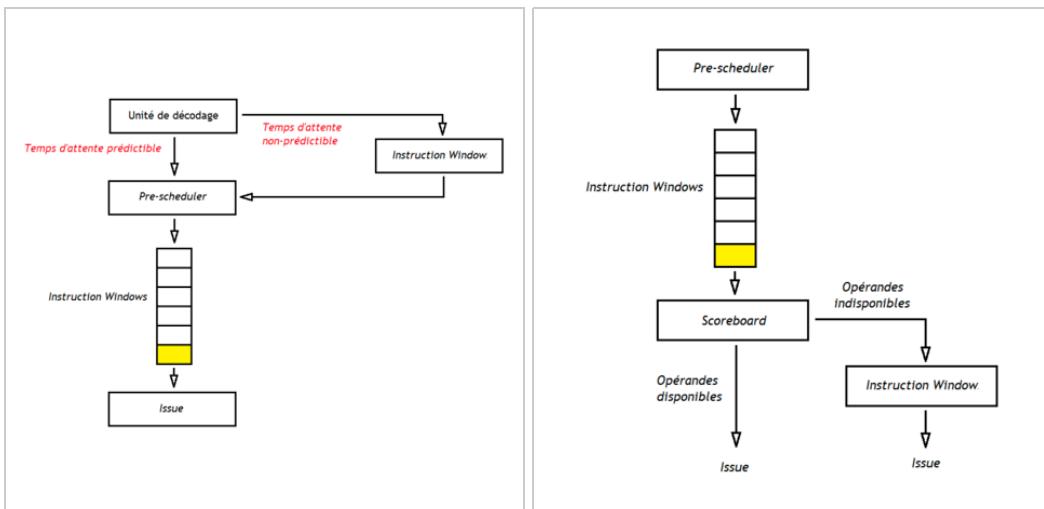
Tampon de scoreboard

La technique précédente peut être améliorée. Au lieu de bloquer totalement la fenêtre d'instruction lorsqu'une instruction émise n'a pas tous ses opérandes disponibles, on peut réinsérer celle-ci dans la fenêtre d'instruction. Ainsi, l'instruction fautive ne bloque pas la fenêtre d'instruction, et retente sa chance autant de fois qu'il le faut.



Techniques hybrides

Il est possible de marier les techniques précédentes (préplanification et autres formes de fenêtres d'instruction), en utilisant deux fenêtres d'instruction : une gérée par la préplanification, et une autre pour les instructions dont le temps d'attente ne peut pas être déterminée. Certains processeurs utilisent ainsi une fenêtre d'instruction qui précède la préplanification, afin de gérer les temps d'attente des instructions d'accès mémoire et des instructions dépendantes. Quand le temps d'attente d'une instruction devient connu, celle-ci est migrée dans le tampon de préplanification. Une autre technique consiste à d'abord tenter de prédire le temps d'attente de l'instruction avec la préplanification, et de basculer sur une fenêtre d'instruction normale si la prédiction s'avère fausse.



Le renommage de registres

Pour améliorer l'exécution dans le désordre, les concepteurs d'ordinateurs ont étudié des méthodes pour éliminer la grosse majorité des dépendances de données lors de l'exécution. Dans ce que j'ai dit précédemment, j'ai évoqué trois types de dépendances de données. Les dépendances RAW sont des dépendances contre lesquelles on ne peut rien faire : ce sont de « vraies » dépendances de données. Quelle que soit la situation, ces dépendances imposent un certain ordre d'exécution de nos instructions. Mais les dépendances write after write et write after read sont de fausses dépendances. Elles proviennent du fait que deux instructions doivent se partager le même registre ou la même adresse mémoire, du fait qu'un emplacement mémoire est réutilisé pour stocker des données différentes à des instants différents. Bien évidemment, l'imagination débridée des concepteurs de processeur a trouvé une solution pour supprimer ces dépendances : le renommage de registres.

Si on change l'ordre de deux instructions ayant une dépendance de données WAW ou WAR, les deux instructions utilisent le même registre pour stocker des données différentes en même temps. Manque de chance, un registre ne peut contenir qu'une seule donnée, et l'une des deux sera perdue. Une solution simple vient alors à l'esprit : conserver les différentes versions d'un même registre dans plusieurs registres séparés, et choisir le registre adéquat à l'utilisation. À chaque fois qu'on veut écrire, l'écriture est redirigée vers un autre registre. Et toutes les lectures vers cette version du registre sont redirigées vers ce registre supplémentaire. Une fois qu'une donnée, une version d'un registre, devient inutile, le registre devient réutilisable. On voit bien qu'un seul registre architectural correspondra en réalité à plusieurs registres, chacun contenant le résultat d'une écriture. On a besoin non seulement de nos registres architecturaux, mais aussi de registres supplémentaires cachés, adressables. Il faut ainsi faire la distinction entre :

- **registres architecturaux**, définis par l'architecture extérieure du processeur, mais fictifs avec le renommage de registres ;
- **registres physiques**, réellement présents dans notre processeur, bien plus nombreux que les registres architecturaux.

Le renommage de registres permet à deux registres physiques de devenir le même registre architectural. Le scoreboarding vu au chapitre précédent ne gère pas le renommage de registres. On peut toutefois adapter celui-ci, en augmentant la taille du banc de registres, et en ajoutant un étage chargé d'effectuer le renommage des registres après les unités de décodage. Avec cette technique, tous les registres virtuels sont stockés dans un seul gros banc de registres, et il n'y a pas de registres architecturaux. Les autres techniques utilisent des registres virtuels et architecturaux séparés. Chaque registre virtuel contiendra une valeur temporaire, destinée à être recopiée dans un registre architectural après un certain temps. Pour faire simple, une fois qu'un registre virtuel devient inutile, une fois les dépendances résolues, on copie le registre architectural correspondant.

L'unité de renommage

Les registres architecturaux sont identifiés par des noms de registre, tout comme les registres physiques. Les noms de registre des registres physiques sont appelés des tags. Pour attribuer un registre architectural à un registre physique, il suffit de remplacer le nom du registre architectural par le tag du registre physique attribué : le registre architectural est renommé. Vu qu'il y a plus de registres physiques que de registres architecturaux, le tag d'un registre physique est plus grand que le nom d'un registre architectural. Ce remplacement est effectué dans un étage supplémentaire du pipeline, intercalé entre le décodage et l'émission.



Table de correspondances de registres

Ce remplacement est effectué par un circuit spécialisé, la **table de correspondance de registres** (register map table). Il existe deux façons pour l'implémenter. La plus ancienne consiste à utiliser une mémoire associative dont le tag contient le nom du registre architectural, et la donnée le nom du registre physique. Sur les processeurs plus récents, on utilise une mémoire RAM, dont les adresses correspondent aux noms de registres architecturaux, et dont le contenu d'une adresse correspond au nom du registre physique associé. On n'a alors qu'une seule correspondance entre registre physique et registre architectural, mais cela ne pose pas de problème si on renomme les instructions dans l'ordre d'émission (ce qui est toujours fait). Ce circuit est conceptuellement divisé en deux gros sous-circuits, qui sont parfois fusionnés en un seul, mais on fera comme si dans les explications qui vont suivre.

Lorsqu'on attribue un nouveau registre virtuel pour un registre architectural, on doit prendre un registre virtuel inutilisé. La table de correspondances de registres doit donc connaître la liste des registres vides dans une petite mémoire : la **liste de disponibilités** (free list). Lorsqu'un registre virtuel n'est plus utilisé par les prochaines instructions, on peut réutiliser le registre comme bon nous semble : il passe alors dans la liste de disponibilités. Lorsqu'un registre est attribué, il quitte la liste de disponibilités. Détecter les registres réutilisables est assez complexe et peut se faire de plusieurs façons, suivant le nombre d'instructions démarrées simultanément. Une solution consiste à attribuer un compteur à chaque registre : ce compteur est incrémenté/décrémenté quand une instruction entre/quitte le tampon de réordonnancement avec ce registre comme opérande.

Maintenant que l'on a réussi à renommer les registres de destination de notre instruction avec la liste de disponibilités, on doit renommer les registres sources, qui contiennent les opérandes. C'est le rôle de la **table d'alias de registres** (register alias table), une mémoire qui contient le registre virtuel associé à chaque registre architectural. Cette correspondance est mise à jour à chaque fois qu'un registre vide est réquisitionné pour servir de registre de destination pour une instruction. À ce moment, la table d'alias de registres est mise à jour : le registre virtuel choisi dans la liste de disponibilités est attribué au registre architectural de destination du résultat. Il suffira de réutiliser cette correspondance par la suite. Cependant, le résultat ne sera pas immédiatement disponible pour notre instruction. Il ne le sera qu'après avoir été calculé. C'est seulement à ce moment-là que notre registre virtuel contiendra une donnée valide et sera utilisable par d'autres instructions. Pour savoir si notre registre contient une donnée valide, la table d'alias de registres contient un bit de validité pour chaque registre architectural. Ce bit de validité est mis à jour lors de l'écriture du résultat dans le registre virtuel correspondant.

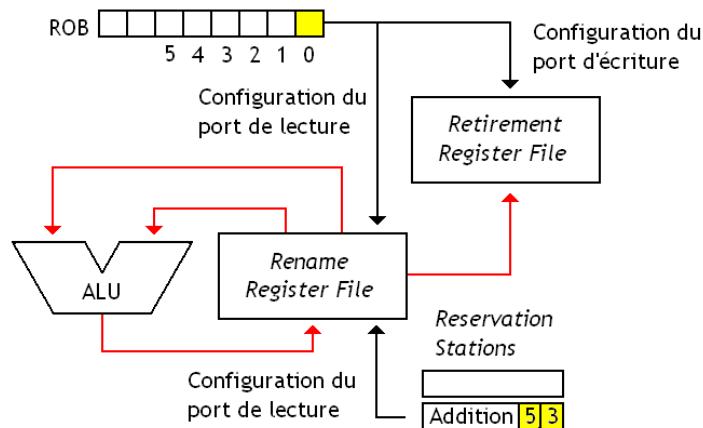
Récupération après une prédiction

Quand une exception ou une mauvaise prédiction de branchement a lieu, les registres virtuels contiennent des données potentiellement invalides, contrairement aux registres architecturaux. En cas de mauvaise prédiction de branchement, la table d'alias de registres est partiellement vidée : on n'a strictement aucune correspondance entre le registre architectural et un registre virtuel invalidé par la prédiction de branchement. De plus, on doit remettre la table d'alias de registres et la table de disponibilités dans l'état antérieur au chargement de l'instruction qui a déclenché la mauvaise prédiction de branchement ou l'exception matérielle. Et cela peut se faire de différentes manières.

On peut stocker dans le tampon de réordonnancement ce qui a été modifié dans l'unité de renommage de registres par l'instruction correspondante. Ainsi, lorsqu'une instruction sera prête à valider, et qu'une exception ou mauvaise prédiction de branchement aura eu lieu, les modifications effectuées dans l'unité de renommage de registres seront annulées les unes après les autres. Une autre solution consiste à garder une copie valide de la table d'alias de registres dans une mémoire à part, pour la restaurer au besoin. Par exemple, si jamais notre table d'alias de registres détecte un branchement, son contenu est sauvegardé dans une mémoire intégrée au processeur. On peut utiliser plusieurs mémoires de sauvegarde de ce type, pour gérer une succession de branchements.

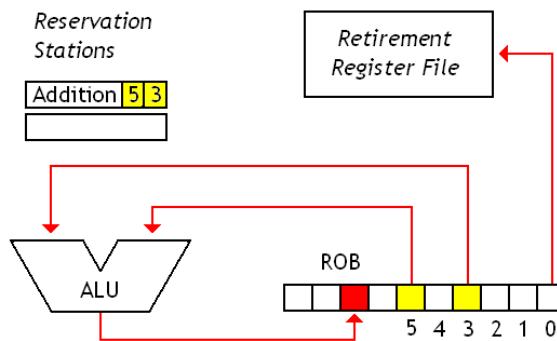
Fichier de registres renommés

Certains processeurs utilisent un banc de registres pour les registres architecturaux, et un autre pour les registres virtuels : ce dernier est appelé le banc de registres renommés (rename register file), tandis que le premier est appelé le banc de registres architecturaux (retirement register file). Tampons de réordonnancement et stations de réservation contiennent des noms de registres virtuels ou réels qui contiennent la donnée. Quand le tampon de réordonnancement veut écrire une donnée dans les registres architecturaux, il doit lire le résultat à écrire depuis le banc de registres renommés. Pareil pour le chargement d'une donnée dans les stations de réservation, qui a lieu depuis ce dernier.



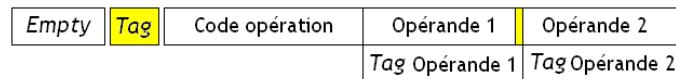
Tampon de réordonnancement

Certains processeurs font le renommage dans le tampon de réordonnancement. Vu que le résultat de l'opération est mémorisé dans le tampon de réordonnancement, l'entrée correspondante peut servir de registre virtuel. Chaque entrée devient ainsi adressable. Avec cette technique, les stations de réservation ne mémorisent pas un nom de registre architectural, mais l'adresse de l'entrée du tampon de réordonnancement qui contient le résultat voulu. Quand une instruction a tous ses opérandes de prêts, ceux-ci sont lus depuis le tampon de réordonnancement (ou depuis la mémoire, en cas de mauvaise prédition de branchement). Pour information, cette technique de renommage était utilisée dans d'anciens processeurs commerciaux, comme les Pentium Pro, le Pentium II, ou encore le Pentium III. Il arrive que certains processeurs fusionnent le tampon de réordonnancement et les stations de réservation dans un seul circuit.

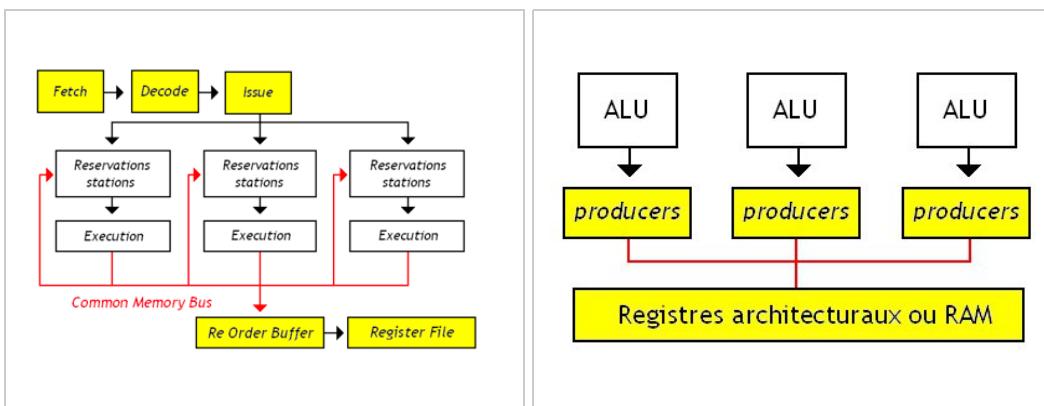


Renommage dans les stations de réservation

Certains processeurs renomment les registres dans les stations de réservation, chaque champ opérande d'une entrée servant de registre virtuel. Les noms de registre de l'entrée sont modifiés de manière à mémoriser des noms de registres virtuels, des tags.



La sortie des unités de calcul est reliée aux entrées des stations de réservation via un bus, le **bus commun d'ordonnancement de la mémoire** (common memory-ordering bus). Ce bus transmet à la fois la donnée aux entrées, mais aussi le nom du registre de destination, histoire de détecter les dépendances RAW. Pour faciliter l'arbitrage de ce bus, une seule unité de calcul peut envoyer des données sur ce bus. Les autres doivent mettre en attente leurs résultats dans une mémoire tampon, soit un registre, soit une FIFO. Cette mémoire est appelée un producteur (producer).



Algorithme de Tomasulo.

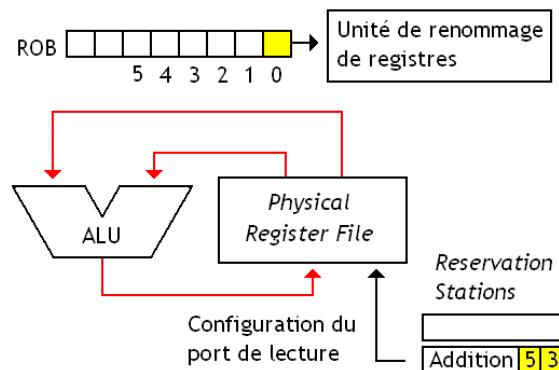
Producteurs en sortie d'une ALU.

Fenêtres d'instruction

Certains processeurs renomment les registres dans la fenêtre d'instruction, chaque entrée de celle-ci servant de registre virtuel. Les entrées doivent toutefois être adaptées, afin de stocker les opérandes de l'instruction à stocker, en plus de l'opcode, des noms de registres, des bits de présence, etc. Le mécanisme de détection des dépendances doit aussi être adapté afin de relier les entrées au bus commun d'ordonnancement de la mémoire : sans cela, les résultats d'un calcul ne peuvent pas être contournés jusqu'aux entrées dans la fenêtre d'instruction.

Banc de registres physiques

On peut aussi utiliser un seul banc de registres pour les registres architecturaux et virtuels : c'est un **banc de registres physiques** (physical register file). Aujourd'hui, presque tous les processeurs utilisent ce genre de structure pour faire du renommage de registres, pour plusieurs raisons. Premièrement, cette méthode est économique en énergie. Il faut dire qu'elle ne demande pas de copier des registres entre un banc de registres renommés et un banc de registres architecturaux, ce qui a en outre un avantage en terme de performance ou de simplification des circuits du processeur. Et c'est sans compter que le tampon de réordonnancement et les stations de réservation contiennent des pointeurs vers nos registres, ce qui prend moins de place que stocker directement les données. Libérer un registre est cependant plus complexe : on ne peut le faire que lorsqu'on est certain qu'aucune instruction n'ira lire ce registre. Pour cela, le mieux est d'attribuer un compteur de lectures pour chaque registre. À noter que pour maintenir des exceptions précises, on est obligé d'attendre que la dernière instruction qui lit le registre ait validé.



Un autre avantage est certaines opérations deviennent inutiles si le renommage de registres implémente certaines optimisations. Il s'agit des instructions de copie d'un registre dans un autre (les MOV) ou d'échange entre deux registres (XCHG). Après la copie de notre registre dans un autre, le contenu des deux registres est identique tant qu'aucun des deux registre ne subit d'écriture. On peut alors utiliser un seul registre physique pour la valeur recopiée et l'original, toute lecture de ces deux valeurs lisant celui-ci. Lors d'une écriture dans un de ces deux registres, il suffira d'utiliser un registre physique vide pour stocker la donnée à écrire et l'associer avec le nom de ce registre.

Le même genre d'optimisation peut être effectué avec des instructions inutiles, dont le résultat vaut soit zéro soit un des opérandes. On peut remarquer que ces calculs correspondent dans la plupart des cas à des calculs dont un opérande vaut 0 ou 1 : des décalages par 0, les additions et soustractions avec 0, la multiplication par 0 ou 1, et bien d'autres. En utilisant intelligemment le renommage de registres, on peut faire en sorte que ces calculs ne soient pas effectués. Les techniques qui permettent cela sont des techniques dites de **calcul trivial** (trivial computation). La détection des calculs simplifiables demande de comparer les opérandes avec 0 ou 1, via un paquet de comparateurs regroupés dans une unité de détection des calculs triviaux. Leur simplification se fait dans la table de renommage de registres, le registre de destination de l'instruction simplifiée étant renommé pour pointer sur l'opérande/résultat. Si le résultat est nul, on considère que la correspondance est invalide et on met le registre physique à une valeur invalide, similaire au pointeur NULL du C. Si un calcul lit un registre physique invalide, celui-ci est automatiquement simplifié par l'unité de renommage, l'autre opérande étant alors attribué automatiquement comme registre de destination du résultat dans la table de renommage. Mais cette technique a tendance à modifier la latence des instruction, qui est réduite après simplification. Cela pose problème au niveau des circuits d'émission et du planificateur, qui n'aiment pas les latences variables, comme on l'a vu il y a quelques chapitres.

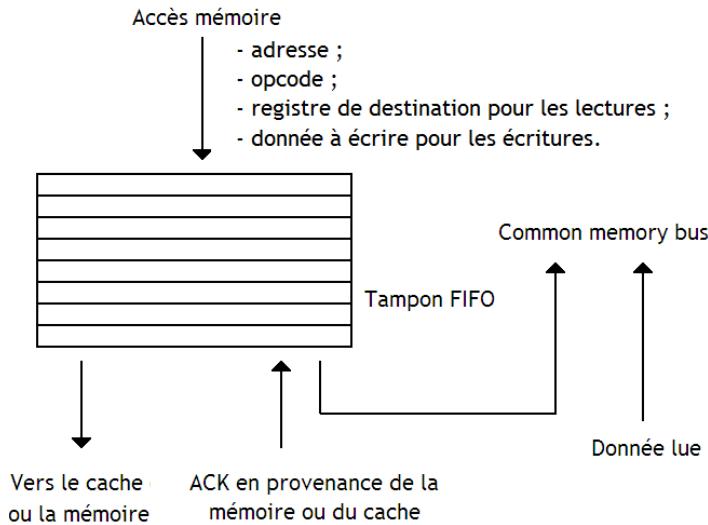
Renommage physique-virtuel

Les méthodes mentionnées au-dessus ont un léger problème : beaucoup de registres physiques sont gâchés, car attribués à une instruction dès l'étage de renommage et libérés lorsque on est certain qu'aucune instruction ne lira le registre physique attribué. Et parfois, les registres sont alloués trop tôt, alors qu'ils auraient pu rester libres durant encore quelques cycles. C'est notamment le cas quand l'instruction renommée attend un opérande dans la fenêtre d'instruction, ou quand le résultat est en cours de calcul dans l'unité de calcul. Ces situations ont toutes la même origine : le renommage a lieu tôt dans le pipeline, pour garder les dépendances entre instructions. Mais dans les faits, rien n'oblige à utiliser les registres architecturaux pour conserver les dépendances. On peut tout simplement attribuer un tag à chaque résultat d'instruction, tag qui ne correspond pas à un registre architectural. Et ce tag sera alloué à un registre architectural le plus tard possible, quand l'instruction fournira son résultat. Ce genre de méthodes a été formalisé avec ce qu'on appelle la technique du **banc de registres physiques-virtuels** (physical-virtual register file).

Cette méthode demande d'ajouter une seconde table de correspondance, qui fait le lien entre le tag et le registre physique. De plus, la table d'alias de registres doit être modifiée : elle ne doit pas seulement faire la correspondance entre le nom de registre et le tag, mais aussi avec le registre physique s'il est déjà attribué. Ainsi, lors du renommage en sortie du décodeur, on peut renommer l'instruction avec les registres physiques si ceux-ci sont connus lors du renommage, et renommer avec des tags dans le cas contraire. Il faut noter que cette méthode a un léger problème : quand une instruction termine et que le résultat doit se voir attribuer un tag, il se peut parfaitement qu'il n'y ait plus de registre physique de libre. Les solutions pour régler ce problème sont assez complexes, aussi je n'en parlerai pas ici.

Désambigüisation de la mémoire

L'exécution dans le désordre modifie l'ordre des instructions pour gagner en efficacité, et le renommage de registres améliore la situation en supprimant certaines dépendances. Mais cela n'est valable que pour les instructions travaillant sur des registres, pas sur celles qui accèdent à la mémoire. Jusqu'à présent, nous n'avons vu que des processeurs qui n'utilisent pas d'exécution dans le désordre pour les accès mémoire. Les lectures et écritures sont émises dans l'ordre du programme. Au niveau du processeur, elles sont accumulées dans une FIFO, en attendant que la mémoire RAM ou le cache soient libres, celle-ci servant de station de réservation dédiée aux accès mémoires. Chaque entrée de la FIFO contient des bits qui indiquent si la donnée à lire ou écrire est disponible, ou si l'adresse d'accès est connue. L'unité d'accès mémoire vérifie à chaque cycle si un accès mémoire est en cours et si les adresses et la donnée à écrire sont disponibles. Une fois l'accès mémoire terminé, la mémoire ou le cache vont envoyer un signal pour indiquer que l'accès est fini. Dans le cas d'une lecture, la donnée lue est alors envoyée à l'étage d'enregistrement, de même que le registre de destination de la donnée lue.



Désambigüisation de la mémoire

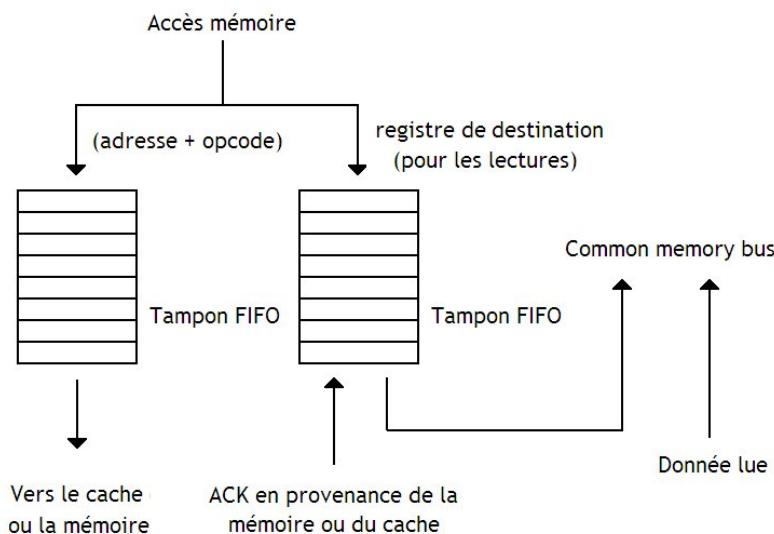
Cette méthode est conservatrice et ne tient pas compte du fait que des accès mémoires indépendants peuvent être réordonnancés. Or, diverses techniques réordonnent les accès mémoires, exactement comme l'exécution dans le désordre le fait pour les autres instructions : ces techniques sont ce qu'on appelle des techniques de **désambigüisation de la mémoire** (memory disambiguation).

File d'écriture

Pour éviter tout problème avec la désambigüisation de la mémoire, deux conditions doivent être remplies. Premièrement, on ne doit pas écrire dans la mémoire tant que l'on ne sait pas si l'écriture a bien lieu, pour éviter tout problème en cas de mauvaise prédiction de branchement ou d'exception. Pour cela, il suffit de mettre en attente les écritures dans une FIFO, une sorte de tampon de réordonnancement dédié aux écritures : la **file d'écriture** (store queue). Celle-ci mémorise, pour chaque écriture, l'adresse et la donnée de l'écriture, ainsi que des bits pour indiquer la disponibilité de celles-ci. Ensuite, une lecture doit lire la donnée provenant de l'écriture la plus récente qui a eu lieu à cette adresse mémoire. Ce qui peut se régler avec plusieurs solutions : lecture depuis la file d'écriture, blocage des lectures tant qu'une écriture est en attente dans la file d'écriture, ou autre.

Lecture par contournement

S'il est impossible de déplacer une lecture avant une écriture si les deux instructions ont une dépendance (partagent la même adresse), cela est parfaitement possible dans le cas contraire. La technique de la **lecture par contournement** (load bypassing) conserve l'ordre entre écritures et entre lectures, mais pas l'ordre entre écritures et lectures. Dit autrement, si une lecture n'a aucune dépendance avec les écritures mises en attente dans la file d'écriture, on la démarre immédiatement. Elle doit être mise en attente dans le cas contraire. Pour cela, on rajoute un ensemble de stations de réservation dédiées aux lectures : la **file de lecture** (load queue).



Réacheminement écriture-vers-lecture

On peut aussi éviter de bloquer la lecture en lisant la donnée depuis la file d'écriture, sans attendre que l'écriture soit terminée. Cette optimisation s'appelle le **réacheminement écriture-vers-lecture** (store-to-load forwarding). Solution efficace, mais qui demande d'annuler les effets de la lecture si l'écriture n'a pas lieu, en cas de mauvaise prédiction de branchement. Pour implémenter cette technique, on construit souvent la file d'écriture sous la forme d'un cache, chaque ligne mémorisant une donnée, et ayant pour tag l'adresse à lire ou écrire. Toute lecture va déclencher automatiquement un accès à ce cache. S'il y a succès de cache, c'est que la lecture a une dépendance avec une écriture de la file d'attente : on renvoie la ligne de cache associée à cette écriture. En cas de défaut de cache, la lecture est effectuée en mémoire RAM ou dans le cache. Mais il faut gérer le cas où plusieurs écritures à la même adresse sont mises en attente dans la file d'écriture. Dans ce cas, la file d'écriture renvoie la dernière donnée écrite, celle fournie par l'écriture la plus récente : cela permet de renvoyer la donnée la plus à jour. En exécutant les lectures dans l'ordre du programme, cela ne pose aucun problème.

Calcul anticipé des adresses

Il est aussi possible de séparer les accès à la mémoire en deux parties, en deux micro-instructions : une pour le calcul d'adresse et l'autre pour les accès mémoire. Cela permet de vérifier à l'avance si l'adresse calculée a des dépendances avec celles des autres instructions. La détection des dépendances est ainsi anticipée de quelques cycles, permettant un accès anticipé sûr des accès mémoire. On parle de **calcul anticipé des dépendances**. Cette technique peut s'implémenter facilement avec la file d'écriture qu'on a vue au-dessus, les instructions pouvant être exécutées directement l'étant, alors que les autres sont mises en attente dans la file d'écriture. Mais on peut aussi utiliser des **matrices de dépendances**, qui permettent de repérer de façon optimale toutes les dépendances entre accès mémoires, sans en laisser passer une seule. Ces matrices forment une espèce de tableau carré, organisé en lignes et en colonnes. Chaque ligne et chaque colonne se voit attribuer une instruction. À l'intersection d'une ligne et d'une colonne, on trouve un bit qui indique si l'instruction de la ligne et celle de la colonne ont une dépendance.

Commençons par voir la version la plus simple de ces matrices de dépendances. Avec celles-ci, on vérifie juste si toutes les adresses des écritures précédentes sont connues ou non. Si elles ne sont pas toutes connues, les lectures vont attendre avant de pouvoir s'exécuter. Dans le cas contraire, on peut alors démarrer nos accès mémoires. La vérification des dépendances (est-ce que deux accès mémoires se font à la même adresse) se fait alors avec une file d'écriture ou dans des circuits spécialisés. Lorsque le processeur démarre une écriture dont il ne connaît pas l'adresse de la donnée à écrire, il va d'abord insérer cette écriture dans ce tableau carré dans une ligne. Cette ligne sera celle d'indice i. Puis, il va mettre tous les bits de la colonne de même indice (i) à 1. Cela permet de dire que notre écriture peut potentiellement avoir une dépendance avec chaque instruction en attente. Vu qu'on ne connaît pas son adresse, on ne peut pas savoir. Lorsque cette adresse est alors connue, les bits de la colonne attribuée à l'écriture sont remis à zéro. Quand tous les bits d'une ligne sont à zéro, la lecture ou écriture correspondante est envoyée vers les circuits chargés de gérer les lectures ou écritures. Ceux-ci se chargeront alors de vérifier les adresses des lectures et écritures, grâce à une file d'écriture et au réacheminement écriture-vers-lecture associé.

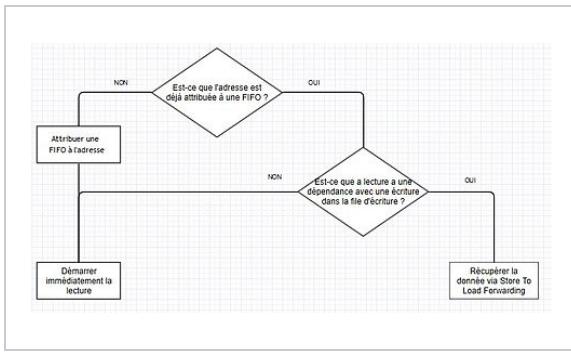
Cette technique peut être améliorée, et gérer la détection des dépendances elle-même, au lieu de les déléguer à une file d'écriture. Dans ce cas, on doit commencer par ajouter l'adresse à laquelle notre instruction va lire ou écrire pour chaque ligne. Puis, à chaque fois qu'une adresse est ajoutée dans une ligne, il suffit de la comparer avec les adresses des autres lignes et mettre à jour les bits de notre matrice en conséquence. Cette technique n'est pas très efficace : il est en effet peu probable que toutes les adresses des écritures précédant une lecture soient connues lorsqu'on veut lire notre donnée. Autant dire que cette technique n'est pas utilisée seule, et elle est augmentée d'autres techniques plus ou moins complémentaires.

Exécution spéculative des accès mémoires

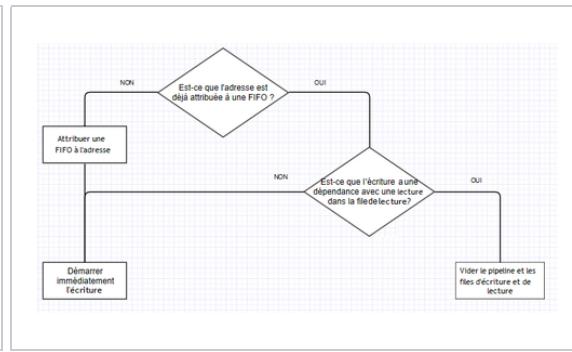
Pour diminuer l'effet des dépendances mémoires, de nombreux processeurs utilisent l'**exécution spéculative** : ils exécutent des instructions de façon anticipée, en supposant certaines choses, et en remettant le processeur à zéro si la supposition se révèle être fausse. La prédiction de branchement est un cas d'exécution spéculative assez connu. Dans le cas des lectures et écritures en mémoire, le processeur suppose qu'il n'y a aucune dépendance mémoire, et démarre les accès mémoires en conséquence. Le processeur vérifie s'il a fait une erreur de prédiction, en vérifiant les adresses à écrire ou lire. En cas d'erreur, il élimine les modifications, comme dans le cas d'une mauvaise prédiction de branchement. Reste que cela peut se faire de différentes manières.

Tampon de résolution d'adresse

Il existe une première méthode pour gérer cette spéculation. Pour la comprendre, il faut se souvenir que les accès mémoires à une même adresse doivent s'effectuer dans l'ordre du programme, mais pas les accès à des adresses différentes. Ainsi, on peut gérer les dépendances en utilisant une FIFO par adresse. Le problème est que le nombre de FIFO peut ne pas suffire, les adresses étant vraiment très nombreuses : il faut bloquer le pipeline si aucune FIFO n'est disponible. Cette méthode est celle qui est utilisée par le **tampon de résolution d'adresse** (address resolution buffer ou ARB). Cette technique, utilise une seule mémoire pour mémoriser toutes les FIFO : la mémoire est divisée en banques, chaque banque correspondant à une FIFO, et chaque mot mémoire correspondant à une lecture ou une écriture mise en attente. Mais ces détails d'implémentation ne sont pas vraiment utiles.



Lecture avec un tampon de résolution d'adresse.



Écriture avec un tampon de résolution d'adresse.

File de lecture

Une autre technique consiste à ajouter des circuits à une file d'écriture, pour la rendre capable d'exécution spéculative. Pour pouvoir fonctionner correctement, notre processeur doit vérifier qu'il n'y a pas d'erreur, ce qui demande de conserver, pour chaque lecture spéculative l'adresse de la lecture effectuée, ainsi que la donnée lue (et éventuellement le program counter). Il faut conserver ces informations jusqu'à ce que toutes les instructions précédant la lecture dans l'ordre du programme soient terminées. Pour cela, on modifie la file de lecture, qui devient un cache : les tags de ce cache mémorisent les adresses, et le reste de chaque ligne de cache s'occupe de la donnée lue.

Pour chaque écriture, il faut vérifier si une lecture qui a accédé à la même adresse se trouve dans la file de lecture. Rien de bien compliqué : on compare l'adresse de l'écriture avec les tags de la file de lecture, qui contiennent l'adresse à laquelle notre lecture a accédé. S'il y a

correspondance (un succès de cache), c'est qu'il y a eu erreur : on doit supprimer du pipeline toutes les instructions exécutées depuis la lecture fautive. Petit détail : sur certains processeurs, la file de lecture et la file d'écriture sont fusionnées dans une seul gros circuit appelé la **file de lecture-écriture** (load-store queue).

Prédiction de dépendances mémoires

Certains processeurs essayent de prédire si deux accès mémoires sont dépendants : ils incorporent une unité qui va fonctionner comme une unité de prédiction de branchement, à la différence qu'elle prédire les dépendances mémoires. Si cette unité prédit que deux accès mémoires sont indépendants, le processeur les exécute dans le désordre, et les exécute dans l'ordre du programme dans le cas contraire. Il faut prendre en compte les erreurs de prédiction, ce qui est fait comme pour les mauvaises prédictions de branchement : on vide le pipeline.

Prédiction agressive

La première méthode consiste à supposer que les lectures n'ont pas de dépendance : une lecture démarre avant même que les écritures qui la précédent dans l'ordre du programme soient connues. Et contraintivement, cela donne de bons résultats. Il faut dire que les situations où une lecture suit de près une écriture sont rares : même pas 2 à 3 % des lectures.

Mémorisation des instructions fautives

On peut améliorer cette technique en mémorisant les instructions qui ont causé une mauvaise prédiction de dépendance mémoire : la prochaine fois qu'on rencontre ceux-ci, on sait qu'il y a de grandes chances qu'il se produise une erreur de prédiction, ce qui pousse à ne pas les exécuter immédiatement. On peut pour cela réutiliser les techniques de prédiction de branchement, tels des compteurs à saturations, mis à jour en cas d'erreurs de prédiction de dépendances mémoire. Dans tous les cas, on trouve un cache, équivalent au branch target buffer, qui mémorise les instructions fautives (leur Program Counter), avec d'autres informations comme les adresses des instructions dépendantes. La première classe de techniques du genre consiste à mémoriser les lectures qui ont causé une violation de dépendance, tandis que l'autre ne mémorise que les écritures. Cette dernière est l'approche utilisée par le **cache de barrières d'écriture** (store barrier cache).

Ensembles d'écritures

Enfin, nous allons conclure avec une dernière technique : celle des **ensembles d'écritures** (store sets). Cette technique est capable de gérer le cas où une lecture dépend de plusieurs écritures : le processeur mémorise l'ensemble des écritures avec lesquelles la lecture a déjà eu une dépendance. Cet ensemble d'écritures associées à une lecture est appelé tout simplement un ensemble d'écritures, et reçoit un identifiant (un nombre) par le processeur. Cette technique demande d'utiliser deux tables :

- une qui assigne un identifiant d'ensemble d'écritures à l'instruction en cours ;
- une autre qui mémorise les ensembles d'écritures sous la forme d'une liste chaînée : le début de la liste correspond à l'écriture la plus ancienne de l'ensemble. Cela permet d'obtenir l'écriture la plus ancienne dans cet ensemble d'écritures directement.

Quand le processeur détecte une violation de dépendance entre une lecture et une écriture, elle ajoute l'écriture dans l'ensemble d'écritures adéquat. Le déroulement d'une lecture demande d'accéder à la première table pour récupérer l'identifiant de l'ensemble d'écritures, et l'utiliser pour adresser la seconde table. S'il y a dépendance, cet accès renvoie l'écriture fautive en cas de dépendance. Quand une écriture est envoyée à l'unité mémoire, celle-ci va accéder à la table de correspondances de la même manière qu'une lecture, et va récupérer l'identifiant de l'ensemble d'écritures auquel elle appartient, identifiant qui est utilisé pour vérifier s'il y a une écriture en cours d'exécution. Si c'est le cas, cela signifie que l'écriture en cours d'exécution doit s'exécuter avant l'écriture qui a consulté la table : cette dernière est mise en attente.

Autres techniques

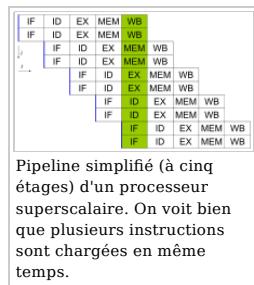
D'autres technique de prédiction des dépendances mémoire ont été inventées et en faire une revue exhaustive serait long et fastidieux : on pourrait citer les ensembles colorés (color sets), et bien d'autres techniques. Quoiqu'il en soit, la recherche sur le sujet est assez riche, comme toujours quand il s'agit d'architecture des ordinateurs.

Processeurs à émissions multiples

Les processeurs vus auparavant ne peuvent émettre au maximum qu'une instruction par cycle d'horloge : ce sont des processeurs à émission unique. Et quand on court après la performance, on en veut toujours plus : un IPC de 1, c'est pas assez ! Pour cela, les concepteurs de processeurs ont inventés des processeurs qui émettent plusieurs instructions par cycle : les processeurs à émissions multiples.

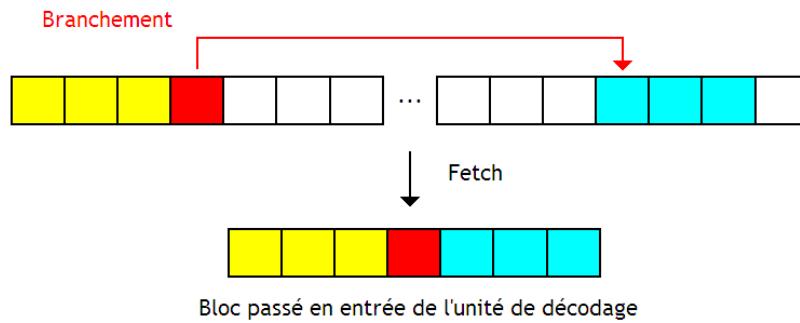
Processeurs superscalaires

Pour que cela fonctionne, il faut répartir les instructions sur différentes unités de calcul, et cela n'est pas une mince affaire. Si le processeur répartit les instructions sur les unités de calcul à l'exécution, on parle de **processeur superscalaire**. Certains processeurs superscalaires n'utilisent pas l'exécution dans le désordre, tandis que d'autres le font. Pour que le processeur répartisse ses instructions sur plusieurs unités de calcul, il faut modifier toutes les étapes entre le chargement et les unités de calcul.

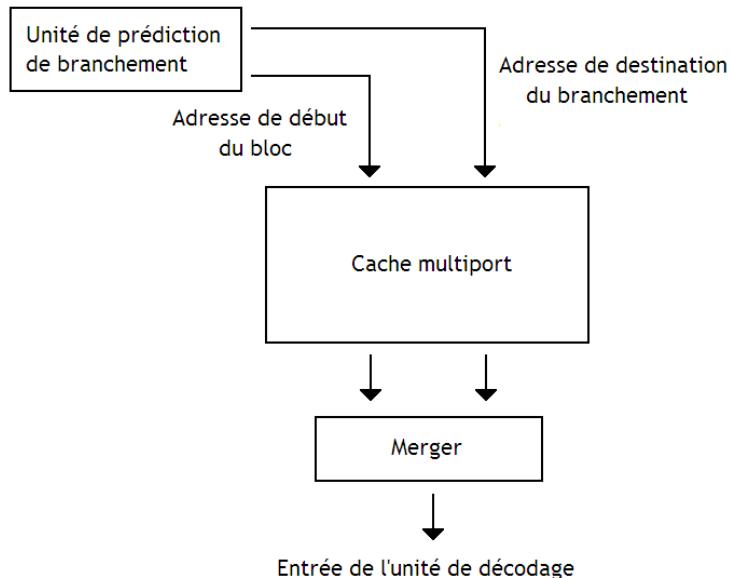


Chargement

Sur les processeurs superscalaires, l'unité de chargement charge un bloc de mémoire de taille fixe, qui contient plusieurs instructions (le program counter est modifié en conséquence). Ceci dit, il faut convenablement gérer le cas où un branchement pris se retrouve en plein milieu d'un bloc. Dans ce qui va suivre, un morceau de code sans branchement est appelé un bloc de base (basic block). Si certains processeurs exécutent toutes les instructions d'un bloc, sauf celles qui suivent un branchement pris, d'autres permettent de charger les instructions de destination du branchement et de les placer à sa suite. Ils vont charger deux blocs à la fois et les fusionner en un seul qui ne contient que les instructions présumées utiles (exécutées). Le principe peut se généraliser avec un nombre de blocs supérieur à deux.



Ces processeurs utilisent des unités de prédiction de branchement capables de prédire plusieurs branchements par cycle, au minimum l'adresse du bloc à charger et la ou les adresses de destination des branchements dans le bloc. De plus, on doit charger deux blocs de mémoire en une fois, via des caches d'instruction multiports. Il faut aussi ajouter un circuit pour assembler plusieurs morceaux de blocs en un seul : le fusionneur (merger). Le résultat en sortie du fusionneur est ce qu'on appelle une **trace**.



Si jamais un bloc est rechargé et que ses branchements sont pris à l'identique, le résultat du fusionneur sera le même. Il est intéressant de conserver cette trace dans un **cache de traces** pour la réutiliser ultérieurement. Pour stocker une trace, rien de plus facile : il suffit de conserver

le résultat obtenu par le fusionneur. Mais il reste encore à déterminer si une trace peut être réutilisée. Une trace est réutilisable quand le premier bloc de base est identique, les branchements sont toujours à la même place dans la trace et les prédictions de branchement restent identiques. Dans ces conditions, le tag du cache de traces doit contenir l'adresse de départ du premier bloc de base, la position des branchements dans la trace et le résultat des prédictions utilisées pour construire la trace. Le résultat des prédictions de branchement utilisées pour construire la trace est stocké sous la forme d'une suite de bits : si la trace contient n branchements, le n -ième bit vaut 1 si ce branchement a été pris, et 0 sinon. Même chose pour la position des branchements dans la trace : le bit numéro n indique si la n -ième instruction de la trace est un branchement : si c'est le cas, il vaut 1, et 0 sinon. Pour savoir si une trace est réutilisable, l'unité de chargement envoie l'adresse de chargement au cache de traces, et le reste des informations étant fournie par l'unité de prédition de branchement. Si le tag est identique, alors on a un succès de cache de traces, et la trace est envoyée directement au décodeur. Sinon, la trace est chargée depuis le cache d'instructions et assemblée.

La présence d'un cache de traces se marie très bien avec une unité de prédition de branchement capable de prédire un grand nombre de branchements par cycle. Malheureusement, ces unités de prédition de branchement sont très complexes et bouffent du circuit. Les concepteurs de processeurs préfèrent utiliser une unité de prédition de branchement normale, qui ne peut prédire l'adresse que d'un seul bloc de base. Pour pouvoir utiliser un cache de traces avec une unité de prédition aussi simple, les concepteurs de processeurs vont ajouter une seconde unité de prédition, spécialisée dans le cache de traces.

Certains caches de traces permettent de stocker plusieurs traces différentes pour une seule adresse de départ, avec une trace par ensemble de prédition. Mais d'autres caches de traces n'autorisent qu'une seule trace par adresse de départ, ce qui diminue la quantité de succès de cache de traces. Sur ces caches, on peut limiter la casse en utilisant des correspondances partielles. Si jamais les prédictions de branchement et la position des branchements n'est pas strictement identique, il arrive quand même que les premières prédictions et les premiers branchements soient les mêmes. Dans ce cas, on peut alors réutiliser les blocs de base concernés : le processeur charge les portions de la trace qui sont valides depuis le cache de traces.

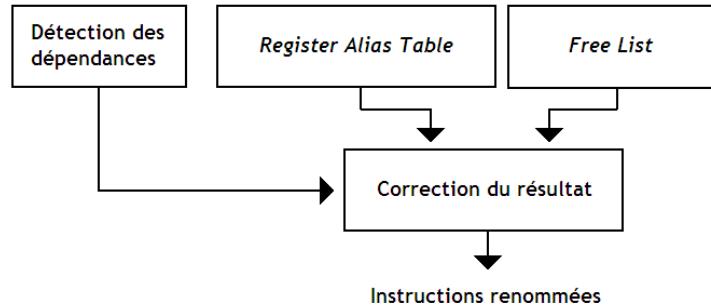
Si les correspondances partielles permettent de limiter la casse, une autre solution est possible : mémoriser les blocs de base dans des caches de blocs de base, et les assembler par un fusionneur pour reconstituer les traces. Par exemple, au lieu d'utiliser un cache de traces dont chaque ligne peut contenir quatre blocs de base, on va utiliser quatre caches de blocs de base. Cela permet de supprimer la redondance que l'on trouve dans les traces, quand elles se partagent des blocs de base identiques, ce qui est avantageux à mémoire égale.

Décodeur d'instruction

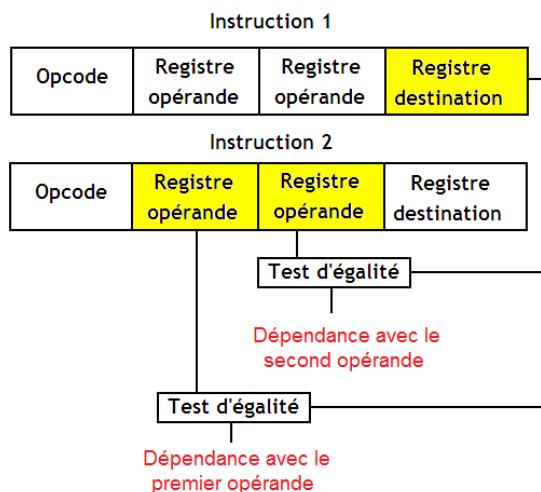
Le séquenceur est lui aussi modifié : celui-ci est maintenant capable de décoder plusieurs instructions à la fois (et il peut aussi éventuellement renommer les registres de ces instructions). Pour ce faire, on peut utiliser un seul décodeur pour décoder plusieurs instructions par cycle, ou plusieurs séquenceurs séparés. Dans certains cas, le séquenceur peut fusionner plusieurs instructions consécutives en une seule micro-instruction. Par exemple, un processeur peut décider de fusionner une instruction de test suivie d'un branchement en une seule micro-opération effectuant les deux en une fois. Cette fusion se fait lors du décodage simultané de plusieurs instructions en même temps.

Unité de renommage

Sur les processeurs superscalaires, l'unité de renommage de registres doit renommer plusieurs instructions à la fois. Mais elle doit aussi gérer le cas où ces instructions ont des dépendances entre elles. Pour faire simple, celle-ci renomme les registres sans tenir compte des dépendances, pour ensuite corriger le résultat après.

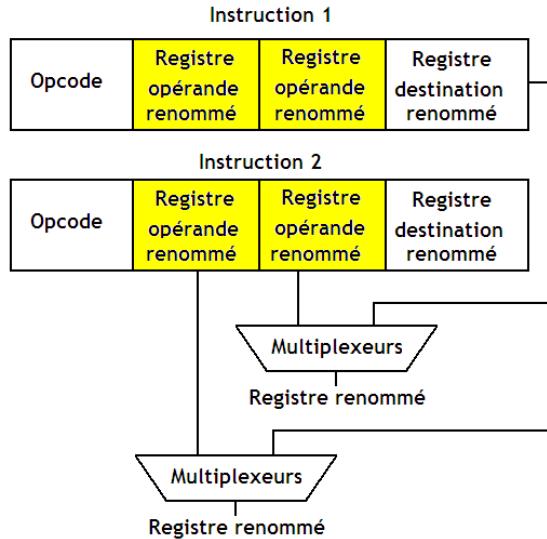


Seules les dépendances lecture-après-écriture doivent être détectées, les autres étant supprimées par le renommage de registres. Repérer ce genre de dépendances se fait assez simplement : il suffit de regarder si un registre de destination d'une instruction est un opérande d'une instruction suivante.



Ensuite, il faut corriger le résultat du renommage en fonction des dépendances. Si une instruction n'a pas de dépendance avec une autre, on la laisse telle quelle. Dans le cas contraire, un registre opérande sera identique avec le registre de destination d'une instruction précédente. Dans ce cas, le registre opérande n'est pas le bon après renommage : on doit le remplacer par le registre de destination de l'instruction avec laquelle il y a dépendance. Cela se fait simplement en utilisant un multiplexeur dont les entrées sont reliées à l'unité de détection des dépendances. On doit faire

ce remplacement pour chaque registre opérande.



Unité d'émission

Sur un processeur à émission multiple, l'unité d'émission doit, en plus de ses fonctions habituelles, détecter les dépendances entre instructions à émettre simultanément. L'unité d'émission d'un processeur superscalaire se voit donc ajouter un nouvel étage pour les dépendances entre instructions à émettre. Sur les processeurs superscalaires à exécution dans l'ordre, il faut aussi gérer l'alignement des instructions dans la fenêtre d'instruction. Dans le cas le plus simple, les instructions sont chargées par blocs et on doit attendre que toutes les instructions du bloc soient émises pour charger un nouveau bloc. Avec la seconde méthode, La fenêtre d'instruction fonctionne comme une fenêtre glissante, qui se déplace de plusieurs crans à chaque cycle d'horloge.

Accès au banc de registres

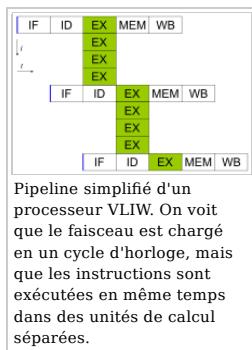
Émettre plusieurs instructions en même temps signifie lire ou écrire plusieurs opérandes à la fois : le nombre de ports du banc de registres doit être augmenté. Mais ces ports ajoutés sont souvent sous-utilisés en situation réelle. On peut en profiter pour ne pas utiliser autant de ports que le pire des cas le demanderait. Pour cela, le processeur doit détecter quand il n'y a pas assez de ports pour servir toutes les instructions : l'unité d'émission devra alors mettre en attente certaines instructions, le temps que les ports se libèrent. Cette détection est réalisée par un circuit d'arbitrage spécialisé, l'arbitre du banc de registres (register file arbiter).

Processseurs VLIW

Les processeurs superscalaires sont tout de même assez complexes et gourmands en circuits : en conséquence, ils chauffent, consomment de l'électricité, etc. On peut améliorer la situation en déléguant le travail de réorganisation des instructions et leur répartition sur les unités de calcul hors du processeur. C'est ainsi que les **processeurs VLW**, ou very long instruction word, sont nés. Ces processeurs sont des processeurs qui n'utilisent pas l'exécution dans le désordre, ce qui fait qu'ils ne peuvent rien contre les dépendances qui ne peuvent être supprimées qu'à l'exécution, comme celles liées aux accès mémoire.

Faisceaux d'instructions

Ces processeurs exécutent des **faisceaux d'instructions** (bundle), des regroupements de plusieurs instructions. Ces instructions peuvent s'exécuter en parallèle sur différentes unités de calcul, mais le faisceau est chargé en une seule fois.



Quand le compilateur regroupe des instructions dans un faisceau, il se peut qu'il ne puisse pas remplir tout le faisceau avec des instructions indépendantes. Sur les anciens processeurs VLIW, les instructions VLIW (les faisceaux) étaient de taille fixe, ce qui forçait le compilateur à remplir d'éventuels vides avec des NOP, diminuant la densité de code. La majorité des processeurs VLIW récents utilisent des faisceaux de longueur variable, supprimant ces NOP.

Chaque instruction d'un faisceau doit expliciter quelle unité de calcul doit la prendre en charge. Et à ce petit jeu, il existe deux possibilités, respectivement nommées **encodage par position** et par **nommage**. Avec la première méthode, la position de l'instruction dans le faisceau détermine l'ALU à utiliser. Un faisceau est découpé en créneaux (slot), chacun étant attribué à une ALU. Avec la seconde solution, chaque instruction d'un faisceau contient un numéro qui indique l'unité de calcul à utiliser. Cette technique est déclinée en deux formes : soit on trouve un identifiant d'ALU par instruction, soit on utilise un identifiant pour tout le faisceau, qui permet à lui seul de déterminer l'unité associée à chaque instruction.

Dépendances intra-faisceaux

Quelques petites difficultés arrivent quand des instructions d'un faisceau manipulent le même registre. Les problèmes apparaissent avec des écritures, notamment quand une instruction lit un registre écrit par une autre instruction. Dans ce cas, le processeur peut gérer la situation de trois manières différentes :

- la lecture du registre renvoie la valeur avant l'écriture ;
- la lecture va lire la valeur écrite par l'écriture ;
- le processeur interdit à deux instructions d'un même faisceau de lire et écrire le même registre.

Exceptions

Que faire lorsqu'une instruction lève une exception dans un faisceau, que se passe-t-il pour les autres instructions ? Là encore, il existe diverses manières de gérer la situation.

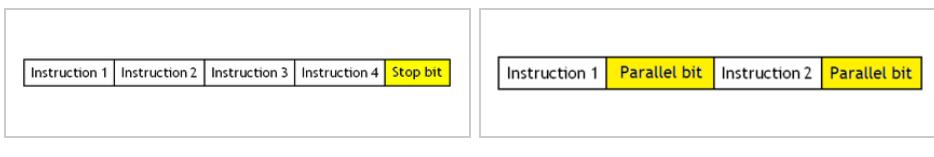
- on invalide toutes les instructions du faisceau ;
- on invalide uniquement les instructions qui suivent l'exception dans l'ordre du programme (le compilateur doit ajouter quelques informations sur l'ordre des instructions dans le faisceau) ;
- on invalide seulement les instructions qui ont une dépendance avec celle qui a levé l'exception ;
- on exécute toutes les instructions du faisceau, sauf celle qui a levé l'exception.

Processeurs EPIC

Pour corriger les problèmes des VLIW, Intel et HP lancèrent un nouveau processeur en 1997 : l'Itanium. Dans un but mercatique évident, Intel et HP prétendirent que l'architecture de ce processeur, bien que ressemblant aux processeurs VLIW, n'était pas du VLIW : ils appellèrent cette nouvelle architecture EPIC, pour explicitly parallel instruction computing.

Faisceaux

La première différence avec les VLIW vient de ce qu'on met dans les faisceaux. Ceux-ci deviennent des groupes d'instructions indépendantes, délimités par un petit groupe de **bits d'arrêt** (stop bits) qui indique la fin d'un faisceau. D'autres processeurs utilisent un **bit de parallélisme** (parallel bit) : un programme est composé d'une suite d'instructions, et chaque instruction termine (ou commence) par un bit qui dit si l'instruction peut s'effectuer en parallèle avec l'instruction encodée.



Bits d'arrêt.

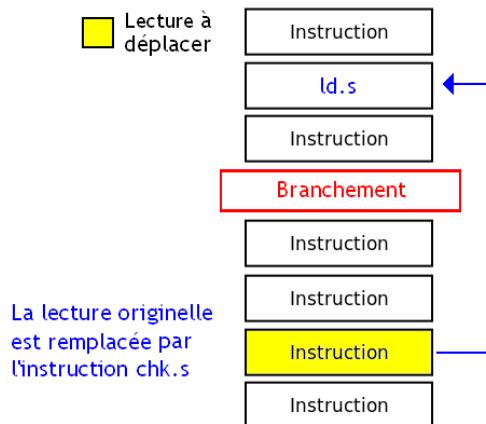
Bit de parallélisme.

Exceptions différées

L'Itanium implémente ce qu'on appelle les **exceptions différées** (delayed exception). Avec cette technique, le compilateur peut créer deux versions d'un même code : une version optimisée qui suppose qu'aucune exception matérielle n'est levée, et une autre version peu optimisée qui suppose qu'une exception est levée. Le programme exécute la version optimisée en premier de manière spéculative, mais annule son exécution et repasse sur la version non-optimisée en cas d'exception. Pour vérifier l'occurrence d'une exception, chaque registre est associé à un bit « rien du tout » (not a thing), mis à 1 s'il contient une valeur invalide. Si une instruction lève une exception, celle-ci écrira un résultat faux dans un registre et le bit « rien du tout » est mis à 1. Les autres instructions propageront ce bit « rien du tout » dans leurs résultats : si un opérande est « rien du tout », le résultat de l'instruction l'est aussi. Une fois le code fini, il suffit d'utiliser une instruction qui teste l'ensemble des bits « rien du tout » et agit en conséquence.

Spéculation sur les lectures

L'Itanium fournit une fonctionnalité similaire pour les lectures, où le code est compilé dans une version optimisée où les lectures sont déplacées sans tenir compte des dépendances, avec un code de secours non-optimisé. Reste ensuite à vérifier que la spéculation était correcte, mais cette vérification ne se fait pas de la même façon selon que la lecture ait été déplacée avant un branchements ou avant une autre écriture. Si on passe une lecture avant un branchements, la lecture et la vérification sont effectuées par les instructions LD.S et CHK.S. Si une dépendance est violée, le processeur lève une exception différée : le bit « rien du tout » du registre contenant la donnée lue est alors mis à 1. CHK.S ne fait rien d'autre que vérifier ce bit. L'Itanium permet de déplacer une lecture avant une écriture, mais délègue la désambiguïsation de la mémoire au compilateur. Tout se passe comme avec les branchements, à part que les instructions sont nommées LD.A et CHK.A.



Pour détecter les violations de dépendance, le processeur maintient une liste des lectures spéculatives qui n'ont pas causé de dépendances mémoire, dans un cache : la table des adresses lues en avance (advanced load address table ou ALAT). Ce cache stocke notamment l'adresse, la longueur de la donnée lue, le registre de destination, etc. Toute écriture vérifie si une lecture à la même adresse est présente dans l'ALAT : si c'est le cas, une dépendance a été violée, et la lecture est retirée de l'ALAT.

Bancs de registres rotatifs

Les processeurs EPIC et VLIW utilisent une forme limitée de renommage de registres pour accélérer certaines boucles. Prenons une boucle simple. Nous montrerons le corps de la boucle, à savoir les instructions de la boucle, sans les branchements et instructions de test. Celle-ci se contente d'ajouter 5 à tous les éléments d'un tableau. L'adresse de l'élément du tableau est stockée dans le registre R2, et est incrémentée automatiquement via le mode d'adressage à pré-incrémentation. Dans le code qui suivra, les crochets serviront à indiquer l'utilisation du mode d'adressage indirect.

loop :

```
load R5 [R2] / add 4 R2 ;
add 5 R5 ;
store [R2] R5 ;
```

La boucle est donc simple. Elle charge l'élément du tableau et calcule l'adresse du prochain élément dans un premier faisceau, le deux pouvant se faire en parallèle. Ensuite, elle ajoute 5 à l'élément. Puis elle l'enregistre. Si on regarde cette boucle du point de vue du pipeline, c'est pas très reluisant : les trois dernières instructions utilisent les mêmes registres et ont donc des dépendances.

On souhaiterait que toutes les instructions soient indépendantes, qu'elles n'utilisent pas les mêmes registres. C'est théoriquement possible : les itérations de la boucle peuvent se recouvrir sans problème, vu qu'elles manipulent des données indépendantes : les éléments du tableau sont manipulés indépendamment. Mais le fait que l'instruction de calcul utilise le même registre que l'instruction de lecture pose quelques problèmes et ajoute une fausse dépendance. Même chose pour le fait que l'écriture utilise le résultat de l'addition, et donc le même registre. Le renommage de registres pourrait les supprimer, mais il n'est pas disponible sur les processeurs VLIW et EPIC.

Pour supprimer ces dépendances, on a inventé les bancs de registres rotatifs (rotating register files). Avec cette technique, un banc de registres est spécialisé dans l'exécution des boucles. De plus, à chaque cycle d'horloge, le nom d'un registre change de manière prévisible. Généralement, le nom d'un registre est augmenté de 1 à chaque cycle d'horloge. Par exemple, le registre nommé R0 à un instant donné devient le registre R1 au cycle d'après. Et c'est la même chose pour tous les registres du banc de registre dédié aux boucles. Évidemment, le code source du programme doit être modifié pour en tenir compte.

Ainsi, le code vu précédemment...

loop :

```
load RB5 [R2] / add 4 R2 ;
add 5 RB6 ;
store [R2] RB7 ;
```

... devient celui-ci.

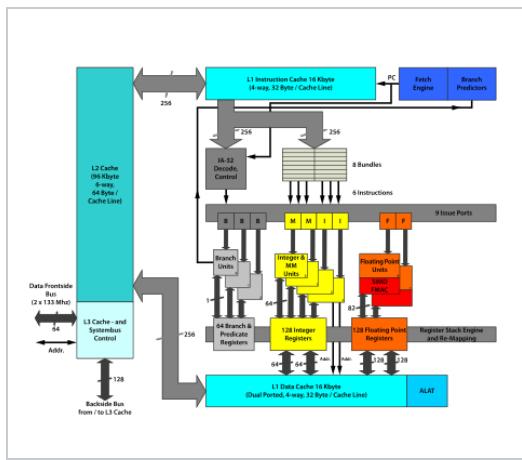
loop :

```
load RB5 [R2] / add 4 R2 ;
add 5 RB6 ;
store [R2] RB7 ;
```

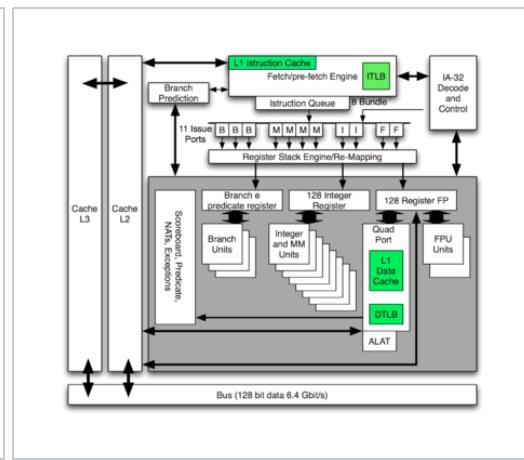
Si on analyse ce qui se passe dans le pipeline, les instructions des différentes itérations deviennent indépendantes. Ainsi, le LOAD d'une itération ne touchera pas le même registre que le LOAD de l'itération suivante. Le nom de registre sera le même, mais le fait que les noms de registre se décalent à chaque cycle d'horloge fera que ces noms identiques ne correspondent pas au même registre. Les dépendances sont supprimées, et le pipeline est utilisé à pleine puissance. Même chose pour toutes les instructions de la boucle : même nom de registre à chaque itération, mais registres physiques différents.

Cette technique s'implémente avec un simple compteur, qui compte de zéro, jusqu'au nombre de registres dédiés aux boucles. Ce compteur est incrémenté à chaque cycle d'horloge. Pour accéder à un registre, le contenu de ce compteur est ajouté au nom de registre à accéder. Évidemment, cette addition a lieu pour chaque port du banc de registre (avec un seul compteur pour tous les ports).

Exemples d'architectures EPIC (Itanium)



Itanium 1.



Itanium 2.

Les architectures découplées

Les processeurs actuels utilisent des optimisations relativement lourdes pour gagner en performances, qui utilisent beaucoup de circuits. Évidemment, les chercheurs ont déjà trouvé quelques alternatives : processeurs VLIW et EPIC, techniques d'exécution dans le désordre, etc. Mais il existe une dernière alternative, relativement mal connue : les architectures découplées. Il existe plusieurs architectures de ce type, que nous allons aborder via quelques exemples.

Découplage des accès mémoire

La première catégorie d'architectures découplées est celle des **architectures découplées accès-exécution**. Ces architectures ont un processeur dédié aux accès mémoire et un autre pour le reste. Le processeur en charge des accès mémoire s'appelle le processeur access, ou A. L'autre processeur, appelé execute, s'occupe des calculs et des branchements. Sur ces architectures, chaque programme est découpé en deux flux d'instructions : un pour les accès mémoire, et l'autre pour le reste. Les deux flux sont exécutés en parallèle, un flux pouvant prendre de l'avance sur l'autre. Ainsi, les lectures peuvent être exécutées en avance. Cette forme de parallélisme est certes plus limitée que celle permise par l'exécution dans le désordre, mais le principe est là. Le découpage du programme en plusieurs flux peut s'effectuer soit à l'exécution, soit à la compilation. Dans le premier cas, on n'utilise qu'un seul et unique programme, qui contient toutes les instructions. Ces processeurs disposent d'une unique unité de chargement, partagée entre les processeurs. Dans le second cas, chaque flux correspondra à un programme séparé. Chaque processeur a un program counter, même s'il y a une synchronisation pour les branchements.

Gestion des accès mémoire

Les accès mémoire sont gérés grâce à diverses mémoires FIFO, qui servent d'interface entre les processeurs et la mémoire. Prenons le cas d'une lecture : la lecture est démarrée par le processeur access, alors que la donnée est destinée au processeur execute. Pour synchroniser les deux processeurs, on trouve une mémoire tampon pour les données lues : la file de lecture de X (X load queue, abrégée XLQ). Quand le processeur X a besoin d'une donnée lue depuis la mémoire, il vérifie si elle est présente dans l'XLQ et se met en attente tant que ce n'est pas le cas.

Le cas des écritures est plus compliqué. En effet, l'adresse de l'écriture est calculée par le processeur A, tandis que la donnée à écrire est calculée par le processeur X, et les deux ne sont pas forcément disponibles en même temps. Pour cela, les adresses et les données sont mises en attente dans deux mémoires tampons qui coopèrent entre elles. La première est intégrée dans le processeur A et met en attente les adresses calculées : c'est la file d'écriture des adresses (store address queue, ou SAQ). La seconde met en attente les données à lire, et relie le processeur X au processeur A : c'est la file d'écriture de X (X store queue, ou XSQ). Quand l'adresse et la donnée sont disponibles en même temps, l'écriture est alors envoyée à la mémoire ou au cache.

Échanges d'information entre processeurs

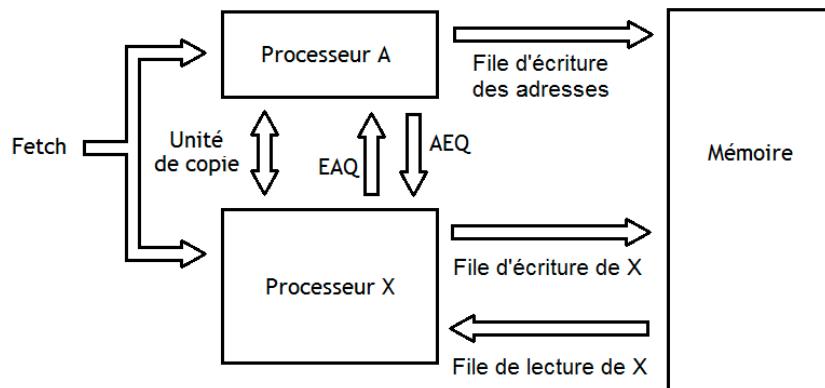
Évidemment, les deux processeurs doivent échanger des informations. Par exemple, une donnée lue par le processeur A peut servir d'opérande pour des instructions sur le processeur X. De même, le calcul d'une adresse sur le processeur A peut utiliser des données calculées par le processeur X. Pour cela, les échanges s'effectuent par copies entre registres : un registre peut être copié du processeur A vers le processeur X, et réciproquement. Pour utiliser cette unité de copie, chaque processeur dispose d'une instruction COPY.

Branchements

Ces deux flux ont exactement la même organisation générale : les branchements sont présents à des endroits identiques dans les deux flux, les structures de contrôle présentes dans un programme sont présentes à l'identique dans l'autre, etc. Or, un branchement pris par un processeur doit être pris sur l'autre : le résultat d'un branchement sur un processeur doit être transmis à l'autre processeur. Pour cela, on trouve deux files d'attente :

- la file A vers X (access-execute queue, abrégée AEQ), pour les transferts du processeur A vers le processeur X ;
- la file X vers A (execute-access queue, abrégée EAQ), pour l'autre sens.

Il faut noter que le processeur peut se trouver définitivement bloqué dans une situation bien précise : si l'AEQ et l'EAQ sont toutes deux totalement remplies ou totalement vides, ce genre de situation peut survenir. Pour éviter cette situation, les compilateurs doivent compiler le code source d'une certaine manière.

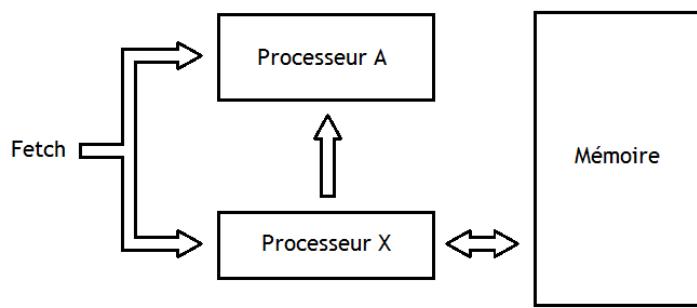


Découplage des branchements

L'idée qui consiste à séparer un programme en deux flux a été adaptée sous d'autres formes.

Découplage simple

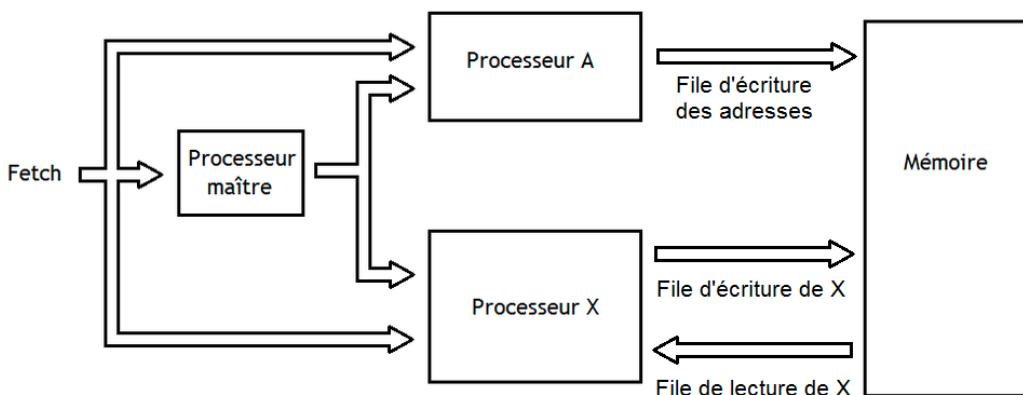
Par exemple, on peut avoir un flux juste pour les branchements, à savoir calculer leurs adresses et les exécuter. Dans ce cas, on a toujours deux processeurs : un pour les branchements et un autre. L'architecture est fortement simplifiée, une seule mémoire FIFO étant nécessaire : celle-ci permet au processeur généraliste d'envoyer des données au processeur qui gère les branchements.



Découplage total du contrôle

Il est possible d'aller encore plus loin : l'article « The effectiveness of decoupling » donne une architecture qui découpe accès mémoire, branchements et calculs. Cette architecture a été implémentée en matériel quelques années après la publication de l'article. On a donc trois processeurs :

- un processeur maître pour les branchements, qui répartit le travail à effectuer sur les deux autres processeurs ;
- un processeur pour la mémoire ;
- un processeur pour les calculs.



Les deux derniers processeurs ne peuvent qu'exécuter des boucles assez spéciales : ces boucles ne doivent pas contenir de branchements ou de boucles imbriquées à l'intérieur. Vu que le processeur maître gère les autres branchements, les files d'attente dédiées aux branchements disparaissent. Le processeur maître gère la répartition des boucles sur les deux autres processeurs. Pour demander l'exécution d'une boucle sur processeur, le processeur maître envoie ce qu'on appelle un bloc de chargement d'instructions (instruction fetch block ou IFB). Cet IFB contient un pointeur sur la première instruction de la boucle, le nombre d'instructions de la boucle et le nombre d'itérations. L'envoi d'un IFB sur les mémoires tampons est réalisé sur ordre d'une instruction machine spécialisée.

Autres

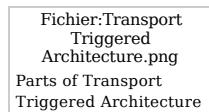
Si les accès mémoire et les branchements peuvent être découplés, la gestion du cache de données ou d'instructions peut aussi l'être. Comme exemple, on peut citer le hierarchical decoupled instruction stream computer : sur cette architecture, le cache n'est pas géré automatiquement par le matériel, mais l'est par des instructions machine. Pour indiquer quelles sont les données préchargées, le processeur de gestion du cache envoie des informations au processeur de gestion des accès mémoire, via une file d'attente.

Les architectures actionnées par déplacement

Sur certains processeurs, les instructions machine sont très simples et correspondent directement à des micro-instructions qui agissent sur le bus. En clair, toutes les instructions machine permettent de configurer directement le bus interne au processeur, et il n'y a pas de séquenceur ! De tels processeurs sont ce qu'on appelle des architectures actionnées par déplacement (transport triggered architectures). Sur une architecture actionnée par déplacement, toutes les instructions (sauf quelques branchements) vont configurer le bus interne du processeur, en reliant ou non les ALU aux registres, en reliant les registres entre eux, etc. Chaque configuration va :

- soit relier une unité de calcul au banc de registres ;
- soit échanger deux données entre registres du banc de registres (connecter un port de lecture du banc de registres sur un port d'écriture) ;
- soit effectuer un branchements.

Certaines de ces architectures utilisent plusieurs bus internes, afin de pouvoir faire plusieurs micro-instructions à la fois.



Implémentation

On peut implémenter ces architectures de deux manières :

- soit en nommant les ports des unités de calcul ;
- soit en intercalant des registres en entrée et sortie des unités de calcul.

Avec des ports

Dans le premier cas, le banc de registres peut être directement connecté sur les entrées et sorties des ALU, que ce soit en entrée ou en sortie. Chaque entrée ou sortie est ce qu'on appelle un port (d'entrée ou de sortie). Mais avec cette organisation, les ports des ALU doivent être sélectionnables : on doit pouvoir dire au processeur que l'on veut connecter tel registre à tel port, tel autre registre à un tel autre port, etc. Ainsi, les ports sont identifiés par une suite de bits, de la même manière que les registres sont nommés avec un nom de registre : chaque port reçoit un nom de port.

Il existe un port qui permet de déclencher le calcul d'une opération : quand on connecte celui-ci sur un des bus internes, l'opération démarre alors. Toute connexion des autres ports d'entrée ou de sortie de l'ALU sur le banc de registres ne déclenche pas l'opération : l'ALU se comporte comme si elle devait faire un NOP et n'en tient pas compte.

Avec des registres

Dans le second cas, on intercale des registres intermédiaires spécialisés en entrée et sortie de l'ALU : le but de ces registres est de stocker les opérandes et le résultat d'une instruction. Tout ce que peut faire le processeur, c'est relier :

- transférer une donnée du banc de registres vers ces registres ;
- transférer des données entre banc de registres et mémoire ;
- éventuellement, faire des échanges de données entre registres généraux.

Ces registres servent spécialement à stocker les opérandes d'une instruction machine, et ne permettent pas de déclencher des opérations : on peut écrire dedans sans que l'ALU ne fasse rien. Par contre, certains registres servent à déclencher des instructions : lorsqu'on écrit une donnée dans ceux-ci, cela va automatiquement déclencher l'exécution d'une instruction bien précise par l'unité de calcul.

Par exemple, un processeur de ce type peut contenir trois registres « ajout.opérande.1 », « ajout.déclenchement » et « ajout.résultat ». Le premier registre servira à stocker le premier opérande de l'addition. Pour déclencher l'opération d'addition, il suffira d'écrire le second opérande dans le registre « ajout.déclenchement », et l'instruction s'exécutera automatiquement. Une fois l'instruction terminée, le résultat de l'addition sera automatiquement écrit dans le registre « ajout.résultat ». Il existera des registres similaires pour la multiplication, la soustraction, les comparaisons, etc.

Utilité

L'utilité de ces architectures n'est pas évidente. Leur raison d'exister est simplement la performance : manipuler le bus interne du processeur directement au lieu de laisser faire les circuits du processeur permet de faire pas mal de raccourcis et permet quelques petites optimisations. Par exemple, on peut envoyer le résultat fourni par une unité de calcul directement en entrée d'une autre, sans avoir à écrire ce résultat dans un registre intermédiaire du banc de registres. Mais l'intérêt est assez faible.

Évidemment, les opérations en elles-mêmes prennent toujours un peu de temps. Et ces temps doivent être connus pour obtenir un programme correct : sans cela, on pourrait par exemple lire une donnée avant que celle-ci ne soit disponible. Cela demande donc des compilateurs dédiés, qui connaissent les temps de latence de chaque instruction.

Sur certains de ces processeurs, on n'a besoin que d'une seule instruction qui permet de copier une donnée d'un emplacement (registre ou adresse mémoire) à un autre. Pas d'instructions LOAD, STORE, etc. : on fusionne tout en une seule instruction supportant un grand nombre de modes d'adresses. Et donc, on peut se passer complètement d'opcode, vu qu'il n'y a qu'une seule instruction : pas besoin de préciser quelle est celle-ci, on le sait déjà. Sympa, non ?

Les processeurs de traitement du signal

Les DSP, les processeurs de traitement du signal, sont des jeux d'instructions spécialement conçus pour travailler sur du son, de la vidéo, des images...

Instructions

Le jeu d'instruction d'un DSP est assez spécial, que ce soit pour le nombre de registres, leur utilisation, ou la présence d'instructions insolites.

Registres

Pour des raisons de couts, tous les DSP utilisent un faible nombre de registres spécialisés. Un DSP a souvent des registres entiers séparés des registres flottants, ainsi que des registres spécialisés pour les adresses mémoires. On peut aussi trouver des registres spécialisés pour les indices de tableau ou les compteurs de boucle. Cette spécialisation des registres pose de nombreux problèmes pour les compilateurs, qui peuvent donner lieu à une génération de code sous-optimale. De nombreuses applications de traitement du signal ayant besoin d'une grande précision, les DSP sont dotés de registres accumulateurs très grands, capables de retenir des résultats de calcul intermédiaires sans perte de précision. De plus, certaines instructions et certains modes d'adressage ne sont utilisables que sur certains types de registres. Certaines instructions d'accès mémoire peuvent prendre comme destination ou comme opérande un nombre limité de registres, les autres leur étant interdits. Cela permet de diminuer le nombre de bits nécessaire pour encoder l'instruction en binaire.

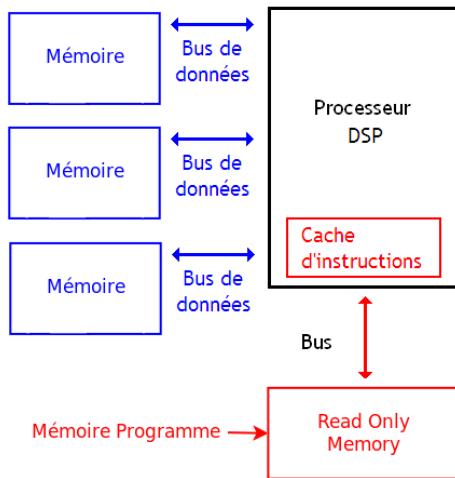
Instructions courantes

Les DSP utilisent souvent l'arithmétique saturée. Certains permettent d'activer et de désactiver l'arithmétique saturée, en modifiant un registre de configuration du processeur. D'autres fournissent chaque instruction de calcul en double : une en arithmétique modulaire, l'autre en arithmétique saturée. Les DSP fournissent l'instruction multiply and accumulate (MAC) ou fused multiply and accumulate (FMAC), qui effectuent une multiplication et une addition en un seul cycle d'horloge, ce calcul étant très courant dans les algorithmes de traitement de signal. Il n'est pas rare que l'instruction MAC soit pipelinée : il suffit d'utiliser un multiplicateur, un additionneur et d'insérer un registre entre l'additionneur et le multiplicateur.

Pour plus d'efficacité dans certaines applications numériques, les DSP peuvent gérer des flottants spéciaux : les **nombres flottants par blocs**. Dans certains cas, il arrive que les nombres flottants à manipuler aient tous le même exposant. Les nombres flottants par blocs permettent alors de mutualiser l'exposant. Par exemple, si je dois encoder huit nombres flottants dans ce format, je devrai utiliser un mot mémoire pour l'exposant, et huit autres pour les mantisses et les signes.

Pour accélérer les boucles for, les DSP ont des instructions qui effectuent un test, un branchemet, et une mise à jour de l'indice en un cycle d'horloge. Cet indice est placé dans des registres uniquement dédiés aux compteurs de boucles. Autre fonctionnalité : les instructions autorépétées, des instructions qui se répètent automatiquement tant qu'une certaine condition n'est pas remplie. L'instruction effectue le test, le branchemet, et l'exécution de l'instruction proprement dite en un cycle d'horloge. Cela permet de gérer des boucles dont le corps se limite à une seule instruction. Cette fonctionnalité a parfois été améliorée en permettant d'effectuer cette répétition sur des suites d'instructions.

Les DSP sont capables d'effectuer plusieurs accès mémoires simultanés par cycle, en parallèle. Certains permettent ainsi aux instructions arithmétiques et logiques de charger tous leurs opérandes depuis la mémoire en même temps, et éventuellement d'écrire le résultat en mémoire lors du même cycle. Dans d'autres cas, les instructions d'accès mémoires sont séparées des instructions arithmétiques et logiques, mais peuvent quand même effectuer plusieurs accès mémoire par cycles : ce sont des déplacements parallèles (parallel moves). Nul doute que dans les deux cas, le processeur doit être conçu pour. Tout d'abord, le processeur doit charger une instruction en même temps que ses opérandes : cela nécessite une architecture Harvard. Ensuite, il faut faire en sorte que la mémoire soit multiport pour gérer plusieurs accès par cycle. Un DSP ne possède généralement pas de cache pour les données, mais conserve parfois un cache d'instructions pour accélérer l'exécution des boucles.



Modes d'adressage

Les DSP incorporent pas mal de modes d'adresses spécialisés.

Indirect à registre avec post- ou préincrément/décrément

Pour accélérer le parcours de tableaux, les DSP incorporent des modes d'adresses spécialisés, comme l'adressage postincrémenté (décrémenté). Avec ce mode d'adressage, l'adresse de l'élément du tableau est stockée dans un registre. À chaque fois qu'une instruction accède à ce registre, son contenu est incrémenté ou décrémenté pour pointer sur le prochain élément. Il existe une variante où l'incrémentation ou la décrémentation s'effectuent avant l'utilisation effective de l'adresse.

Adressage « modulo »

Les DSP implémentent des modes d'adresses servant à faciliter l'utilisation de files, des zones de mémoire où l'on stocke des données dans un certain ordre. On peut y ajouter de nouvelles données, et en retirer, mais les retraits et ajouts ne peuvent pas se faire n'importe comment : quand on retire une donnée, c'est la donnée la plus ancienne qui quitte la file. Tout se passe comme si ces données étaient rangées dans l'ordre en mémoire.

Ces files sont implémentées avec un tableau, auquel on ajoute deux adresses mémoires : une pour le début de la file et l'autre pour la fin. Le début de la file correspond à l'endroit où l'on insère les nouvelles données. La fin de la file correspond à la donnée la plus ancienne en mémoire. À chaque ajout de donnée, on doit mettre à jour l'adresse de début de file. Lors d'une suppression, c'est l'adresse de fin de file qui doit être mise à jour.

Ce tableau a une taille fixe. Si jamais celui-ci se remplit jusqu'à la dernière case, (ici la cinquième), il se peut malgré tout qu'il reste de la place au début du tableau : des retraits de données ont libéré de la place. L'insertion continue alors au tout début du tableau. Cela demande de vérifier si l'on a atteint la fin du tableau à chaque insertion. De plus, en cas de débordement, si l'on arrive à la fin du tableau, l'adresse de la donnée la plus récemment ajoutée doit être remise à la bonne valeur : celle pointant sur le début du tableau. Tout cela fait pas mal de travail.

Le mode d'adressage « modulo » a été inventé pour faciliter la gestion des débordements. Avec ce mode d'adressage, l'incrémentation de l'adresse au retrait ou à l'ajout est donc effectué automatiquement. De plus, ce mode d'adressage vérifie automatiquement que l'adresse ne déborde pas du tableau. Et enfin, si cette adresse déborde, elle est mise à jour pour pointer au début du tableau.

Suivant le DSP, ce mode d'adressage est géré plus ou moins différemment. La première méthode utilise des registres « modulo », qui stockent la taille du tableau. Chaque registre est associé à un registre d'adresse pour l'adresse/indice de l'élément en cours. Vu que seule la taille du tableau est mémorisée, le processeur ne sait pas quelle est l'adresse de début du tableau, et doit donc ruser. Cette adresse est souvent alignée sur un multiple de 64, 128, ou 256. Cela permet ainsi de déduire l'adresse de début de la file : c'est le multiple de 64, 128, 256 strictement inférieur le plus proche de l'adresse manipulée.

Autre solution : utiliser deux registres, un pour stocker l'adresse de début du tableau et un autre pour sa longueur. Et enfin, dernière solution, utiliser un registre pour stocker l'adresse de début, et un autre pour l'adresse de fin.

Adressage à bits inversés

L'adressage à bits inversés (bit-reverse) a été inventé pour accélérer les algorithmes de calcul de transformée de Fourier (un « calcul » très courant en traitement du signal). Cet algorithme va prendre des données dans un tableau, et va fournir des résultats dans un autre tableau. Seul problème, l'ordre d'arrivée des résultats dans le tableau d'arrivée est assez spécial. Par exemple, pour un tableau de 8 cases, les données arrivent dans cet ordre : 0, 4, 2, 6, 1, 5, 3, 7. L'ordre semble être totalement aléatoire. Mais il n'en est rien : regardons ces nombres une fois écrits en binaire, et comparons-les à l'ordre normal : 0, 1, 2, 3, 4, 5, 6, 7.

Ordre normal	Ordre Fourier
000	000
001	100
010	010
011	110
100	001
101	101
110	011
111	111

Comme vous le voyez, les bits de l'adresse Fourier sont inversés comparés aux bits de l'adresse normale. Nos DSP disposent donc d'un mode d'adressage qui inverse tout ou partie des bits d'une adresse mémoire, afin de gérer plus facilement les algorithmes de calcul de transformées de Fourier. Une autre technique consiste à calculer nos adresses différemment. Il suffit, lorsqu'on ajoute un indice à notre adresse, de renverser la direction de propagation de la retenue lors de l'exécution de l'addition. Certains DSP disposent d'instructions pour faire ce genre de calculs.

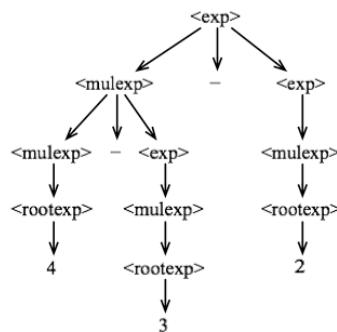
Les architectures pour langages fonctionnels

Si vous programmez depuis un certain temps, vous savez sûrement qu'il existe plusieurs paradigmes de programmation : le paradigme procédural, l'impératif, l'objet, le fonctionnel, etc. Au départ, les premiers ordinateurs étaient optimisés pour exécuter des langages de programmation impératifs très simples. Puis l'assembleur de ces processeurs a évolué pour supporter les fonctionnalités des langages de haut niveau : le nombre de types supportés par le processeur s'est étendu, des instructions spéciales pour supporter les sous-programmes ont été inventées, et de nombreux modes d'adressage dédiés aux tableaux et structures ont fait leur apparition. Toutes ces améliorations ont fait que nos processeurs sont particulièrement adaptés à des langages procéduraux de haut niveau comme le C, le Pascal, le BASIC ou le Fortran. Quant la programmation orientée objet s'est démocratisée dans les années 1980, des processeurs prenant en charge la programmation orienté objet ont été inventés (l'Intel iAPX 432, par exemple). De même, il existe de nombreuses architectures spécialement adaptées à l'exécution de langages fonctionnels ou logiques. Ces architectures étaient assez étudiées dans les années 1980, mais sont aujourd'hui tombées en désuétude.

Les premières tentatives se contentaient de quelques fonctionnalités architecturales qui accéléraient l'exécution des langages fonctionnels, tels des architectures à tags pour gérer le typage dynamique en matériel, des ramasse-miettes matériels ou du filtrage par motif en matériel. Dans le genre, on peut citer les **machines Lisp**. L'étape suivante allait un peu plus loin que simplement accélérer certaines fonctionnalités des langages fonctionnels. Comme vous le savez sûrement, les langages de programmation sont souvent compilés vers un langage intermédiaire, avant d'être transformés en langage machine. Ce langage intermédiaire peut être vu comme l'assembleur d'une machine abstraite, que l'on appelle une machine virtuelle. Les concepteurs de langages fonctionnels apprécient notamment la **machine SECD**. Celle-ci a été implémentée en matériel par plusieurs équipes, par l'intermédiaire de micro-code. La première implémentation a été créée par les chercheurs de l'université de Calgary, en 1989. D'autres architectures ont effectué la même chose, en rajoutant des instructions notamment. D'autres processeurs implémentent des machines virtuelles différentes, comme les combinatoires SKIM, ou toute autre machine virtuelle pour langages fonctionnels. Ce n'est que par la suite que de véritables architectures spécialisées pour les langages fonctionnels ont vu le jour, les architectures par réduction de graphe et les architectures dataflow en étant les deux exemples cardinaux.

Architectures à réduction

Poursuivre ce chapitre nous demandera quelques rappels sur les langages fonctionnels. Pour rappel, la syntaxe d'un langage de programmation décrit toujours un certain nombre de primitives : des constantes, des variables, des opérateurs, des fonctions, etc. Ces primitives peuvent ensuite être combinées ensemble pour calculer un résultat, une valeur : l'ensemble forme alors une expression. Un programme écrit dans un langage fonctionnel peut être vu comme une grosse expression, elle-même composée de sous-expressions, et ainsi de suite. Exécuter un programme écrit dans un langage fonctionnel, c'est simplement l'évaluer. Il existe diverses méthodes pour évaluer une expression, lui donner sa valeur. La première démarre l'évaluation quand toutes les valeurs des primitives sont connues : on parle d'évaluation stricte. L'autre méthode calcule l'expression au fil de l'eau, au fur et à mesure que ses primitives renvoient une valeur : on parle d'évaluation non stricte. Les architectures matérielles qui vont suivre sont capables d'évaluer un programme fonctionnel en utilisant une stratégie non stricte.



Architectures à réécriture

Les **architectures à réécriture de chaîne de caractère** stockent le programme sous la forme d'une chaîne de caractères qui est éditée au fur et à mesure de l'exécution du programme. Au fur et à mesure que les expressions sont calculées, la chaîne de caractères qui leur correspond est réécrite : elle est remplacée par la valeur évaluée. Ces architectures sont des architectures à réécriture. Comme architectures qui fonctionnent sur ce principe, on trouve :

- la Magoo's FFP machine ;
- la GMD reduction machine ;
- la Newcastle reduction machine ;
- la cellular tree machine.

Architectures à réduction de graphes

Sur les **architectures à réduction de graphe**, l'expression est représentée sous la forme d'un graphe dont les nœuds sont des constantes, variables, opérateurs ou fonctions. Évaluer une expression, c'est simplement remplacer les nœuds correspondant aux opérateurs, par les valeurs calculées. Le graphe est ainsi progressivement réduit, jusqu'à ce qu'il n'y ait plus qu'un seul nœud : le résultat. Les architectures à réduction de graphe sont conçues pour effectuer automatiquement cette réduction.

Chaque noeud d'un graphe est stocké sous la forme d'une structure, dont le format dépend de l'ordinateur. Réduire un noeud demande de créer un nouveau noeud pour le résultat de la réduction, et de mettre à jour les pointeurs vers le noeud initial (pour qu'il pointe vers le résultat et non l'opérateur). La conséquence, c'est que les architectures à réduction de graphe doivent allouer elles-mêmes les nœuds et doivent mettre à jour les pointeurs automatiquement. Ces architectures effectuent aussi le ramasse-miettes en matériel.

Diverses architectures spécialisées dans la réduction de graphes ont été implémentées. Certaines de ces architectures étaient des processeurs normaux auxquels des instructions de réduction de graphe étaient ajoutées via microcode. D'autres sont des machines multiprocesseurs. Plus récente, on pourrait citer le Reducer, une architecture monoprocesseur qui exécute du code machine fonctionnel sans microcode. Voici une liste non exhaustive de ces architectures à réduction de graphe :

- la machine ALICE ;
- la machine GRIP ;
- la machine de COBWEB ;
- la machine de COBWEB-2 ;
- la machine Rediflow ;
- la SKIM machine ;
- la machine PACE ;
- l'architecture AMPS ;

- l'APERM machine ;
- la machine MaRS-Lisp ;
- le Reduceron ;
- la machine NORMA.

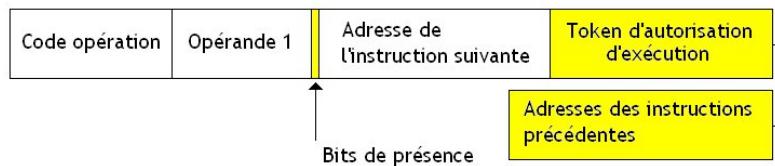
Architectures dataflow

Dans les années 1970 à 1980, les chercheurs tentaient de créer des architectures parallèles, pouvant se passer de program counter. Les architectures dataflow furent une de ces réponses. Sur ces architectures, une instruction s'exécute dès que ses opérandes sont disponibles, comme sur les architectures à exécution dans le désordre. Sauf que cette fois-ci, c'est directement le jeu d'instruction qui permet ce genre de prouesses. Supprimer les dépendances de données est alors le rôle du compilateur. Pour supprimer ces dépendances, les compilateurs ajoutent une contrainte simple : impossible de modifier la valeur d'une donnée/variable tant que celle-ci peut encore être utilisée. Or, les langages fonctionnels permettent d'initialiser une variable lors de sa création, mais pas de la modifier ! Il va de soi qu'un programme sur une architecture dataflow doit préciser les dépendances de données entre instructions, vu que le program counter a disparu. Pour cela, chaque instruction indique l'adresse mémoire de toute instruction qui a besoin de son résultat. Vu que chaque instruction ne s'exécute que si ses opérandes sont disponibles — ont été calculés — détecter la disponibilité des opérandes des instructions est alors primordial.

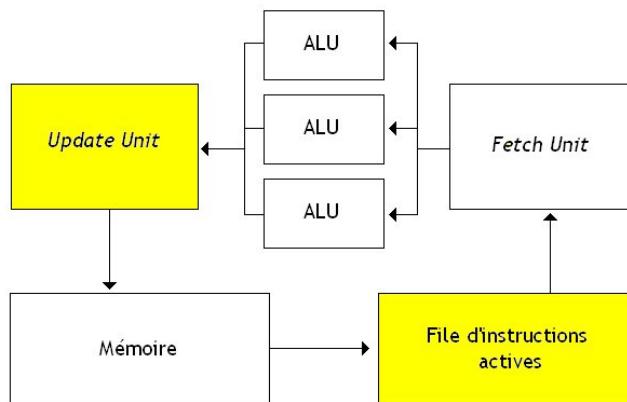
Architectures statiques

La détection de la disponibilité des opérandes est primordiale sur les architectures dataflow. La première manière de faire est celle utilisée sur les **architectures dataflow statiques**. Sur ces architectures, on doit se débrouiller pour n'avoir qu'une seule valeur pour chaque opérande potentiel. Une instruction contient alors ses opérandes (ou leur adresse), son opcode et quelques bits de présence qui indiquent si l'opérande associé est disponible. Tout cela ressemble aux entrées des stations de réservation sur les processeurs à exécution dans le désordre, à un détail près : tout est stocké en mémoire, dans la suite de bits qui code l'instruction.

Cependant, l'absence de program counter fait que certaines instructions peuvent fournir leurs résultats dans le désordre, notamment avec des boucles. Prenons le cas de deux instructions A et B , A calculant un opérande de B : rien n'empêche A d'être exécuté deux fois de suite alors que l'instruction B n'a pas encore consommé le premier résultat : dans ces conditions, le résultat de l'instruction A ne doit pas être recopié dans le champ opérande. Pour empêcher cela, chaque instruction contient un bit, un jeton d'autorisation qui indique si elle peut calculer un nouveau résultat. Quand une instruction s'exécute, elle prévient l'instruction qui a calculé l'opérande qu'elle est prête à accepter un nouvel opérande : le jeton de celle-ci est mis à jour. Cela signifie que chaque instruction doit aussi savoir quelle est la ou les instructions qui la précédent dans l'ordre des dépendances. Pour cela, une seule solution : stocker toutes leurs adresses dans l'instruction. Et cela est à l'origine d'un défaut assez problématique : les fonctions réentrant et récursives ne sont pas implantables, et les boucles dont les itérations sont indépendantes ne peuvent pas être parallélisées. En effet, lorsqu'on exécute une fonction réentrant ou plusieurs itérations d'une boucle, plusieurs versions des mêmes instructions peuvent s'exécuter en même temps : une version de la fonction par appel récursif ou par itération. Et chaque version va vouloir écrire dans les champs opérandes et mettre à jour les bits de présence. Heureusement, les jetons d'autorisation vont éviter tout problème, en ne laissant qu'une seule version (la plus ancienne) s'exécuter, en faisant attendre les autres.

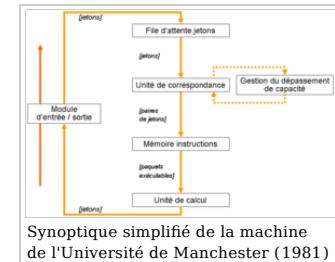
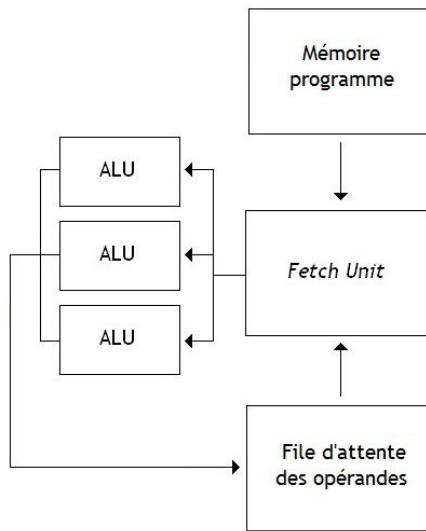


Dans les grandes lignes, une architecture dataflow statique est composée de plusieurs éléments, représentés sur le schéma que vous pouvez voir en-dessous. La file d'instructions est une mémoire qui met en attente les instructions dont toutes les données sont prêtes : elle contient leurs adresses, qu'elle peut fournir à l'unité de chargement. L'unité de mise à jour enregistre les résultats des instructions en mémoire et met à jour les bits de présence. Elle va aussi vérifier que l'instruction de destination du résultat a tous ses opérandes disponibles, et charge celle-ci dans la file d'instructions si c'est le cas.



Architectures dynamiques

La solution est venue avec les **architectures dataflow dynamiques**. Avec elles, les fonctions récursives et réentrant sont supportées, et les boucles sont automatiquement déroulées par le processeur à l'exécution. Pour cela, on ajoute à chaque opérande des informations qui permettent de préciser à quel exemplaire d'une fonction ou d'une boucle ils sont destinés. Ces informations sont stockées dans l'opérande, et sont regroupées dans ce qu'on appelle le tag. Dorénavant, une instruction s'exécute une fois que tous ses opérandes ayant le même tag sont disponibles. Cela a une grosse conséquence : on ne peut coder les instructions comme sur les architectures statiques. Vu qu'une instruction dans une boucle ou une fonction peut s'exécuter en un nombre indéterminé d'exemplaires, on ne peut réservé de la place pour les opérandes à l'avance. C'est maintenant aux opérandes d'indiquer quelle est l'instruction qui doit les manipuler : chaque opérande indique l'adresse de l'instruction à laquelle il est destiné. Évidemment, les opérandes sont mémorisés à part des instructions, dans une mémoire séparée de la mémoire d'instructions (en clair, les architectures dynamiques sont de type Harvard). Les opérandes sont mis en attente dans une file d'attente. À chaque ajout d'un opérande dans la file d'attente, le processeur vérifie quels sont les opérandes de la file qui possèdent le même tag et la même adresse d'instruction. Ainsi, la file d'attente est une mémoire associative.



Autres

Les **architectures Explicit Token-Store**, qui utilisent une pile : l'opérande est remplacé par un pointeur vers le cadre de pile qui contient la donnée et de quoi localiser l'opérande dans la pile. Elles représentent une amélioration assez conséquente des précédentes. Cependant, même si les architectures dataflow sont certes très belles et élégantes, mais elles ont des performances assez médiocres. En conséquence, elles ont été abandonnées, après pas mal de recherches. Mais certains chercheurs ont quand même décidé de donner une dernière chance au paradigme dataflow, en le modifiant de façon à le rendre un peu plus impératif.

Ainsi sont nées les **architectures Threaded Dataflow** ! Sur ces architectures, un programme est découpé en plusieurs blocs d'instructions. À l'intérieur de ces blocs, les instructions s'exécutent les unes à la suite des autres. Par contre, les blocs eux-mêmes seront exécutés dans l'ordre des dépendances de données : un bloc commence son exécution quand tous les opérandes nécessaires à son exécution sont disponibles.

Les architectures **Explicit Data Graph Execution** se basent sur le fonctionnement des compilateurs modernes pour découper un programme procédural ou impératif en blocs de code. Généralement, ceux-ci vont découper un programme en petits morceaux de code qu'on appelle des blocs de base, qui sont souvent délimités par des branchements (ou des destinations de branchements). Ces sont ces blocs de base qui seront exécutés dans l'ordre des dépendances de données. Certaines architectures de ce type ont déjà été inventées, comme les processeurs WaveScalar et TRIPS.

Les architectures à capacités

Certains processeurs incorporent des méthodes qui permettent d'améliorer la sûreté de fonctionnement ou la sécurité. Elles permettent d'éviter certaines attaques logicielles, comme des virus ou des accès non autorisés directement en matériel. Ces jeux d'instructions sont conçus en synergie avec certains systèmes d'exploitation. Il s'agit des capability based processors, ou **architectures à capacités**. Dans les grandes lignes, il s'agit d'architectures orientées objet, dont le langage machine est un langage orienté objet (bien qu'étant un langage machine).

Les capacités : les principes de haut-niveau

Sur les Capability Based processors, la notion même d'adresses mémoire n'existe pas (ou presque). À la place, chaque donnée manipulée par le processeur est stockée dans un **objet**, une sorte de conteneur générique placé quelque part dans la mémoire, sans que l'on ait de moyen de savoir où. Cet objet peut être absolument n'importe quoi : cela peut être un objet spécifié par le programmeur, ou des objets prédefinis lors de la fabrication du processeur. Par exemple, on peut considérer chaque périphérique comme un objet, auquel on a défini des méthodes bien particulières qui permettront de communiquer avec celui-ci ou de le commander. Sur d'autres architectures, chaque programme en cours d'exécution est considéré comme un objet, avec des méthodes permettant d'agir sur son état. On peut ainsi stopper l'exécution d'un programme via des méthodes adaptées, par exemple. Mais ce qui va nous permettre d'adapter des langages de programmation orientés objet sur de telles architectures, c'est la possibilité de créer soi-même des objets non définis lors de la fabrication du processeur.

Au lieu d'utiliser des adresses mémoire et autres mécanismes divers et variés, chaque objet se voit attribuer un **identifiant** bien précis. Cet identifiant est unique à un objet (deux objets ne peuvent avoir le même identifiant), et il ne change pas au cours du temps : il est défini lorsqu'un objet est créé et ne peut être réutilisé pour un autre objet que lorsque l'objet possédant cet identifiant est détruit. Dans les grandes lignes, on peut voir cet identifiant comme une sorte d'adresse virtuelle, qui permet de localiser l'objet mais peut correspondre à une adresse physique totalement différente. De plus, chaque objet est associé à des **autorisations d'accès**. Par exemple, le code d'un système d'exploitation aura accès en écriture à certains objets critiques, qui contiennent des paramètres de sécurité critique, mais les autres programmes n'y auront accès qu'en lecture (voire pas du tout). Les droits d'accès ne seront pas les mêmes et les capacités différentes (parfois pour un même objet). Ces droits d'accès sont rassemblés dans une suite de bits, que l'on appelle une **capacité**. Celles-ci décrivent les droits d'accès, l'identifiant et éventuellement le type de la donnée. Souvent, la partie de l'objet ou de la capability spécifiant le type permet d'identifier certains types prédefinis, mais peut aussi être configurée de façon à utiliser des types définis par le programmeur : sa signification dépendra alors de ce qu'a décidé le programmeur.

Les instructions de lecture et écriture prennent comme argument un identifiant et une capacité. Pour avoir accès à un identifiant, le programme doit fournir automatiquement la capacité qui va avec : il doit la charger dans des registres spécialisés. Ces capacités sont mémorisées dans la mémoire RAM : chaque programme ou fonction a accès à une **liste de capacités** en mémoire RAM. Les instructions qui manipulent les registres de capacités ne peuvent pas, par construction, augmenter les droits d'accès d'une capacité : ils peuvent retirer des droits, pas en ajouter. Ce mécanisme interdit donc à tous sous-programme ou programme de modifier un objet qui n'est pas dans sa liste de capacité : le programme ne pourra pas récupérer la capacité et n'aura donc pas accès à l'objet voulu. Avec ce genre de mécanismes, il devient difficile d'exécuter certains types d'attaques, ce qui est un gage de sûreté de fonctionnement indéniable. Du moins, c'est la théorie : tout repose sur l'intégrité des listes de capacité : si on peut modifier celles-ci, alors il devient facile de pouvoir accéder à des objets auxquels on n'aurait pas eu droit.

Implémentation matérielle des capacités

Divers mécanismes dépendants du processeur permettent d'implémenter l'héritage ou d'autres fonctionnalités objet en autorisant des manipulations, accès, copies ou partages temporaires de listes de capacités. Généralement, ce genre de fonctionnalité objet est géré directement au niveau des instructions du processeur : le processeur contient pour ce faire des instructions spéciales. Ces instructions sont souvent des instructions d'appels de fonction particulières. Pour ceux qui ne le savent pas, une instruction d'appel de fonction sert à demander au processeur d'exécuter une fonction bien précise. Sur les processeurs optimisés pour les langages procéduraux, une fonction est identifiée par son adresse, tandis que ces processeurs fournissent sa capacité à l'instruction chargée d'exécuter notre fonction. Sur ces processeurs, de nombreuses instructions d'appel de fonction sont disponibles : par exemple, l'instruction pour appeler une fonction définie dans la classe de l'objet qu'elle va manipuler ne sera pas la même de celle devant appeler une fonction héritée d'une autre classe : il faudra en effet faire quelques accès pour modifier ou accéder à des listes de capacités extérieures, etc.

En effet, accéder aux données de l'objet demande de connaître son adresse mémoire. Pour cela, il faudra fatalement convertir l'identifiant d'objet en une adresse mémoire. Cette conversion s'effectuera dans la MMU, mais la méthode de conversion dépend de la conception du processeur, aussi il sera difficile de faire des généralités sur le sujet. Toujours est-il que le processeur doit contenir une liste de correspondance entre objet et adresse de celui-ci. On peut préciser que ces techniques s'appuient souvent sur la segmentation. Chaque objet est stocké dans un segment, qui commence à une adresse physique bien précise. Les attributs de l'objet sont stockés dans ce segment, à une place prédefinie. L'identifiant est alors simplement l'adresse logique, virtuelle, du segment qui contient l'objet. Mais d'autres techniques peuvent être possibles.

Premier exemple : le processeur Rekursiv

Nous allons commencer par aborder le processeur Rekursiv. Ne soyez pas perturbé par son nom : il ne s'agit pas d'une coïncidence, comme on le verra plus tard. Ce processeur fut inventé par la compagnie Linn Product, un fabricant de matériel Hi-Fi, qui voulait améliorer ses chaînes de production automatisées. Celles-ci fonctionnaient avec un ordinateur DEC VAX assez correct pour l'époque. Cette compagnie avait lancé un grand projet de rajeunissement de sa chaîne de production. Au tout début, le projet consistait simplement à créer de nouveaux outils logiciels pour faciliter le fonctionnement de la chaîne de production. Au cours de ce projet, un langage de programmation orienté objet, le Lingo, fut créé dans ce but. Mais les programmes créés dans ce langage fonctionnaient vraiment lentement sur les DEC VAX de l'entreprise. L'entreprise, qui n'avait pas hésité à créer un nouveau langage de programmation pour ce projet, prit ce problème de performances à bras le corps et décida carrément d'inventer un nouveau processeur spécialement adapté à Lingo. Ainsi naquit le processeur Rekursiv, premier processeur orienté objet de son genre. Rekursiv était au départ prévu pour être utilisé sur des stations de travail Sun 3. Mais malgré ses nombreuses qualités, Rekursiv ne s'est pas beaucoup vendu dans le monde : à peine 20 exemplaires furent vendus. La majorité des acheteurs étaient des chercheurs en architecture des ordinateurs, et rares furent les entreprises qui achetèrent des processeurs Rekursiv. Il faut dire que ce processeur était relativement spécialisé et difficile à utiliser, sans compter que d'autres processeurs concurrents firent leur apparition, comme l'Intel 432. Ce processeur fut donc un échec commercial retentissant, malgré une réussite technique indéniable.

Vu de loin, ce processeur ressemble à un processeur tout à fait normal, découpé en quatre grands circuits principaux bien connus :

- Numerik : l'unité de calcul ;
- Logik : le séquenceur ;
- Objekt : une MMU orientée objet ;
- et Klock, une unité regroupant des timers et un générateur d'horloge.

Le support du paradigme objet était géré par Logik et par Objekt, aussi nous verrons plus en détail leurs possibilités dans la suite de ce tutoriel. Mais nous n'allons pas passer sous silence Numerik et Klock. Klock est chargée de synchroniser les différents composants de ce processeur. Plus précisément, elle contient des timers, des composants permettant de mesurer des durées, et de quoi générer le signal d'horloge du processeur. Ce processeur avait une fréquence d'environ 10 Mhz, ce qui n'était pas si mal pour l'époque.

Numerik est le nom donné à l'ALU de ce processeur. Son jeu d'instructions est donc assez limité. On peut néanmoins dire que cette unité de calcul contient un circuit capable d'effectuer des multiplications ainsi qu'un barrel shifter, un circuit capable d'effectuer des instructions de décalage et de rotation. Cette unité de calcul est rattachée à 16 registres 32 bits, rassemblés dans un register file. Numerik est capable de manipuler des nombres de 32 bits. Cette unité de calcul est un peu particulière : elle est formée de petites unités de calcul 4 bits, de marque AMD : des

AMD2900, pour être précis. Ces unités de calcul AMD de 4 bits sont reliées entre elles pour former Numerik. Cette technique qui consiste à créer des unités de calcul plus grosses à partir d'unités de calcul plus élémentaires s'appelle en jargon technique du Bit Slicing.

Son séquenceur était micro-codé, son Control Store de Rekursiv n'étant autre qu'une mémoire d'environ 64 kibioctets. Elle était accessible en écriture, ce qui fait qu'il était parfaitement possible de reprogrammer le jeu d'instructions du processeur sans restrictions. Cela pouvait même se faire à l'exécution, ce qui pouvait servir à adapter le jeu d'instructions à un langage particulier, voire l'adapter temporairement à un objet que l'on était en train de manipuler. L'ensemble des instructions du processeur (son jeu d'instructions) était donc programmable ! Ce processeur possédait même une petite particularité : on pouvait, de par l'organisation de son micro-code, créer des instructions récursives ! Ainsi, les instructions de copie ou de recherche dans un arbre ou une liste présentes dans son jeu d'instructions étaient codées via ce genre d'instructions récursives.

Certaines des instructions du processeur étaient adaptées au paradigme objet et permettaient de gérer plus simplement les diverses fonctionnalités orientées objet au niveau du matériel. Ces instructions étaient toutes micro-codées, bien évidemment. Par exemple, il existait des instructions CREATE, chacune capable de créer un objet d'une certaine classe. Cette instruction n'était autre qu'un constructeur pour un certain type d'objet. Elle avait besoin de certaines données pour fonctionner (sa taille, son type et les valeurs initiales de ses attributs) qui étaient passées par la pile vue ci-dessus. Il existait aussi des instructions de transfert de messages entre objets (comme SEND), des instructions pour accéder à un champ d'un objet localisé sur la pile (GET), pour modifier un champ dans l'objet (PUT), et bien pire encore. On peut signaler que ces instructions ne pouvaient opérer que sur certains types d'objets : certaines instructions ne pouvaient ainsi manipuler que des objets d'une certaine classe et pas d'une autre.

Mais ce qui fait que Rekursiv était un processeur orienté objet ne vient pas seulement de son jeu d'instructions : le principal intérêt de Rekursiv tient dans son unité chargée de gérer la mémoire. Les capacités de ce processeur étaient codées sur 40 bits. Objekt, la MMU, se chargeait de convertir les capacités en adresses mémoires de façon transparente pour les instructions. La MMU stockait diverses informations sur chaque objet : ainsi, la MMU pouvait retrouver l'adresse mémoire de l'objet, son type, et sa taille à partir de l'identifiant. Ces informations étaient stockées dans la mémoire, dans une table segments dédiée. Du fait de l'usage de la segmentation, un objet gardait en permanence la même capacité. On pouvait déplacer l'objet dans la mémoire, son identifiant restait le même (alors que son adresse mémoire changeait). De même, la MMU pouvait décider de déplacer un objet sur le disque dur sans que le programme ne s'en aperçoive.

Objekt implémentait un Garbage Collector matériel assez simple, mais suffisamment efficace. Pour rappel, un garbage collector, ou ramasse-miettes, est un programme ou un système matériel qui se charge de supprimer de la mémoire les objets ou données dont on n'a plus besoin. Dans certains langages de programmation comme le C ou le C++, on est obligé de libérer la mémoire à la main. Ce n'est pas le cas pour certains langages orientés objet, comme JAVA ou Lingo : un garbage collector permet de gérer la mémoire automatiquement, sans demander au programmeur de se fatiguer à le faire lui-même (du moins, en théorie). Ce garbage collector avait souvent besoin de déplacer des objets pour faire un peu de place en mémoire, et compacter les objets ensemble, pour faire de la place. Le fait que les objets soient manipulés avec une capacité facilitait énormément le travail du garbage collector matériel.

Intel iAPX 432

Passons maintenant à un autre processeur orienté objet un peu plus connu : l'Intel iAPX 432. Oui, vous avez bien lu : Intel a bel et bien réalisé un processeur orienté objet dans sa jeunesse. La conception du processeur Intel iAPX 432 commença en 1975, afin de créer un successeur digne de ce nom aux processeurs 8008 et 8080. Ce processeur s'est très faiblement vendu. Il faut dire que ce processeur avait des performances assez désastreuses et des défauts techniques certains. Par exemple, ce processeur ne contenait pas de mémoire cache et n'avait pas de registres (c'était une machine à pile). Autre détail : ce processeur ne pouvait pas effectuer directement de calculs avec des constantes entières autres que 0 et 1 (une sombre histoire de mode d'adressage immédiat non supporté). De plus, celui-ci avait été conçu pour exécuter en grande partie le langage ADA, un langage de programmation orienté objet très sécurisé et très utilisé dans le domaine de l'embarqué pour ses qualités intrinsèques. Malheureusement, le compilateur qui traduisait les codes sources écrits en ADA en programmes compréhensibles par le processeur était mauvais et avait une certaine tendance à massacrer les performances. Sans compter que l'ADA n'était pas très populaire chez les programmeurs et n'était utilisé que par les militaires et les entreprises travaillant sur des systèmes embarqués ou critiques, ce qui n'a pas aidé à faire vendre ce processeur.

Modèle mémoire

Ce processeur utilisait la segmentation pour définir ses objets. Chaque objet était stocké dans un segment : toutes ses données et autres informations permettant de déterminer intégralement l'état d'un objet étaient regroupées dans un de ces segments. Chacun de ces segments est découpé en deux parties de tailles égales : une partie contenant les données d'un objet, et des informations supplémentaires destinées à gérer l'objet correctement. Ces informations pouvaient être des capacités pointant vers d'autres objets (pour créer des objets assez complexes), par exemple. Au fait : sur ce processeur, les capacités étaient appelées des Access Descriptors. Sur l'Intel iAPX432, chaque segment (et donc chaque objet) pouvait mesurer jusqu'à 64 kibioctets : c'est très peu, mais suffisant pour stocker des objets suffisamment gros pour que cela ne pose pas vraiment de problèmes. Ce processeur pouvait gérer jusqu'à 2^{24} segments différents.

L'Intel 432 possédait dans ses circuits un garbage collector matériel. Pour faciliter son fonctionnement, certains bits de l'object descriptor des objets permettaient de savoir si l'objet en question pouvait être supprimé ou non.

Support de l'orienté objet

L'Intel 432 est conçu pour permettre l'utilisation de « types », de classes de base, déjà implémentées dans le processeur, mais cela ne suffit évidemment pas à supporter la programmation orientée objet. Pour cela, le processeur permet de définir ses propres classes, utilisables au besoin, et définies par le programmeur.

L'Intel 432 permet, à partir de fonctions définies par le programmeur, de créer des domaines objects, qui contiennent un ensemble de capacités pointant chacune vers des fonctions. Chacune de ces fonctions peut accéder à un nombre restreint d'objets, tous du même type, et à rien d'autre. Chaque domaine object est divisé en deux parties : une partie publique, qui contient des capability identifiant les fonctions exécutables au besoin par tout morceau de code ayant accès au domaine object, et une partie privée, qui contient des capability identifiant des fonctions/méthodes internes au domaine object : seules les fonctions déclarées dans la partie publique possèdent des capability pointant vers ces fonctions et donc, seules ces fonctions de la partie publique peuvent les utiliser. En clair, ces domaines objects ne sont rien de moins que l'ensemble des méthodes d'une classe ! Chacun de ces domaines objects possède une capacité rien que pour lui, qui permettait de l'identifier et que l'on devait utiliser pour accéder aux fonctions qu'il contient. Évidemment, ce processeur supportait de nombreuses instructions et fonctionnalités permettant à des capacités pointant vers des fonctions publiques d'être présentes dans des domaines objects différents. Celles-ci pouvaient être paramétrées de façon plus ou moins fine afin de choisir quelles fonctions d'un domaine object devaient être partagées ou non. Cela permettait de supporter des fonctionnalités objet telles que l'héritage, l'inheritance, etc.

Sur ce processeur, chaque processus est considéré comme un objet à part entière. De même, l'état du processeur (le programme qu'il est en train d'exécuter, son état, etc.) est défini par un objet, stocké en mémoire, qu'il est possible de manipuler : toute manipulation de cet objet permettra d'effectuer une action particulière sur notre processeur. Il en est de même pour chaque fonction présente en mémoire : elle était encapsulée dans un objet, sur lequel seules quelques manipulations étaient possibles (l'exécuter, notamment). Et ne parlons pas des appels de fonctions qui stockaient l'état de l'appelé directement dans un objet spécial. Bref, de nombreux objets système sont prédefinis par le processeur : les objets stockant des fonctions, les objets stockant des processus, etc.

Il est aussi possible pour le programmeur de définir de nouveaux types non supportés par le processeur, en faisant appel au système d'exploitation de l'ordinateur. Au niveau du processeur, chaque objet est typé au niveau de son object descriptor : celui-ci contient des informations qui permettent de déterminer le type de l'objet. Chaque type se voit attribuer un domaine object qui contient toutes les fonctions capables de manipuler les objets de ce type et que l'on appelle le type manager. Lorsque l'on veut manipuler un objet d'un certain type, il suffit d'accéder à une capacité spéciale (le TCO) qui pointera dans ce type manager et qui précisera quel est l'objet à manipuler (en sélectionnant la bonne entrée dans la liste de capacité). Le type d'un objet prédefini par le processeur est ainsi spécifié par une suite de 8 bits, tandis que le type d'un objet défini par le

programmeur est défini par la capacité spéciale pointant vers son type manager.

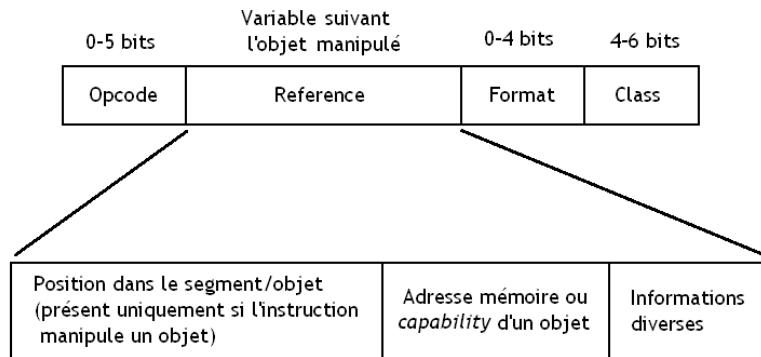
Jeu d'instruction

Ce processeur est capable d'exécuter pas moins de 230 instructions différentes. Le processeur supporte certains objets de base, prédéfinis dans le processeur. Il fournit des instructions spécialement dédiées à la manipulation de ces objets, et contient notamment des instructions d'appel de fonction assez élaborées. Il contient aussi des instructions n'ayant rien à voir avec nos objets, qui permettent de manipuler des nombres entiers, des caractères, des chaînes de caractères, des tableaux, etc. Beaucoup de ces instructions sont micro-codées. Le processeur est une machine à pile.

Ce processeur contenait aussi des instructions spécialement dédiés à la programmation système et idéales pour programmer des systèmes d'exploitation. De nombreuses instructions permettaient ainsi de commuter des processus, faire des transferts de messages entre processus, etc. Environ 40 % du micro-code était ainsi spécialement dédié à ces instructions spéciales. Ces instructions chargées de prendre en charge le travail d'un système d'exploitation étaient des manipulations comme un changement de contexte ou un passage de message entre processus et se contentaient de faire des manipulations sur des objets représentant le processeur, des processus, ou d'autres choses dans le genre.

On peut aussi préciser que ces instructions sont de longueur variable. Sur un processeur normal, les instructions ont une longueur qui est souvent multiple d'un octet, mais notre Intel iAPX 432 fait exception à cette règle : ses instructions peuvent prendre n'importe quelle taille comprise entre 10 et 300 bits, sans vraiment de restriction de taille. Sur l'Intel iAPX 432, les bits d'une instruction sont regroupés en 4 grands blocs, 4 champs, qui ont chacun une signification particulière. Comme vous allez le voir dans un instant, l'encodage des instructions reflète directement l'organisation de la mémoire en segments : le jeu d'instructions a dû s'adapter à l'organisation de la mémoire.

- Le premier champ s'appelle classe. Il permet de dire combien de données différentes l'instruction va devoir manipuler, et quelles seront leurs tailles.
- Le second champ, le champ format, n'utilise que 4 bits et a pour but de préciser si les données à manipuler sont en mémoire ou sur la pile.
- Le troisième champ, référence, doit être interprété différemment suivant la donnée à manipuler. Si cette donnée est un entier, un caractère ou un flottant, ce champ indique l'emplacement de la donnée en mémoire. Alors que si l'instruction manipule un objet, ce champ spécifie la capability de l'objet en question. Ce champ est assez complexe et il est sacrément bien organisé.
- Le dernier champ s'appelle l'opcode et permet d'identifier l'instruction à effectuer : s'agit-il d'une addition, d'une création d'objet, d'un passage de message entre deux processus, d'une copie d'un objet sur la pile, etc.



Conclusion

Pour ceux qui veulent en savoir plus, je conseille la lecture de ce livre, disponible gratuitement sur internet (merci à l'auteur pour cette mise à disposition) :

- Capability-Based Computer Systems (<https://homes.cs.washington.edu/~levy/capabook/>).

Pour avoir plus d'informations sur le jeu d'instructions du processeur Rekursiv, voyez le lien suivant :

- Jeu d'instruction du processeur Rekursiv (<http://www.ncl.ac.uk/computing/research/seminars/pdfs/chapters/129.pdf>).

Et enfin, voici un document à propos du processeur HISC :

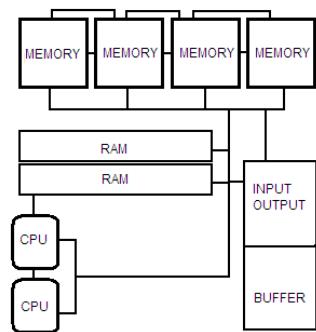
- Processeur HISC (<http://www.ee.cityu.edu.hk/~hisc/HISC.pdf>)

Les architectures tolérantes aux pannes

Un ordinateur n'est jamais un composant parfait, ce qui fait que des pannes peuvent survenir de temps à autre. Ces pannes peuvent être aussi bien logicielles (un logiciel qui plante ou qui a un bug), que matérielles (un composant cesse de fonctionner ou qui donne un résultat faux). Certaines erreurs peuvent être transitoires et ne se manifester qu'occasionnellement, tandis que d'autres sont de vraies pannes qui empêchent le bon fonctionnement d'un ordinateur tant qu'elle ne sont pas résolues. Pour donner un exemple de panne transitoire, on peut citer l'incident de Schaeerbeek. Le 18 mai 2003, dans la petite ville belge de Schaeerbeek, on constata une erreur sur la machine à voter électronique de la commune : il y avait un écart de 4096 voix en faveur d'un candidat entre le dépouillement traditionnel et le dépouillement électronique. Mais ce n'était pas une fraude : le coupable était un rayon cosmique, qui avait modifié l'état d'un bit de la mémoire de la machine à voter. Et si les pannes causées par des rayons cosmiques sont rares, d'autres pannes peuvent avoir des conséquences similaires.

Dans des milieux comme l'aéronautique, les satellites, ou dans tout système dit critique, on ne peut pas se permettre que de telles pannes aient des conséquences : des vies peuvent être en jeu. Dans une telle situation, on doit limiter l'impact des pannes. Pour cela, il existe des systèmes tolérants aux pannes, qui peuvent continuer de fonctionner, même en ayant un ou plusieurs composants en panne. Cette tolérance aux pannes se base sur la redondance : on duplique du matériel, des données, ou des logiciels en plusieurs exemplaires. Ainsi, si un exemplaire tombe en panne, les autres pourront prendre la relève. Ces solutions matérielles pour tolérer les pannes sont relativement coûteuses, dupliquer du matériel n'étant pas sans cout. Aussi, les méthodes vues dans ce cours ne s'utilisent que pour des cas bien précis, où le besoin de fiabilité est fort, typiquement dans les grosses industries (aéronautique, spatiale, ferroviaire, automobile et autres). Il est peu probable qu'un développeur lambda aie affaire à ce genre d'architecture, alors que la redondance de données peut très bien faire partie de son quotidien. Dans nos ordinateurs, cette redondance peut prendre plusieurs formes :

- une redondance des données, qui est à la base des codes correcteurs d'erreur et des systèmes RAID ;
- une redondance matérielle : on duplique des serveurs, des unités de calcul, des processeurs ou de la mémoire, des disques durs (RAID), et ainsi de suite ;
- une redondance logicielle, où plusieurs exemplaires d'un même programme font leurs calculs dans leur coin.

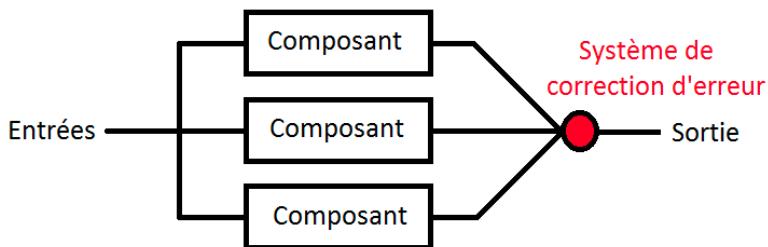


De manière générale, les architectures tolérantes aux pannes dupliquent du matériel, que ce soit des ordinateurs, des composants (processeurs, mémoires, disques durs), voire des portions de composants (cœurs de processeurs, unités de calcul) : si un composant tombe en panne, les autres permettent au système de fonctionner. Par exemple, on peut utiliser plusieurs ordinateurs identiques, qui font la même chose en parallèle : si un ordinateur tombe en panne, les autres prendront le relais. Comme autre exemple, on peut utiliser plusieurs processeurs ou dupliquer les unités de calcul dans un processeur. On peut classer les techniques de redondances matérielles en deux :

- les méthodes actives, où l'on doit détecter les erreurs et reconfigurer le circuit pour corriger la panne ;
- les méthodes passives, qui masquent les erreurs sans pour autant faire quoique ce soit sur le composant fautif ;
- les méthodes hybrides, qui mélègent les méthodes passives et les méthodes actives.

Redondance matérielle passive

Avec la redondance matérielle passive, tous les composants travaillent en parallèle : ils reçoivent les données en entrée, les traitent, et fournissent un résultat plus ou moins en même temps. La sortie des composants est reliée à un système qui se chargera de corriger les erreurs ou fautes en sortie, sans pour autant les détecter. Par exemple, on peut imaginer ce que cela donnerait avec des unités de calcul redondantes : toutes les unités de calcul recevraient les opérandes en même temps, feraient leurs calculs indépendamment les unes des autres, et fourniraient leur résultat à un système qui corrigerait d'éventuelles erreurs de calcul ou pannes.



Vote à majorité

Dans la plupart des cas, le système de correction des erreurs se base sur ce qu'on appelle un **vote à majorité**. Avec le vote à majorité "classique", si différentes valeurs sont disponibles sur ses entrées, il prend simplement la valeur majoritaire sur les autres. Par exemple, prenons le cas avec 5 composants : si un composant tombe en panne, les quatre autres donneront un résultat correct : à 4 sorties contre une, c'est le résultat correct qui l'emportera. Il faut savoir que cette méthode ne fonctionne convenablement que si le nombre de composants est impair : dans le cas contraire, on peut avoir autant de composants en panne que de composants fonctionnels, ce qui fait qu'aucune majorité ne peut être dégagée. Si le nombre de composants en panne est inférieur au nombre de composants sans panne, ce système de vote à majorité donnera systématiquement le bon résultat. Ainsi, utiliser 3 composants permet de résister à une panne de composant, utiliser 5 composants permet de résister à une panne de 2 composants, en utiliser 7 permet de résister à 3 composants en panne, etc.

Unité de calcul 1 :	0 1 0 1 1 1 0 1
Unité de calcul 2 :	1 1 1 1 0 1 0 1
Unité de calcul 3 :	0 1 0 1 1 1 0 1
Unité de calcul 4 :	1 1 1 1 0 1 0 1
Unité de calcul 5 :	0 1 0 1 1 1 0 1
Unité de calcul 6 :	0 1 0 1 1 1 0 1
Unité de calcul 7 :	0 1 0 1 1 1 0 1

Le résultat 0 1 0 1 1 1 0 1 est majoritaire

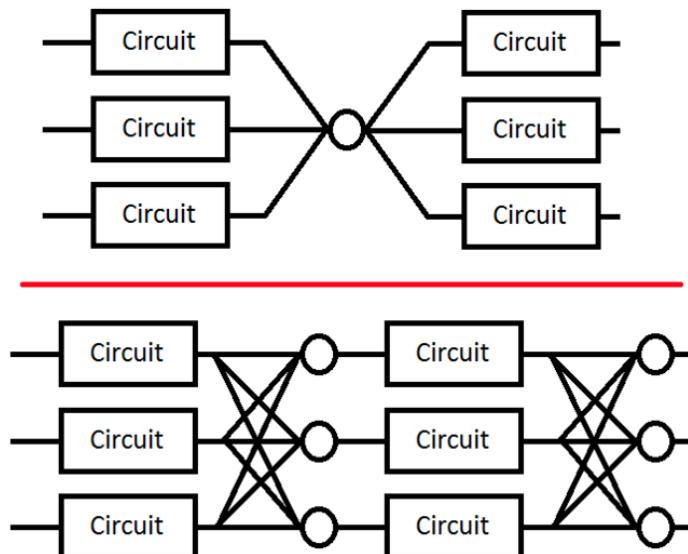
Ce vote à majorité peut aussi s'effectuer non au niveau du résultat, mais au niveau des bits. Dans ce cas, le circuit de correction d'erreur va placer les bits des différents résultats sur la même colonne, et choisit pour chaque colonne le bit qui est majoritaire. Ce calcul peut s'effectuer naturellement en utilisant ces portes à majorité, des portes logiques spécifiquement conçues pour déterminer quel est le bit majoritaire sur ses entrées. Des variantes de ce système de vote existent. Celles-ci consistent à prendre non pas le résultat ou bit majoritaire, mais seulement le plus fréquent (ou la médiane des différents résultats).

0 1 0 1 1 1 0 1	1	Le bit majoritaire dans la colonne est 1
1 1 1 1 0 1 0 1	1	
1 1 0 0 1 1 1 0	0	

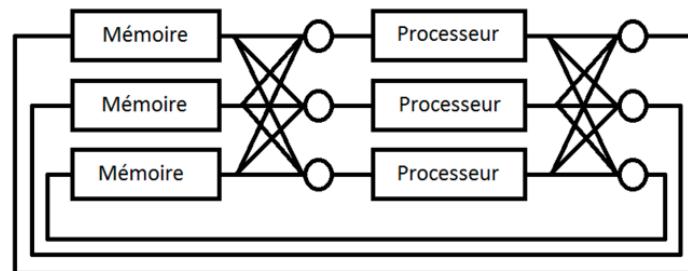
1 1 0 1 1 1 0 1

Implémentation

Ce mécanisme fonctionne très bien, à un détail près : le circuit de vote à majorité est un point faillible du système : s'il tombe en panne, tout le système tombe en panne. Pour éviter cela, il est là encore possible de dupliquer ce système de vote à majorité, ce qui est utilisé quand le résultat doit être réutilisé par d'autres (qui sont eux-mêmes dupliqués).



Ce système de vote à majorité peut s'utiliser pour les communications avec la mémoire. Il peut notamment servir pour gérer les lectures ou écritures dans une mémoire, voire les deux. On peut aussi l'utiliser pour gérer les communications à l'intérieur d'un composant. Par exemple, on pourrait imaginer utiliser ces méthodes pour l'unité de calcul et les registres : dans les schémas suivants, il suffirait de remplacer la mémoire par les bancs de registre (register files) et le processeur par l'unité de calcul. Bref, les possibilités sont relativement nombreuses.



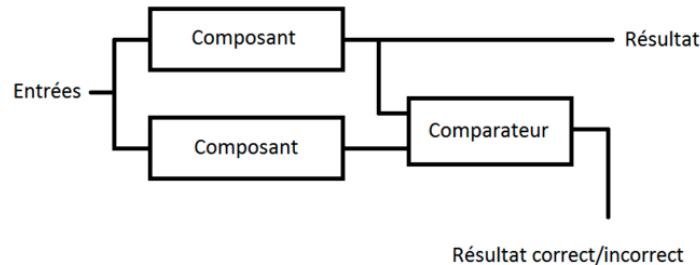
Redondance matérielle active

La redondance active ne masque pas les pannes comme peut le faire la redondance passive. Elle va détecter les pannes et passer le relais du composant en panne à un composant fonctionnel. Ces méthodes se déclinent en deux grandes catégories :

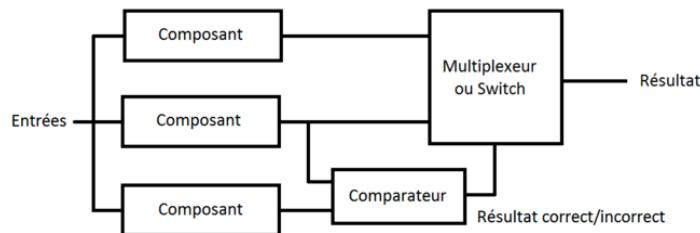
- d'un côté celles où les composants travaillent en parallèle et où la reconfiguration du circuit suffit ;
- de l'autre, celles où un seul composant fonctionne à la fois, et où la correction d'une panne demande de reprendre les calculs de zéro.

Duplication avec comparaison

Une première technique de redondance active se contente de dupliquer le composant en un composant principal et un composant de réserve. On peut alors détecter une erreur en comparant la sortie des deux composants : si elle est différente, on est certain qu'il y a eu une erreur (on suppose qu'il n'y en a pas eu en cas d'accord entre les deux composants). Une fois l'erreur détectée, on ne peut cependant pas la corriger. Le premier processeur à utiliser cette méthode était l'EDVAC, dans les années 1950. Il comprenait deux unités de calcul, et continuait d'exécuter son programme tant que les deux unités de calcul donnaient des résultats identiques. En cas de non-agrément entre les deux unités de calcul, le processeur ré-exécutait l'instruction fautive.

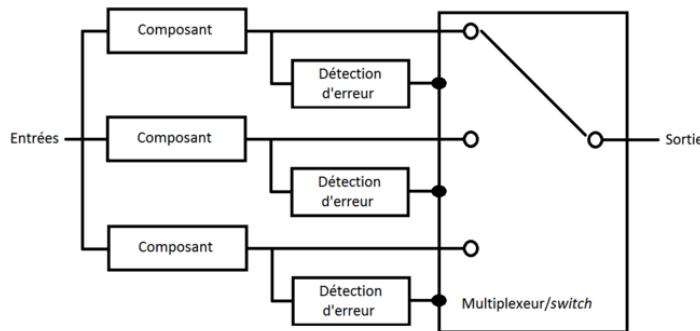


On peut améliorer ce circuit afin qu'il puisse corriger l'erreur. Pour cela, on rajoute un troisième composant de réserve, dont on suppose qu'il ne sera pas en panne. Si une erreur est détectée par le comparateur, on préfère utiliser la sortie du composant de réserve.



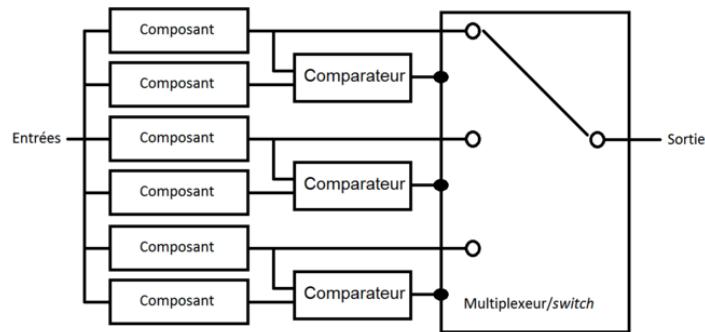
Standby Sparing

Avec cette méthode, le système de correction des pannes choisit un résultat parmi ceux qu'il reçoit, et considère que ce résultat est le bon. En somme, il choisit la sortie d'un composant parmi toutes les autres : c'est donc un multiplexeur ou un switch. Quand le composant choisi tombe en panne, le multiplexeur/switch se reconfigure et choisit alors une autre sortie (celle d'un autre composant). Reste que cette configuration du switch demande de détecter les pannes, afin de commander le multiplexeur switch. On trouve donc, pour chaque composant, un système de détection des pannes, ainsi qu'un circuit combinatoire qui commande le multiplexeur/switch.



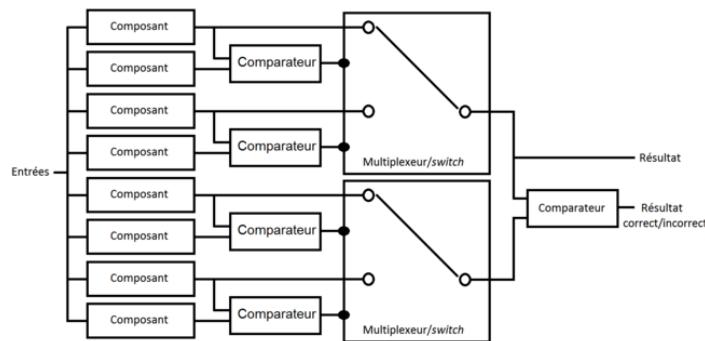
Pair And Spare

Il faut noter que les deux techniques précédentes sont loin d'être incompatibles. On peut notamment les utiliser de concert : la technique de duplication par comparaison peut être utilisée pour détecter les erreurs, et la technique du Standby Sparing pour effectuer la correction. On peut aussi faire l'inverse.



Spare and Pair

Un exemple typique est l'architecture Stratus (aussi connue IBM/System 88). Celui-ci contient quatre processeurs logiques qui font leurs calculs en parallèle : le résultat est choisi parmi les processeurs sans pannes. Une panne ou erreur est détectée avec duplication par comparaison : chaque processeur logique est dupliqué et une panne est détectée si les deux processeurs sont en désaccord sur le résultat. L'ensemble contient donc huit processeurs.



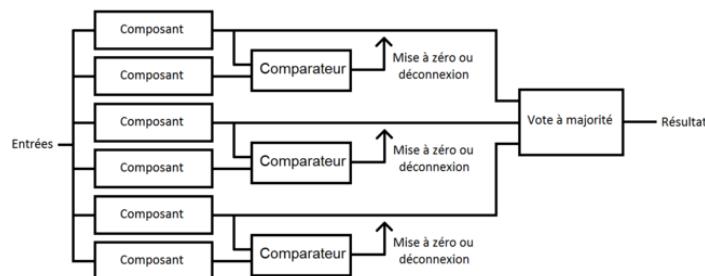
Redondance matérielle hybride

Les méthodes de redondance hybride mélangeant les techniques vues plus haut. Il en existe grossièrement trois principales :

- la redondance passive avec composants de réserve ;
- la redondance passive auto-correctrice ;
- la redondance à triple duplex.

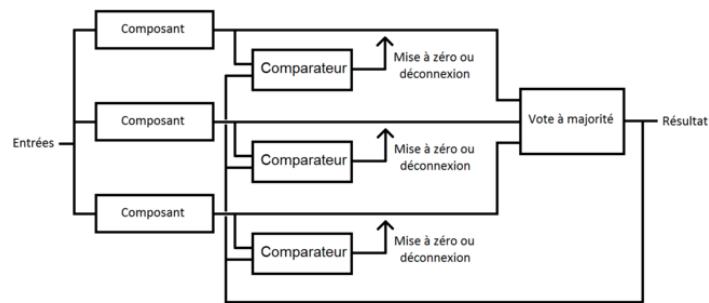
Redondance à triple duplex

Avec la redondance à triple duplex, plusieurs composants qui utilisent la duplication avec comparaison sont suivis par une porte à majorité. Le principe de cette technique est simple : si un composant est en panne, alors son résultat ne doit pas être pris en compte dans le calcul du vote à majorité.



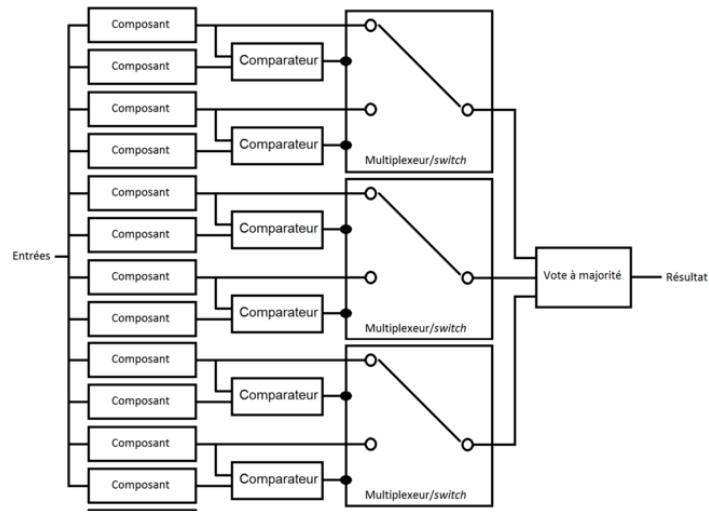
Redondance passive auto-correctrice

La redondance passive auto-correctrice est similaire à la technique précédente, à un détail près : on n'utilise pas vraiment la duplication par comparaison de la même manière. Le principe de cette technique est le même que la précédente : si un composant est en panne, alors son résultat ne doit pas être pris en compte. Sauf que cette fois-ci, on détecte une panne en comparant le résultat du composant avec le vote majoritaire : il y a une panne si les deux ne sont pas identiques. Ainsi, au lieu d'avoir deux composants en entrée du comparateur, on n'en aura qu'un seul : l'autre entrée du comparateur sera reliée à la sortie de la porte à majorité.



Redondance passive avec composants de réserve

Avec la redondance passive avec composants de réserve, plusieurs modules qui utilisent la redondance active sont suivis par un système de vote à majorité.



Les architectures parallèles

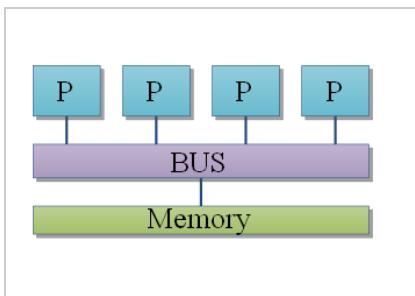
De nos jours, il n'est pas rare de posséder des processeurs contenant plusieurs cœurs. L'utilité de tels processeurs est très simple : la performance ! De tels processeurs exécutent des instructions indépendantes dans des processeurs séparés, ce qui porte le doux nom de **parallelisme**. Mais les processeurs multicoeurs ne sont pas les seuls processeurs permettant de faire ceci : entre les ordinateurs embarquant plusieurs processeurs, les architectures dataflow, les processeurs vectoriels et les autres architectures parallèles, il y a de quoi être perdu. Pour se faciliter la tâche, diverses classifications ont été inventées pour mettre un peu d'ordre dans cette jungle de processeurs.

Partage de la mémoire

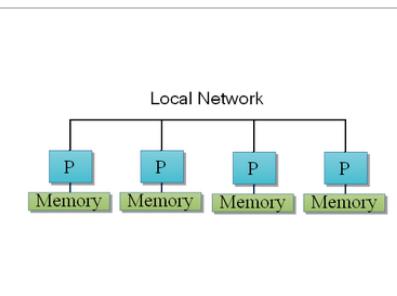
On peut classer les architecture suivant la façon dont les processeurs doivent se partager la mémoire est aussi très importante. Dans le premier cas, on a une **mémoire partagée** entre tous les processeurs. Avec ce genre d'architecture, rien n'empêche plusieurs processeurs de vouloir accéder à la mémoire en même temps, ce qui n'est pas possible sans mémoires multiports. Il va donc falloir trouver des moyens pour arbitrer les accès à la mémoire entre les processeurs. Pour cela, le matériel fournit des instructions pour faciliter la communication ou la synchronisation entre les différents morceaux de programmes (interruptions inter-processeurs, instructions atomiques, et autres). Les couts de synchronisation peuvent être conséquents et peuvent réduire les performances.

Viennent ensuite les **architectures distribuées** : chaque processeur possède sa propre mémoire et les processeurs sont reliés par un réseau local (ou par internet). Les processeurs peuvent ainsi accéder à la mémoire d'un autre processeur via le réseau local : il leur suffit de faire une demande au processeur qui détient la donnée. Cette demande va traverser le réseau local et arriver à son destinataire : la donnée demandée est alors envoyée via le réseau local et est copiée dans la mémoire locale de l'ordinateur demandeur. Il va de soi que les communications entre les différents processeurs peuvent prendre un temps relativement long, et que ceux-ci sont loin d'être négligeables. Avec une organisation de ce genre, la qualité et les performances du réseau reliant les ordinateurs est très important pour les performances. Il va de soi que les communications sur le réseau peuvent prendre du temps : plus le réseau est rapide, meilleures seront les performances.

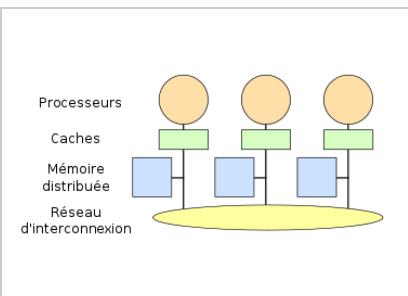
Il va de soi qu'il est possible d'utiliser des caches aussi bien sur des architectures distribuées qu'avec la mémoire partagée. Néanmoins, la gestion de ces caches peut poser quelques problèmes, comme nous le verrons d'ici quelques chapitres. Ces problèmes surviennent quand une donnée est présente dans plusieurs caches distincts : chaque processeur pouvant modifier cette donnée indépendamment des autres, les données dans un cache peuvent ne pas être à jour. Or, un processeur doit toujours avoir, pour éviter tout problème, une valeur jour de la donnée dans l'ensemble de ses caches : cela s'appelle la **cohérence des caches**. Pour cela, les mises à jour dans un cache doivent être propagées dans les caches des autres processeurs. Nous aborderons le sujet en détail plus tard, dans quelques chapitres.



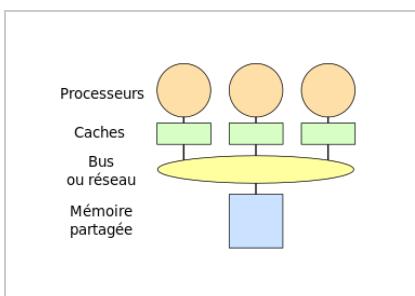
Mémoire partagée.



Architecture distribuée.



Architecture distribuée avec des caches.

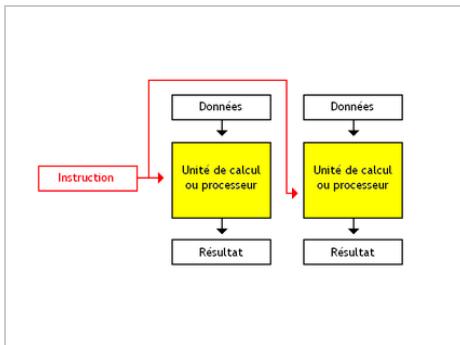


Architecture à mémoire partagée avec des caches dédiés pour chaque CPU.

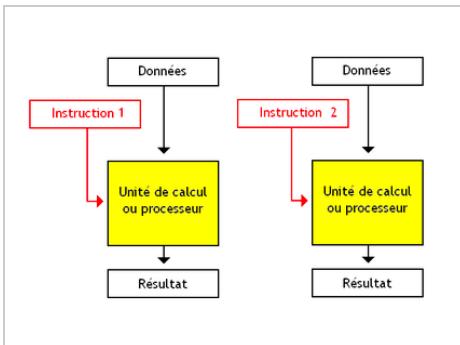
Taxonomie de Flynn

Dans les années 1966, un scientifique américain assez connu dans le milieu du hardware qui se nomme Flynn a classé ces architectures en 4 grandes catégories : SISD, SIMD, MISD, et MIMD. Cette classification a remarquablement tenu le coup au fil du temps. Mais elle n'est pas parfaite, et certaines architectures sont difficiles à classer. Mais cette classification, bien que simpliste, est particulièrement didactique.

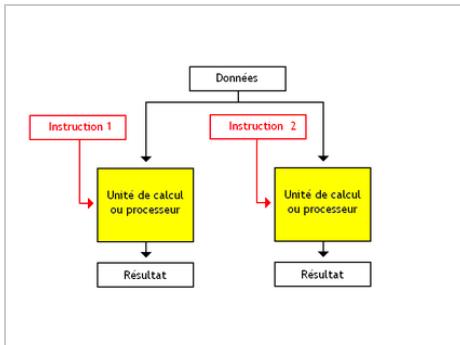
Architecture	Description
Architectures SISD	Processeurs incapables de toute forme de parallélisme.
Architectures SIMD	Architectures capables d'exécuter une instruction sur plusieurs données à la fois. Avec cette solution, les processeurs exécutent un seul et unique programme ou une suite d'instructions, mais chaque processeur travaille sur une donnée différente.
Architectures MISD	Exécutent des instructions différentes en parallèle sur une donnée identique. Autant prévenir tout de suite : on ne verra aucun exemple de ce type dans le tutoriel. Cette catégorie d'architectures est vraiment très rare. On peut citer comme exemples d'architectures MISD les architectures systoliques et cellulaires.
Architectures MIMD	Architectures capables d'exécuter des instructions différentes sur des données différentes. Les processeurs multicœurs et multiprocesseurs font partie de la catégorie MIMD. La catégorie MIMD est elle-même découpée en deux sous-catégories. <ul style="list-style-type: none"> ■ Le Single Program Multiple Data, ou SPMD, consiste à exécuter un seul programme sur plusieurs données à la fois : la différence avec le SIMD est que le SPMD peut exécuter des instructions différentes d'un même programme sur des données. ■ Le Multiple Program Multiple Data, qui consiste à exécuter des programmes en parallèle sur des données différentes. On peut ainsi découper un programme en sous-programmes indépendants exécutables en parallèle ou exécuter plusieurs copies d'un programme. Dans les deux cas, chaque copie ou morceau de programme est appelé un thread.



SIMD - taxonomie de Flynn



MIMD - taxonomie de Flynn



MISD - taxonomie de Flynn

\	Données traitées en série	Données traitées en parallèle
Instructions traitées en série	SISD	SIMD
Instructions traitées en parallèle	MISD	MIMD

Parallélisme de données et de tâches

Les architectures SIMD et SPMD effectuent ce qu'on appelle du **parallélisme de données**, à savoir exécuter un même flux d'instruction ou un même programme sur des données différentes, en parallèle. Les processeurs pouvant faire ce genre de chose ne sont pas rares, bien au contraire : la quasi-totalité des processeurs est aujourd'hui de ce type. Plus précisément, tous les processeurs Intel et AMD actuels, ainsi que leurs confrères de chez ARM, MIPS et VIA en sont capables. Le parallélisme de données est aussi massivement utilisé dans les cartes graphiques, qui sont des composants devant exécuter les mêmes instructions sur un grand nombre de données : chaque calcul sur un pixel est plus ou moins indépendant des transformations qu'on effectue sur ses voisins.

Ce parallélisme de données s'oppose au **parallélisme de tâches**, qui exécute plusieurs programmes en parallèle. Celui-ci s'exploite en découpant un programme en plusieurs sous-programmes indépendants qu'on peut faire exécuter en parallèle. Ces sous-programmes indépendants sont ce qu'on appelle des Threads. Il suffit de faire exécuter chaque Thread sur un processeur séparé pour pouvoir paralléliser le tout. Les architectures permettant d'exécuter des threads en parallèle sont donc des architectures multiprocesseurs ou multicœurs, ainsi que quelques autres processeurs un peu spéciaux. Avec ce genre de parallélisme, le découpage d'un programme en threads est avant tout un problème logiciel, du ressort du compilateur, du programmeur, voire du langage de programmation. Mais, plus étonnant, certains processeurs sont capables de découper un programme à l'exécution, éventuellement grâce à des indications fournies par le programme lui-même ! C'est tout de même assez rare, même si cela a déjà été tenté : on reviendra dessus quand je parlerai des architectures EDGE et du spéculative multithreading dans ce tutoriel.

Parallélisme de tâches

Ces deux formes de parallélisme n'ont pas du tout la même efficacité, comme nous allons le voir. La limite du parallélisme de tâches tient dans le fait que tous les programmes ne peuvent pas utiliser plusieurs processeurs à la perfection. Tout programme conservera une certaine portion de code qui ne profitera pas du parallélisme. Appelons cette portion de code le code série, tandis que la portion de code qui profite du parallélisme sera appelée le code parallèle. Le ou les processeurs passeront un certain temps à exécuter le code série, et un autre temps dans le code parallèle. Notons ces deux temps T_s et T_p . On sait que le temps T mis par un programme à s'exécuter vaut donc :

$$T = T_s + T_p$$

Maintenant, ajoutons une seconde hypothèse : le code parallèle profite au mieux de la présence de plusieurs processeurs. Dit autrement, si on possède N processeurs, tous auront une part égale du code parallèle, en temps d'exécution. Dans ces conditions, le temps d'exécution T du programme devient celui-ci :

$$T_{\text{parallel}} = T_s + \frac{T_p}{N}$$

Ces deux formules nous permettent de le rapport entre le temps sans parallélisme et celui avec. On a alors :

$$\frac{T_{\text{parallel}}}{T} = \frac{T_s + \frac{T_p}{N}}{T}$$

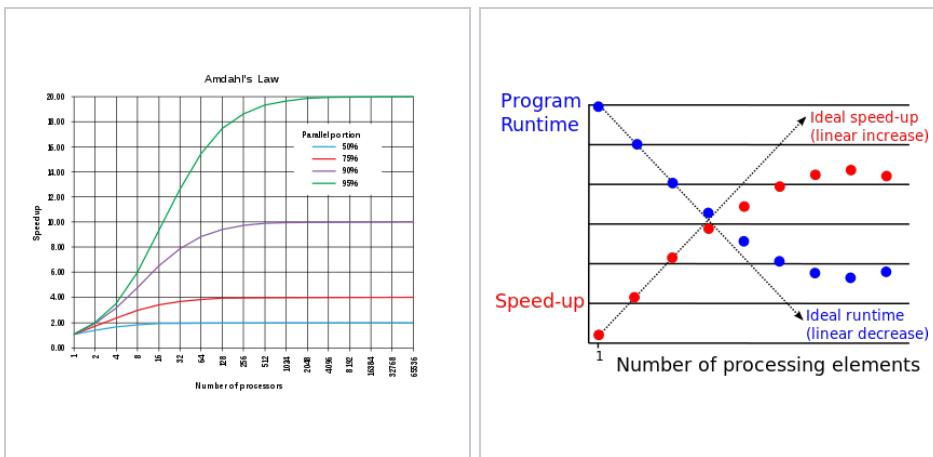
Maintenant, notons S et P le pourcentage de temps d'exécution passé respectivement dans le code série et dans le code parallèle. Par définition, ces deux pourcentages valent $\frac{T_s}{T}$ et $\frac{T_p}{T}$. Dans l'équation précédentes, on peut identifier ces deux pourcentage (leur inverse, précisément). On a alors, en faisant le remplacement :

$$\frac{T_{\text{parallel}}}{T} = S + \frac{P}{N}$$

Cette formule nous permet de calculer le gain obtenu grâce au parallélisme. Ce gain se calcule en divisant le temps avant parallélisation et le temps obtenu après. C'est intuitif : si un programme va deux fois plus vite, c'est que son temps d'exécution a été divisé par deux entre avant et après. Dit autrement, il s'agit de l'inverse du rapport calculé précédemment. L'équation obtenue ainsi, qui donne le gain en performance que l'on peut attendre en fonction du nombre de processeur et du pourcentage de code parallèle, est appelée la **loi d'Amdhal**.

$$G = \frac{1}{S + \frac{P}{N}} = \frac{1}{1 - P + \frac{P}{N}}$$

On peut déduire de cette loi qu'il existe une limite maximal du gain obtenu par parallélisme de tâche. En effet, si le nombre de processeurs devient infini, le gain tend naturellement la borne maximale $\frac{1}{S}$. Dit autrement, le temps d'exécution d'un programme ne peut pas descendre en-dessous du temps d'exécution du code série, même avec une infinité de processeurs. De plus, le rendement du parallélisme diminue rapidement avec le nombre de processeurs. Passer de 2 à 4 processeurs permet d'obtenir des gains substantiels, alors que passer de 16 à 32 processeurs ne donne de gains sensibles que si P est extrêmement élevé.



Loi d'Amdhal.

Parallelization diagram.

Il faut cependant préciser que la loi d'Amdhal est optimiste : elle a été démontrée en postulant que le code parallèle peut être réparti sur autant de processeurs qu'on veut et peut profiter d'un grand nombre de processeurs. Dans la réalité, rares sont les programmes de ce genre : certains programmes peuvent à la rigueur exploiter efficacement 2, 4, voire 8 processeurs mais pas au-delà. Elle ne tient pas compte des nombreux problèmes techniques, aussi bien logiciels que matériels qui limitent les performances des programmes conçus pour exploiter plusieurs processeurs. La loi d'Amdhal donne une borne théorique maximale au gain apporté par la présence de plusieurs processeurs, mais le gain réel sera quasiment toujours inférieur au gain calculé par la loi d'Amdhal.

Parallélisme de données

Le parallélisme de données est très utilisé pour des applications où les données sont indépendantes, et peuvent se traiter en parallèle. Un bon exemple est le calcul des graphismes d'un jeu vidéo. Chaque pixel étant colorié indépendamment des autres, on peut rendre chaque pixel en parallèle. C'est pour cela que les cartes vidéo contiennent un grand nombre de processeurs et sont des architectures SIMD capables d'exécuter un grand nombre de calculs en parallèle.

Le parallélisme de données n'a pas les limitations intrinsèques au parallélisme de tâches. En effet, celui-ci est avant tout limité certes par le code série, mais aussi par la quantité de données à traiter. Prenons le cas d'une carte graphique, par exemple : ajouter des processeurs ne permet pas de diminuer le temps de calcul, mais permet de traiter plus de données en même temps. Augmenter la résolution permet ainsi d'utiliser des processeurs qui auraient été en trop auparavant. Reste à quantifier de tels gains, si possible avec une loi similaire à celle d'Amdhal. Pour cela, nous allons partir du cas où on dispose d'un processeur qui doit traiter N données. Celui-ci met un temps T_s à exécuter le code série et un temps T_p pour traiter une donnée. On a donc :

$$T = T_s + T_p \times N$$

Avec N processeurs, le traitement des N données s'effectuera en un temps T_p , vu qu'il y a une donnée est prise en charge par un processeur, sans temps d'attente.

$$T_{\text{parallel}} = T_s + T_p \times N$$

Le gain est donc :

$$\frac{T_{\text{parallel}}}{T} = \frac{T_s + T_p \times N}{T}$$

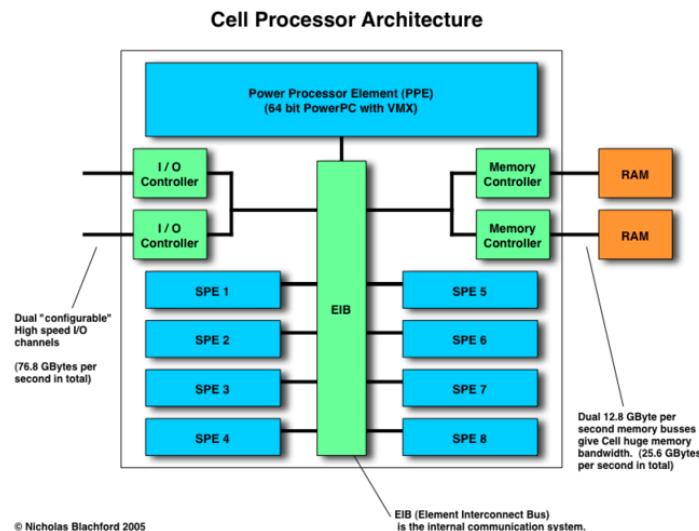
$$G = S + P \times N = 1 - P + P \times N$$

On voit que le gain est proportionnel au nombre de données à traiter. Plus on a de données à traiter, plus on gagnera à utiliser un grand nombre de processeurs. Il n'y a pas de limites similaires à celles obtenues à la loi d'Amdhal, du moins, tant qu'on dispose de suffisamment de données à traiter en parallèle.

Architectures multiprocesseurs et multicœurs

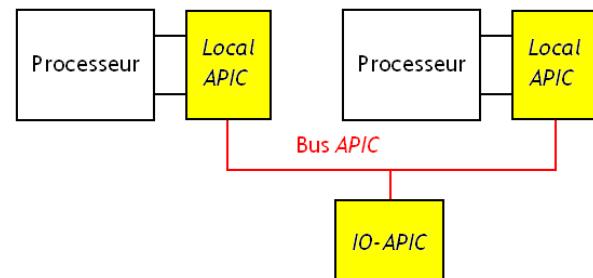
Rien ne vaut l'utilisation de plusieurs processeurs pour réellement tirer partie du parallélisme de tâches. Les premières tentatives consistaient à relier plusieurs ordinateurs via un réseau local, ou à les placer sur la même carte mère. De nos jours, il est possible de placer plusieurs processeurs sur la même puce : on obtient un **processeur multicœur** (chaque processeur s'appelle un cœur). Suivant le nombre de coeurs présents dans notre processeur, celui-ci sera appelé un processeur dual-core (deux coeurs), quad-core (4 coeurs), octo-core (8 coeurs), etc. Dans la grosse majorité des cas, les coeurs d'un processeur multicœurs sont tous identiques. Mais ce n'est certainement pas une obligation : on peut très bien mettre plusieurs processeurs assez différents sur la même puce, sans que cela ne pose problème. On peut très bien utiliser un cœur principal avec des coeurs plus spécialisés autour, par exemple. Cela s'appelle du **multicœurs asymétrique**. Ce terme est à opposer au **multicœurs symétrique**, dans lequel on place des processeurs identiques sur la même puce de silicium.

Le processeur CELL est un des exemples les plus récent de processeur multicœurs asymétrique. Vous connaissez sûrement ce fameux processeur, et vous en possédez peut-être un chez vous. Évidemment, il ne faut pas chercher dans l'unité centrale de votre PC de bureau, non. Il faut chercher dans votre console de jeux : et oui, votre PS3 contient un processeur CELL. Pour simplifier, notre processeur CELL peut être vu comme intégrant un cœur principal POWER PC version 5, qu'on retrouvait autrefois dans les Mac, et environ 8 processeurs auxiliaires. Le processeur principal est appelé le PPE et les processeurs auxiliaires sont les SPE. Les SPE sont reliés à une mémoire locale de 256 kibioctets, le Local Store, qui communique avec le processeur principal via un bus spécial. Les SPE communiquent avec la RAM principale via des contrôleurs DMA. Les SPE possèdent des instructions permettant de commander leur contrôleur DMA et que c'est le seul moyen qu'ils ont pour récupérer des informations depuis la mémoire. Et c'est au programmeur de gérer tout ça ! C'est le processeur principal qui va envoyer aux SPE les programmes qu'ils doivent exécuter. Il va déléguer des calculs à effectuer aux SPE. Pour cela, notre processeur principal va simplement écrire dans le local store du SPE, et va lui envoyer une demande lui ordonnant de commencer l'exécution du programme qu'il vient d'écrire.



Sur certains processeurs multicœurs, certains circuits sont partagés entre plusieurs coeurs. Cette technique consistant à ne pas dupliquer certains circuits et à en partager certains s'appelle le **cluster multithreading**. Cette technique est notamment utilisée sur les processeurs FX-8150 et FX-8120 d'AMD, et les autres processeurs basés sur l'architecture Bulldozer. Avec ces processeurs, tous les coeurs se partagent l'unité de calcul sur les nombres flottants (les nombres à virgule). Le partage de circuits permet d'éviter de dupliquer trop de circuits. Il est en effet évident qu'un seul circuit partagé entre tous les coeurs prendra moins de place et utilisera moins de composants électroniques que plusieurs circuits. Le problème est que ce partage est source de dépendances structurelles, ce qui peut entraîner des pertes de performances.

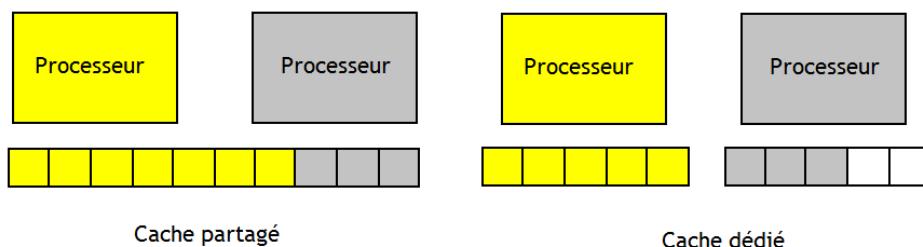
Quelque soit la technique de multicœur utilisée, les différents coeurs sont gérés par le système d'exploitation de l'ordinateur, avec l'aide d'interruptions inter-processeurs, des interruptions déclenchées sur un cœur et exécutées sur un autre. Pour générer des interruptions inter-processeur, le contrôleur d'interruption doit pouvoir rediriger des interruptions déclenchées par un processeur vers un autre. Les anciens PC incorporaient sur leur carte mère un contrôleur d'interruption créé par Intel, le 8259A, qui ne gérait pas les interruptions inter-processeurs. Pour gérer cette situation, les cartes mères multiprocesseurs devaient incorporer un contrôleur spécial en complément. De nos jours, chaque cœur x86 possède son propre contrôleur d'interruption, le local APIC, qui s'occupe de gérer les interruptions en provenance ou arrivant vers ce processeur. On trouve aussi un IO-APIC, qui s'occupe de gérer les interruptions en provenance des périphériques et de les redistribuer vers les local APIC. Ce IO-APIC s'occupe aussi de gérer les interruptions inter-processeurs en faisant passer les interruptions d'un local APIC vers un autre. Tous les local APIC et l'IO-APIC sont reliés ensemble par un bus APIC spécialisé, par lequel ils vont pouvoir communiquer et s'échanger des demandes d'interruptions.



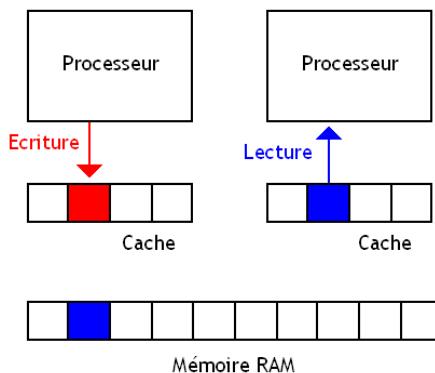
On peut préciser quel est le processeur de destination en configurant certains registres du IO-APIC, afin que celui-ci redirige la demande d'interruption d'un local APIC vers celui sélectionné. Cela se fait avec l'aide d'un registre de 64 bits nommé l'Interrupt Command Register. Pour simplifier le mécanisme complet, chaque processeur se voit attribuer un Id au démarrage qui permet de l'identifier (en fait, cet Id est attribué au local APIC de chaque processeur). Certains bits de ce registre permettent de préciser quel est le type de transfert : doit-on envoyer l'interruption au processeur émetteur, à tous les autres processeurs, à un processeur particulier. Dans le dernier cas, certains bits du registre permettent de préciser l'Id du processeur qui va devoir recevoir l'interruption. A charge de l'APIC de faire ce qu'il faut en fonction du contenu de ce registre.

Cohérence des caches

Quand on conçoit un processeur multicœurs, il ne faut pas oublier ce qui arrive à la pièce maîtresse de tout processeur actuel : le cache ! Vu que la quantité de cache est limitée, on peut se demander s'il vaut mieux un petit cache pour chaque processeur ou un gros cache partagé. Un **cache partagé** répartit le cache de manière optimale : un programme gourmand peut utiliser autant de cache qu'il veut, laissant juste ce qu'il faut à l'autre programme. Par contre, plusieurs programmes peuvent entrer en compétition pour le cache, que ce soit pour y placer des données ou pour les accès mémoire. De plus, les caches partagés ont un temps d'accès nettement plus élevé.



Les **caches dédiés** ont un problème, que je vais introduire par un exemple. Prenons deux processeurs qui ont chacun une copie d'une donnée dans leur cache. Si un processeur modifie sa copie de la donnée, l'autre ne sera pas mise à jour. L'autre processeur manipule donc une donnée périmée : il n'y a pas cohérence des caches.



Pour maintenir cette **cohérence des caches**, on a inventé de quoi détecter les données périmées et les mettre à jour : des protocoles de cohérence des caches. Tout protocole de cohérence des caches doit répondre ce problème : comment les autres caches sont-ils au courant qu'ils ont une donnée périmée ? Pour cela, il existe deux solutions : l'espionnage du bus et l'usage de répertoires. Certains protocoles utilisent un **dictionnaire**, une table de correspondance qui mémorise, pour chaque ligne de cache, toutes les informations nécessaires pour maintenir sa cohérence. Ces protocoles sont surtout utilisés sur les architectures distribuées : ils sont en effet difficiles à implémenter, tandis que leurs concurrents sont plus simples à planter sur les machines à mémoire partagée.

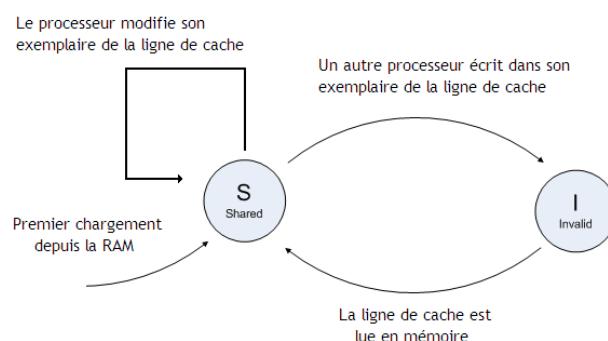
Avec l'**espionnage du bus**, les caches interceptent les écritures sur le bus (qu'elles proviennent ou non des autres processeurs), afin de détecter à jour la ligne de cache ou de la périmer. Avec ces protocoles, chaque ligne de cache contient des bits qui indiquent si la donnée contenue est à jour ou périmée. Quand on veut lire une donnée, les circuits chargés de lire dans le cache vont vérifier ces bits. Si ces bits indiquent que la ligne de cache est périmée, le processeur va lire les données depuis la RAM ou les autres caches et mettre à jour la ligne de cache. La mise à jour de ces bits de validité a lieu à chaque écriture (y compris les caches des autres coeurs). Plus précisément, nos caches sont interconnectés entre eux, afin de maintenir la cohérence. Si un cache contient une donnée partagée, ce cache devra prévenir tous les autres caches qu'une donnée a été mise à jour. La mise à jour des données périmées peut être automatique ou basée sur une invalidation. Avec la **mise à jour sur écriture**, les caches sont mis à jour automatiquement le plus tôt possible, avant même toute tentative de lecture. Avec l'**invalidation sur écriture**, les écritures invalident les versions de la donnée dans les autres caches. Ces versions seront mises à jour quand le processeur les lira : il détectera que la ligne de cache a été invalidée et la mettra à jour si besoin.

Protocole sans nouveaux états

Pour commencer, nous allons voir le protocole de cohérence des caches le plus simple : le protocole SI. Ce protocole ne fonctionne qu'avec un certain type de mémoires caches : les caches Write-through. Avec les caches Write-through, toute donnée écrite dans le cache est écrite en même temps dans la mémoire RAM et dans les niveaux de caches inférieurs s'ils existent. Le contenu de la mémoire est donc toujours à jour, mais ce n'est pas le cas pour les caches des autres processeurs.

Protocole SI

Dans ces conditions, seuls deux états suffisent pour décrire l'état d'une ligne de cache : Shared, qui indique que la ligne de cache est cohérente et Invalid, qui indique que la donnée est périmée. On obtient le protocole de cohérence des caches le plus simple qui soit : le protocole SI. Voici le fonctionnement de ce protocole :



Protocoles à état Shared

La cohérence des caches est très simple quand on a des caches write-through. Malheureusement, ces caches sont à l'origine de beaucoup d'écritures en mémoire qui saturent le bus. Aussi, d'autres caches ont été inventés, qui n'ont pas ce problème : les caches Write-back, où le contenu de la mémoire n'est pas cohérent avec le contenu du cache. Si on écrit dans la mémoire cache, le contenu de la mémoire RAM n'est pas mis à jour. On doit attendre que la donnée soit effacée du cache pour l'enregistrer en mémoire ou dans les niveaux de caches inférieurs (s'ils

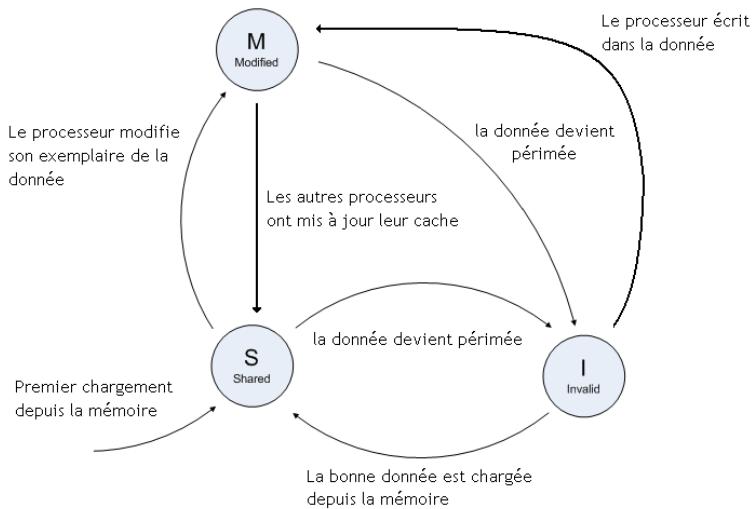
existent) : cela évite de nombreuses écritures mémoires inutiles.

Ces protocoles scindent l'état Shared en deux sous-états. Au final, ces protocoles à invalidation utilisent trois états pour une ligne de cache :

- Modified ;
- Shared ;
- Invalid.

L'état Invalid ne change pas et correspond toujours au cas où la donnée présente dans le cache est périmée. L'état Shared change et correspond maintenant à une donnée à jour, présente dans plusieurs caches. L'état Modified correspond à une donnée que le processeur a modifiée dans son cache : elle est à jour, mais les copies des autres caches sont périmées.

Divers protocoles de cohérence des caches existent pour les caches Write Back. Le plus simple d'entre eux est le protocole MSI. Avec ce protocole, un processeur peut lire des lignes de caches dans l'état Modified ou Shared sans problème, mais toute tentative de lecture d'une ligne Invalid demande de récupérer la version à jour dans la mémoire. Pour l'écriture, rien de plus simple : toute écriture est possible et place la ligne de cache dans l'état Modified.



Protocoles à état Exclusif

Le protocole MSI n'est pas parfait : si un seul cache possède une donnée, on aura prévenu les autres caches pour rien en cas d'écriture. Ces communications sur le bus ne sont pas gratuites et en faire trop peut ralentir fortement notre ordinateur. Pour régler ce problème, on a scindé l'état Shared en deux états :

- une donnée sera dans l'état Exclusive si les autres processeurs ne possèdent pas de copie de la donnée ;
- et sera dans l'état Shared sinon.

Avec cette distinction, on peut éviter l'envoi de messages aux autres caches (ou aux circuits chargés de gérer la cohérence des caches) si on écrit dans une donnée marquée Exclusive : on sait que les autres caches ne possèdent pas de copie de la donnée, alors il ne sert à rien de prévenir inutilement.

Comment le processeur fait-il pour savoir si les autres caches ont une copie de la donnée ? Pour cela, il faut ajouter un fil Shared sur le bus, qui sert à dire si un autre cache a une copie de la donnée. Lors de chaque lecture, l'adresse à lire sera envoyée à tous les caches, qui vérifieront s'ils possèdent une copie de la donnée. Une fois le résultat connu, chaque cache fournit un bit qui indique s'il a une copie de la donnée. Le bit Shared est obtenu en effectuant un OU logique entre toutes les versions du bit envoyé par les caches.

Le fonctionnement du protocole MESI est identique au protocole MSI, avec quelques ajouts. Par exemple, si une donnée est chargée depuis la mémoire pour la première fois dans un cache, elle passe soit en Exclusive (les autres caches ne contenaient pas la donnée), soit en Shared (les autres caches en possèdent une copie). Une donnée marquée Exclusive peut devenir Shared si la donnée est chargée dans le cache d'un autre processeur.

Protocole à état Owned

Les protocoles MESI et MSI ne permettent pas de transférer des données entre les caches sans passer par la mémoire. Si le processeur demande la copie valide d'une donnée, tous les caches ayant la bonne version de la donnée vont répondre en même temps : la donnée est envoyée en plusieurs exemplaires !

Pour éviter ce problème, on doit rajouter un état supplémentaire : l'état Owned. Si un processeur écrit dans son cache, il mettra sa donnée en Owned. Les autres caches passeront leur donnée en version Modified, voire Shared une fois la mémoire mise à jour : ainsi, un seul processeur pourra avoir une donnée dans l'état Owned. Seul le processeur qui possède la donnée en Owned va répondre aux demandes de mise à jour (ceux possédant la donnée Shared ne répondant pas à la demande formulée par le processeur demandeur).

Lors d'une lecture, le cache va vérifier si la lecture envoyée sur le bus correspond à une de ses données. Mais cette vérification va prendre du temps, et le processeur va devoir attendre un certain temps. Si au bout d'un certain temps, aucun cache n'a répondu, le processeur postule qu'aucun cache n'a la donnée demandée et va lire la donnée en mémoire. Ce temps est parfois fixé une fois pour toute lors de la création des processeurs, mais il peut aussi être variable, qui est géré comme suit :

- pour savoir si un cache contient une copie de la donnée demandée, chaque cache devra répondre en fournissant un bit ;
- quand le cache a terminé la vérification, il envoie un 1 sur une sortie spécifique, et un 0 sinon ;
- un ET logique est effectué entre tous les bits fournis par les différents caches, et le résultat final indique si tous les caches ont effectué leur vérification.

Atomicité

Afin de gérer le partage de la mémoire sans problèmes, chaque processeur doit définir un modèle mémoire, un ensemble de restrictions et de contraintes qui garantissent que les instructions ne puissent pas être interrompues (ou donnent l'impression de ne pas l'être) : c'est la propriété

d'atomicité. Un thread doit utiliser plusieurs instructions successives sur la donnée pour pouvoir en faire ce qu'il veut, et cela peut poser des problèmes : dans certaines situations, les instructions peuvent être interrompues par une exception ou tout autre chose. Par exemple, il est possible qu'une lecture démarre avant que la précédente soit terminée. De même, rien n'empêche une lecture de finir avant l'écriture précédente et renvoyer la valeur d'avant l'écriture.

Prenons un exemple, avec un entier entre plusieurs threads, chaque thread s'exécutant sur un processeur x86. Chaque thread veut l'incrémenter. Seul problème, l'incrémantation n'est pas effectuée en une seule instruction sur les processeurs x86. Il faut en effet lire la donnée, l'augmenter de 1, puis l'écrire. Et si deux processeurs veulent augmenter simultanément cette donnée, on court à la catastrophe. Chaque thread peut être interrompu à n'importe quel moment par un autre processeur qui voudra modifier sa donnée. Les instructions de nos threads s'exécuteront en série, mais le processeur peut parfaitement être dérangé par un autre processeur entre deux instructions. Dans notre exemple, une telle situation est illustrée ci-dessous. On a donc une valeur de départ de 5, qui est augmentée de 1 deux fois, ce qui donne au final 6.

Processeur 1	Processeur 2	Valeur de la donnée partagée
Lecture de la donnée 5		
Addition 5 + 1 = 6	Lecture de la donnée 5	5
Ecriture du résultat 6	Addition 5 + 1 = 6	6
	Ecriture du résultat 6	6

Temps

Pour avoir le bon résultat il y a une seule et unique solution : le processeur qui accède à la donnée doit avoir un accès exclusif à la donnée partagée. Sans cela, l'autre processeur ira lire une version de la donnée qui n'aura pas encore été modifiée par l'autre processeur. Dans notre exemple, un seul thread doit pouvoir manipuler notre compteur à la fois. Et bien sûr, cette réponse peut, et doit se généraliser à presque toutes les autres situations impliquant une donnée partagée. Chaque thread doit donc avoir un accès exclusif à notre donnée partagée, sans qu'aucun autre thread ne puisse manipuler notre donnée. On doit donc définir ce qu'on appelle une **section critique** : un morceau de temps durant lequel un thread aura un accès exclusif à une donnée partagée : notre thread est certain qu'aucun autre thread n'ira modifier la donnée qu'il manipule durant ce temps. Autant prévenir tout de suite : créer de telles sections critiques se base sur des mécanismes mêlant le matériel et le logiciel. Il existe deux grandes solutions, qui peuvent être soit codées sous la forme de programmes, soit implantées directement dans le silicium de nos processeurs.

Voyons la première de ces solutions : l'**exclusion mutuelle**. Avec celle-ci, on fait en sorte qu'un thread puisse réserver la donnée partagée. Un thread qui veut manipuler cette donnée va donc attendre qu'elle soit libre pour la réserver afin de l'utiliser, et la libérera une fois qu'il en a fini avec elle. Si la donnée est occupée par un thread, tous les autres threads devront attendre leur tour. Pour mettre en œuvre cette réservation/dé-réservation, on va devoir ajouter un compteur à la donnée partagée, qui indique si la donnée partagée est libre ou déjà réservée. Dans le cas le plus simple, ce compteur vaudra 0 si la donnée est réservée, et 1 si elle est libre. Ainsi, un thread qui voudra réserver la donnée va d'abord devoir vérifier si ce nombre est à 1 avant de pouvoir réserver sa donnée. Si c'est le cas, il réservera la donnée en passant ce nombre à 0. Si la donnée est réservée par un autre thread, il devra tout simplement attendre son tour. On a alors créé ce qu'on appelle un verrou d'exclusion mutuelle. Seul problème : cette vérification et modification du compteur pose problème. Celle-ci ne peut pas être interrompue, sous peine de problèmes. On peut reprendre l'exemple du dessus pour l'illustrer. Si notre compteur est à 0, et que deux threads veulent lire et modifier ce compteur simultanément, il est possible que les deux threads lisent le compteur en même temps : ils liront alors un zéro, et essayeront alors de se réserver la donnée simultanément. Bref, retour à la case départ...

Instructions atomiques

Cette vérification et modification du compteur se fait en plusieurs étapes : une lecture du compteur, puis une écriture si le compteur est à la bonne valeur. Il faudrait que cette lecture et l'écriture se fassent en une seule fois. Pour régler ce problème, certains processeurs fournissent des instructions spécialisées, in-interruptibles, capables d'effectuer cette modification du compteur en une seule fois. Ces instructions peuvent ainsi lire ce compteur, décider si on peut le modifier, et écrire la bonne valeur sans être dérangé par un autre processeur qui viendrait s'inviter dans la mémoire sans autorisation ! Par exemple, sur les processeurs x86, la vérification/modification du compteur vue plus haut peut se faire avec l'instruction test and set. D'autres instructions atomiques similaires existent pour résoudre ce genre de problèmes. Leur rôle est toujours d'implémenter des verrous d'exclusion mutuelle plus ou moins sophistiqués, en permettant d'effectuer une lecture, suivie d'une écriture en une seule fois. Ces instructions permettent ainsi de créer des sémaphores, des Locks, etc. Généralement, un programmeur n'a pas à devoir manipuler des instructions atomiques lui-même, mais ne fait que manipuler des abstractions basées sur ces instructions atomiques, fournies par des bibliothèques ou son langage de programmation.

Ces instructions in-interruptibles sont appelées des **instructions atomiques**. De plus, elles empêchent tout accès mémoire tant qu'elles ne sont pas terminées. De telles instructions garantissent que les écritures et lectures s'exécutent l'une après l'autre. L'instruction atomique va réserver l'accès au bus en configurant un bit du bus mémoire, ou par d'autres mécanismes de synchronisation entre processeurs. Si la donnée est dans le cache, pas besoin de bloquer la mémoire : le processeur a juste à écrire dans la mémoire cache et les mécanismes de cohérence des caches se contenteront de mettre à jour la donnée de façon atomique. Voici la plupart de ces instructions atomiques les plus connues :

Compare And Swap	Cette instruction va lire une donnée en mémoire, va comparer celle-ci à l'opérande de notre instruction (une donnée fournie par l'instruction), et va écrire un résultat en mémoire si ces deux valeurs sont différentes. Ce fameux résultat est fourni par l'instruction, ou est stocké dans un registre du processeur.
Fetch And Add	Cette instruction charge la valeur de notre compteur depuis la mémoire, l'incrémente, et écrit sa valeur en une seule fois. Elle permet de réaliser ce qu'on appelle des sémaphores. Elle permet aussi d'implémenter des compteurs concurrents.
XCHG	Cette instruction peut échanger le contenu d'un registre et d'un morceau de mémoire de façon atomique. Elle est notamment utilisée sur les processeurs x86 de nos PC, qui implémentent cette instruction.

Comme je l'ai dit plus haut, ces instructions empêchent tout autre processeur d'aller lire ou modifier la donnée qu'elles sont en train de modifier. Ainsi, lors de l'exécution de l'instruction atomique, aucun processeur ne peut aller manipuler la mémoire : notre instruction atomique va alors bloquer la mémoire et en réserver l'accès au bus mémoire rien que pour elle. Cela peut se faire en envoyant un signal sur le bus mémoire, ou pas d'autres mécanismes de synchronisation entre processeurs. Quoiqu'il en soit, le cout de ce blocage de la mémoire est assez lourd : cela peut prendre un sacré bout de temps, ce qui fait que nos instructions atomiques sont lentes. Du moins, c'est le cas si la donnée est en mémoire et que le processeur est un peu stupide. En effet, sur certains processeurs, on peut optimiser le tout dans le cas où la donnée est dans le cache. Dans ce cas, pas besoin de bloquer la mémoire : le processeur a juste à écrire dans la mémoire cache, et les mécanismes de cohérence des caches se contenteront de mettre à jour la donnée de façon atomique automatiquement. Le cout des instructions atomiques est alors fortement amorti.

Instructions LL/SC

Une autre technique de synchronisation est basée sur deux instructions : **Load-Link** et **Store-Conditional**. L'instruction Load-Link va lire une donnée depuis la mémoire de façon atomique. L'instruction Store-Conditional écrit une donnée chargée avec Load-Link, mais uniquement à condition que la copie en mémoire n'aie pas été modifiée entre temps : si ce n'est pas le cas, Store-conditional échoue. Pour indiquer un échec, il y a plusieurs solutions : soit elle met un bit du registre d'état à 1, soit elle écrit une valeur de retour dans un registre. Sur certains processeurs, l'exécution d'interruptions ou d'exceptions matérielles après un Load-Link fait échouer un Store-conditional ultérieur. Implémenter ces deux instructions est assez simple, et peut se faire en utilisant les techniques de bus-snapping vues dans le chapitre sur la cohérence des caches. Pour implémenter l'instruction SC, il suffit de mémoriser si la donnée lue par l'instruction LL a été invalidée depuis la dernière instruction LL. Pour cela, on utilise un registre qui mémorise l'adresse lue par l'instruction LL, à laquelle on ajoute un bit d'invalidation qui dit si cette adresse a été invalidée. L'instruction LL va initialiser le registre d'adresse avec l'adresse lue, et le bit est mis à zéro. Une écriture a lieu sur le bus, des circuits vérifient si l'adresse écrite est identique à celle contenue dans le registre d'adresse et mettent à jour le bit d'invalidation. L'instruction SC doit vérifier ce bit avant d'autoriser l'écriture.

Mémoire Transactionnelle Matérielle

Pour résoudre les différents problèmes posés par les verrous d'exclusion mutuelle, les chercheurs ont inventé la **mémoire transactionnelle**. La mémoire transactionnelle permet de rendre atomiques des morceaux de programmes complets, que l'on appelle des transactions. Pendant qu'une transaction s'exécute, tout se passe comme si la transaction ne modifiait pas notre donnée et reste plus ou moins "invisible" des autres processeurs. Dans le cas où la donnée partagée n'a pas été modifiée par un autre processeur durant l'exécution d'une transaction, celle-ci peut alors écrire son résultat en mémoire. Mais dans le cas contraire, la transaction échoue et doit repartir depuis le début : les changements effectués par la transaction ne sont pas pris en compte.

Cette mémoire transactionnelle peut être implantée en matériel par diverses techniques. Avec ces techniques, les écritures dans le cache ne sont pas propagées tant que la transaction ne termine pas correctement. Et si la transaction échoue, les données modifiées par la transaction sont marquées Invalid par les mécanismes de cohérence des caches. De plus, les changements effectués par la transaction sur les registres doivent être annulés si celle-ci échoue : on peut sauvegarder une copie des registres du processeur au démarrage de la transaction, ou annuler les changements effectués par la transaction (ROB ou autre méthode capable de gérer les interruptions précises).

Il existe plusieurs manières d'implémenter la mémoire transactionnelle matérielle. La plus simple utilise un cache dédié aux transactions : le cache transactionnel. A cela, il faut ajouter des instructions pour manipuler le cache transactionnel. Les données dans le cache transactionnel utilisent un protocole légèrement différent des autres caches, avec des états et des transitions en plus. Les autres méthodes préfèrent réutiliser les mémoires caches déjà présentes dans le processeur en guise de cache transactionnel. Une solution, utilisée sur le processeur Blue gene d'IBM, consiste à avoir plusieurs exemplaires d'une donnée dans le cache. Si un seul processeur a manipulé la donnée partagée, celle-ci ne sera présente qu'en une seule version dans les caches des autres processeurs. Mais si la transaction échoue, alors cela veut dire que plusieurs processeurs ont modifié cette donnée : plusieurs versions d'une même donnée différente sera présente dans le cache.

Ces techniques ont un défaut : la quantité de données manipulées par la transaction est limitée à la taille du cache. Pour éviter ce petit problème, certains chercheurs travaillent sur une mémoire transactionnelle capable de gérer des transactions de taille arbitraires. Ces techniques mémorisent les données modifiées par la transaction en mémoire RAM, dans des enregistrements que l'on appelle des logs.

Speculative Lock Elision

Les instructions atomiques sont lentes, sans compter qu'elles sont utilisées de façon pessimistes : l'atomicité est garantie même si aucun autre thread n'accède à la donnée lue/écrite. Aussi, pour accélérer l'exécution des instructions atomiques, des chercheurs se sont penchés sur ce problème de réservations inutiles et ont trouvé une solution assez sympathique, basée sur la mémoire transactionnelle. Au lieu de devoir mettre un verrou et de réservé notre donnée juste au cas où, on peut agir d'une façon un peu plus optimiste. Rien n'empêche de transformer nos instructions atomiques servant pour les réservations en instructions permettant de démarrer des transactions. Bien évidemment, les instructions atomiques servant à libérer la donnée partagée vont marquer la fin de notre transaction. Ce mécanisme tente donc de se passer des instructions atomiques en les transformant en transaction une première fois, puis revient à la normale en cas d'échec. Il s'appelle le **Speculative Lock Elision**.

Ainsi, on n'exécute pas l'instruction atomique permettant d'effectuer une réservation, et on la transforme en une simple instruction de lecture de la donnée partagée. Une fois cela fait, on commence à exécuter la suite du programme en faisant en sorte que les autres processeurs ne voient pas les modifications effectuées sur nos données partagées. Puis, vient le moment de libérer la donnée partagée via une autre instruction atomique. A ce moment, si aucun autre thread n'a écrit dans notre donnée partagée, tout ira pour le mieux : on pourra rendre permanents les changements effectués. Par contre, si jamais un autre thread se permet d'aller écrire dans notre donnée partagée, on doit annuler les changements faits. A la suite de l'échec de cette exécution optimiste, cette transaction cachée, le processeur reprendra son exécution au début de notre fausse transaction, et va exécuter son programme normalement : le processeur effectuera alors de vraies instructions atomiques, au lieu de les interpréter comme des débuts de transactions.

Il arrive parfois que le premier essai échoue lamentablement : si un autre thread a beaucoup de chance de manipuler une donnée partagée en même temps qu'un autre, le premier essai a de fortes chances de planter. Pour plus efficacité, certains processeurs cherchent parfois à éviter ce genre de situation en estimant la probabilité que le premier essai (la transaction) échoue. Pour cela, ils incorporent un circuit permettant d'évaluer les chances que notre premier marche en tant que transaction : le Transaction Predictor. Une fois cette situation connue, le processeur décide ou non d'exécuter ce premier essai en tant que transaction.

L'exemple avec le x86

Autre exemple, on peut citer les processeurs Intel récents. Je pense notamment aux processeurs basés sur l'architecture Haswell. Au alentours de mars 2013, de nouveaux processeurs Intel sortiront sur le marché : ce seront les premiers processeurs grand public qui supporteront la mémoire transactionnelle matérielle. Attardons-nous un peu sur ces processeurs, et sur l'implémentation de la mémoire transactionnelle matérielle de ces processeurs. Sur ces processeurs, deux modes sont disponibles pour la mémoire transactionnelle matérielle : le mode TSX, et le mode HLE.

Le mode TSX correspond simplement à quelques instructions supplémentaires permettant de gérer la mémoire transactionnelle matérielle. On trouve ainsi trois nouvelles instructions : XBEGIN, XEND et XABORT. XBEGIN sert en quelque sorte de top départ : elle sert à démarrer une transaction. Toutes les instructions placées après elles dans l'ordre du programme seront ainsi dans une transaction. Au cas où la transaction échoue, il est intéressant de laisser le programmeur quoi faire. Pour cela, l'instruction XBEGIN permet au programmeur de spécifier une adresse. Cette adresse permet de pointer sur un morceau de code permettant de gérer l'échec de la transaction. En cas d'échec de la transaction, notre processeur va reprendre automatiquement son exécution à cette adresse. Évidemment, si on a de quoi marquer le début d'une transaction, il faut aussi indiquer sa fin. Pour cela, on utilise l'instruction XEND. XABORT quand à elle, va servir à stopper l'exécution d'une transaction : elle sert à faire planter notre transaction si jamais on s'aperçoit d'un problème lors de l'exécution de notre transaction. Lors de la fin d'une transaction, le processeur va automatiquement reprendre à l'adresse indiquée par XBEGIN, et va remettre le processeur dans l'état dans lequel il était avant le début de la transaction : les registres modifiés par la transaction sont remis dans leur état initial, à une exception près : EAX. Celui-ci sert à stocker un code d'erreur qui indique les raisons de l'échec d'une transaction. Cela permet de donner des informations au code de gestion d'échec de transaction, qui peut alors gérer la situation plus finement. HLE

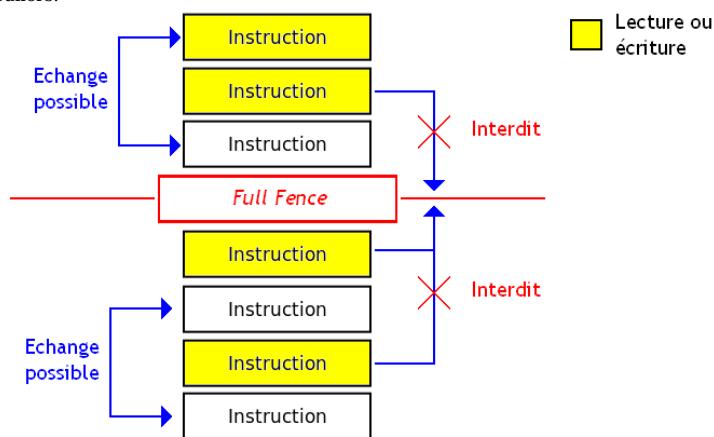
Les processeurs Haswell supportent aussi le Lock Elision. Les instructions atomiques peuvent ainsi supporter l'exécution en tant que transaction à une condition : qu'on leur rajoute un préfixe. Le préfixe, pour les instructions x86, correspond simplement à un octet optionnel, placé au début de notre instruction dans la mémoire. Cet octet servira à donner des indications au processeur, qui permettront de modifier le comportement de notre instruction. Par exemple, les processeurs x86 supportent pas mal d'octets de préfixe : LOCK, qui permet de rendre certaines instructions atomiques, ou REPNZE, qui permet de répéter certaines instructions tant qu'une condition est requise. Le fait est que certains préfixes n'ont pas de signification pour certaines instructions : les placer devant ces instructions n'a alors pas de sens. Autrefois, ils étaient totalement ignorés, et le processeur ne tenait pas compte de ces préfixes sans signification. Pour supporter le Lock Elision, ces préfixes sans significations sont réutilisés histoire de dire au processeur : cet instruction atomique doit subir la Lock Elision et doit être tentée en tant que transaction. Deux "nouveaux"

préfixes font leur apparition : XAQUIRE qui sert à indiquer que notre instruction atomique doit être tentée en tant que transaction ; et XRELEASE qui dit que la transaction spéculative est terminée. Ainsi, un programme peut être conçu pour utiliser la Lock Elision, tout en fonctionnant sur des processeurs plus anciens, qui ne la supportent pas ! Belle tentative de garder la rétrocompatibilité.

Consistance mémoire

On a vu dans les chapitres précédents que les processeurs modernes modifient l'ordre des accès mémoire pour gagner en efficacité. Avec plusieurs processeurs, chaque processeur fait cela chacun dans son coin sans se préoccuper des autres. Les opérations d'écriture peuvent être mises dans le désordre du point de vue des autres processeurs, et les lectures effectuées par les autres processeurs peuvent alors renvoyer de vieilles données. Pour que les accès mémoires dans le désordre ne posent pas de problèmes, il existe différentes solutions.

Modèle de consistance	Description
Consistance séquentielle	Aucune réorganisation de l'ordre des instructions n'est possible. Tout se passe comme si les instructions de chaque programme s'effectuaient dans l'ordre imposé par celui-ci, et que les lectures ou écritures des différents processeurs étaient exécutées les unes après les autres. Ce modèle de consistance interdit de nombreux cas de réorganisation parfaitement légaux, même avec plusieurs processeurs.
Total Store Ordering	Un processeur peut immédiatement réutiliser une donnée qu'il vient d'écrire dans son cache. Tout se passe donc comme si l'écriture en question n'était pas in-interruptible : on peut lire la donnée écrite avant que toutes les versions de la donnée soient mises à jour. C'est ce qui était fait dans les protocoles de cohérence des caches qu'on a vus précédemment : le statut modified n'empêchait pas les lectures de la donnée.
Partial Store Ordering	Permet des écritures simultanées, ainsi que des réorganisations de l'ordre des écritures qui portent sur des données différentes. Ainsi, on peut démarrer une nouvelle écriture avant que les écritures précédentes ne soient terminées. La seule condition pour éviter les catastrophes est que ces écritures manipulent des données différentes, placées à des endroits différents de la mémoire.
No Store Ordering	Toutes les réorganisations possibles entre lectures et écritures sont autorisées, tant qu'elle se font sur des données différentes. On peut parfaitement démarrer une lecture pendant que d'autres sont en attente ou en cours. Ces processeurs garantissent que les accès à des données partagées se déroulent correctement avec des instructions que doivent utiliser les compilateurs ou les programmeurs : des Memory Barriers, ou Fences . Ce sont des instructions qui vont forcer le processeur à terminer toutes les écritures et/ou lectures qui la précède. Pour plus d'efficacité, certains processeurs possèdent deux Fences : une pour les lectures, et une autre pour les écritures. Certains processeurs utilisent un nombre de Fences plus élevé, et peuvent ainsi avoir des fences dédiées à des cas particuliers.



Le placement des fences dans un programme est géré par un subtil équilibre entre programmeur et compilateur. Généralement, un compilateur peut placer ces Fences au bon endroit, avec un peu d'aide du programmeur. Certains langages de programmation permettent d'indiquer au compilateur qu'une donnée doit toujours être lue depuis la mémoire RAM, via le mot-clé volatile. C'est très utile pour préciser que cette donnée est potentiellement partagée par plusieurs processeurs ou manipulable par des périphériques. Les compilateurs peuvent placer des Fences lors des lectures ou écritures sur ces variables, mais ce n'est pas une obligation. Il arrive aussi que le programmeur doive manipuler explicitement des Fences. Utiliser l'assembleur est alors une possibilité, mais qui est rarement exploitée, pour des raisons de portabilité. Pour limiter la casse, certains systèmes d'exploitations ou compilateurs peuvent aussi fournir des Fences explicites, encapsulées dans des bibliothèques ou cachées dans certaines fonctions.

Après avoir vu la théorie, passons maintenant à la pratique. Dans cette partie, on va voir les modèles de consistances utilisés sur les processeurs x86, ceux qu'on retrouve dans nos PC actuels. Le modèle de consistance des processeurs x86 a varié au cours de l'existence de l'architecture : un vulgaire 486DX n'a pas le même modèle de consistance qu'un Core 2 duo, par exemple. Quoiqu'il en soit, les modèles de consistance des processeurs x86 ont toujours été assez forts, avec pas mal de restrictions. Si on compare aux autres processeurs, le x86 est assez strict. Bref, le premier modèle de consistance utilisé sur les processeurs x86 est apparu sur les premiers processeurs x86 et est resté en place sur tous les processeurs de marque Pentium. Ce modèle est assez simple : hormis une exception, tout doit se passer comme si le processeur accédait à la mémoire dans l'ordre des opérations de notre programme. Cette exception concerne les lectures : dans certains cas, on peut les exécuter avant certaines écritures, sous réserve que les conditions suivantes soient respectées :

- ces écritures doivent se faire dans la mémoire cache ;
- la lecture doit se faire dans la mémoire ;
- nos écritures doivent aller écrire à des adresses différentes de l'adresse accédée en lecture ;
- aucune transaction avec un périphérique ne doit être en cours.

Sur les processeurs à partir du Pentium 4, les choses changent. Le Pentium 4 est en effet le premier processeur à implémenter des techniques permettant d'exécuter plusieurs processus en parallèle. Ce processeur est en effet le premier à utiliser l'Hyperthreading. En conséquence, le modèle de consistance a du être assoupli pour éviter de perdre bêtement en performance. Voici un résumé de ce modèle de consistance :

- une lecture ne peut pas être déplacée avant ou après une autre lecture ;
- une écriture ne peut pas être déplacée avant une lecture ;
- à part pour quelques exceptions, l'ordre des écritures dans la mémoire ne change pas : une écriture dans la mémoire ne peut pas être déplacée avant ou après une autre écriture ;
- on ne peut pas déplacer une écriture ou une lecture avant ou après une instruction atomique, ainsi que quelques instructions supplémentaires (celles qui accèdent aux périphériques ou aux entrées-sorties, notamment) ;
- des lectures peuvent être déplacées avant des écritures, si ces écritures et la lecture ne se font pas au même endroit, à la même adresse.

Dans cette liste, j'ai mentionné le fait que les écritures en mémoire peuvent changer dans certains cas exceptionnels. Ces cas exceptionnels sont les écritures effectuées par les instructions de gestion de tableaux et de chaînes de caractères, ainsi que certaines instructions SSE. Ces instructions SSE sont les instructions qui permettent d'écrire des données sans passer par le cache, mentionnées il y a quelques chapitres. Ce sont donc les instructions MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, MOVNTPD. Mais ce ne sont pas les seules : le x86 possède quelques instructions permettant de travailler directement sur des chaînes de caractères ou des tableaux : ce sont les instruction REP MOVSD, REPSCACB, et bien d'autres encore. Et bien sachez que les écritures effectuées dans ces instructions peuvent se faire dans un désordre complet (ou presque).

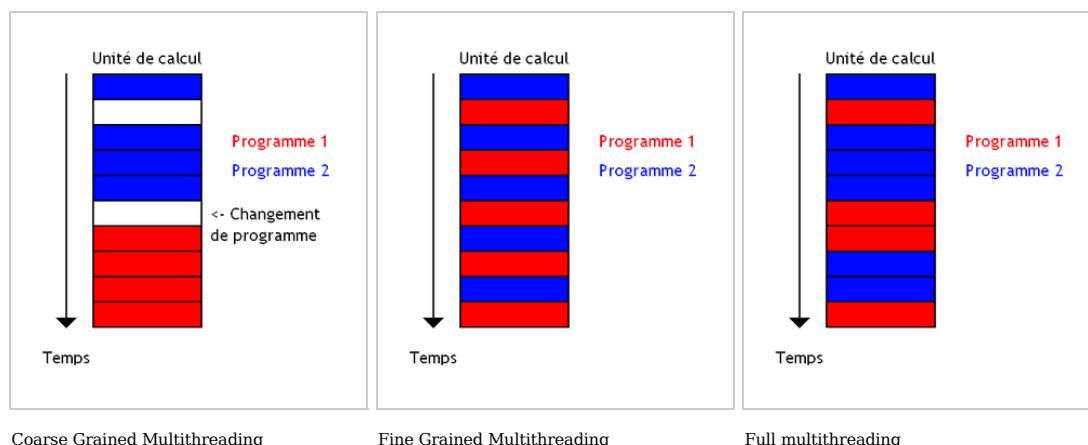
Architectures multithreadées et Hyperthreading

Vous pensez sûrement qu'il faut obligatoirement plusieurs processeurs pour exécuter plusieurs programmes en parallèle, mais sachez que c'est faux ! Il est parfois possible pour un processeur seul d'exécuter plusieurs programmes en même temps. Pour cela, il faut utiliser des processeurs spéciaux, qui utilisent des ruses de sioux.

Du parallélisme avec un seul processeur

A l'intérieur d'un processeur, on trouve un circuit qui fait des calculs : l'unité de calcul. Il arrive que l'unité de calcul d'un processeur ne fasse rien, pour diverses raisons. Et si on n'arrive pas à donner du travail à l'unité de calcul avec un seul programme, pourquoi ne pas remplir les vides avec les instructions d'un autre programme ? Et bien cela peut se faire de trois manières différentes.

- Première technique : le **Coarse Grained Multithreading**, qui change de programme quand un événement bien précis a lieu. Sur certains processeurs, on utilise une instruction de changement de programme, fournie par le jeu d'instruction du processeur. Mais certains processeurs décident tout seuls quand changer de programme. Suivant le processeur, les événements faisant changer de programme ne sont pas forcément les mêmes. Sur certains processeurs, on change de programme lorsque certaines instructions sont exécutées : accès à la mémoire, branchements, etc. Généralement, ces processeurs changent de programme à exécuter lorsqu'on doit accéder à la mémoire (lors d'un cache miss). Il faut dire que l'accès à la mémoire est quelque chose de très lent, aussi changer de programme et exécuter des instructions pour recouvrir l'accès à la mémoire est une bonne chose.
- La seconde méthode, le **Fine Grained Multithreading**, change de programme à chaque cycle d'horloge. Vu que deux instructions successives n'appartiennent pas au même programme, il n'y a pas de dépendances entre instructions successives. En conséquence, tous les circuits liés à la détection ou la prise en compte de ces dépendances disparaissent. Mais cela, on est obligé d'avoir autant de programmes en cours d'exécution qu'il y a d'étages de pipeline. Et quand ce n'est pas le cas, on trouver des solutions pour limiter la casse. Par exemple, certains processeurs lancent plusieurs instructions d'un même programme à la suite, chaque instruction contenant quelques bits pour dire au processeur : tu peux lancer 1, 2, 3 instructions à la suite sans problème. Cette technique s'appelle l'**anticipation de dépendances**.
- D'autres processeurs prennent le problème à bras le corps : plusieurs programmes s'exécutent en même temps, et chaque programme utilise l'unité de calcul dès que les autres la laissent libre. On parle alors de **multithreading total**.



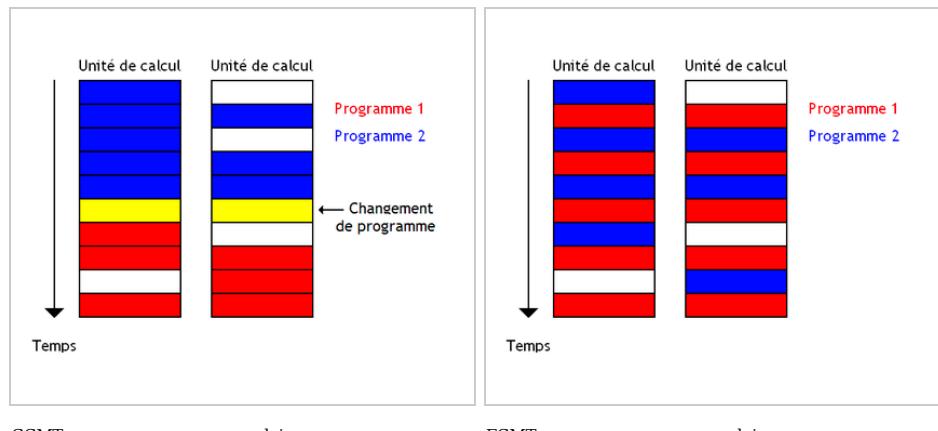
Coarse Grained Multithreading

Fine Grained Multithreading

Full multithreading

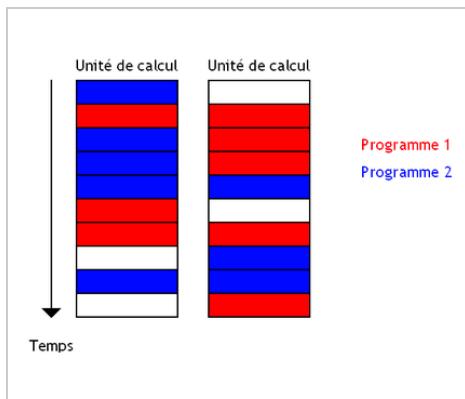
Simultaneous Multithreading

Les techniques vues au-dessus peuvent s'adapter sur les processeurs capables de démarrer plusieurs instructions simultanément sur des unités de calcul séparées. On peut ainsi adapter le Coarse-grained multithreading et le Fine-grained multithreading. Dans les deux cas, un seul programme a accès aux unités de calcul à chaque cycle d'horloge. Mais on peut aussi faire en sorte que plusieurs programmes puissent faire démarrer leurs instructions en même temps. A chaque cycle d'horloge, le processeur peut exécuter des instructions provenant de divers programmes. On obtient la technique du **Simultaneous Multi-Threading** ou SMT.



CGMT sur processeur superscalaire

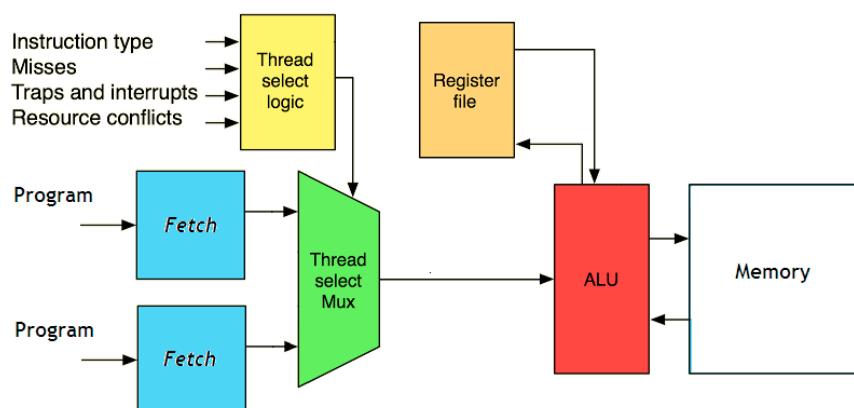
FGMT sur processeur superscalaire



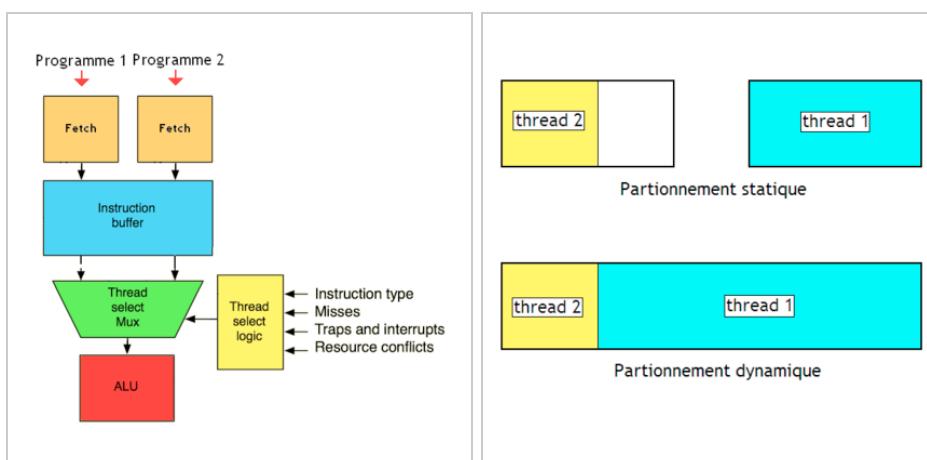
Simultaneous Multi-Threading

Et dans le processeur ?

Il est obligatoire de dupliquer les registres pour que chaque programme puisse avoir son ensemble de registres rien qu'à lui, ce qui demande d'utiliser soit un banc de registre (register file) par programme, soit un banc de registre commun géré par fenêtrage de registre (chaque programme ayant sa propre fenêtre de registres). De plus, les processeurs permettant d'exécuter plusieurs programmes utilisent le fait que l'unité de Fetch et l'unité de calcul sont séparés et fonctionnent en pipeline. Pour simplifier, le processeur peut exécuter une instruction sur l'unité de calcul et commencer à charger l'instruction suivante, en même temps. Le processeur doit charger des instructions en provenance de programmes différents, ce qui demande d'utiliser plusieurs Program Counter et éventuellement plusieurs unités de Fetch si la mémoire ou le cache d'instruction sont multiports. Dans le cas où il n'y a qu'une seule unité de Fetch, un seul programme peut charger ses instructions à chaque cycle. L'unité de Fetch donne la main à chacun des programmes les uns après les autres, en boucle. Un circuit se charge de choisir le Program Counter - le thread - qui aura la chance de charger ses instructions. On peut par exemple donner la priorité aux threads qui accèdent le moins à la mémoire RAM (peu de cache miss), à ceux qui ont exécuté le moins d'instructions de branchement récemment, etc.



Généralement, la fenêtre d'instruction (instruction windows) est partagée entre les différents programmes. Celle-ci est partitionnée en morceaux réservés à un programme bien précis. Ces morceaux ont la même taille : on dit que le partitionnement est statique. Dans d'autres cas, si jamais un programme a besoin de plus de place que l'autre, il peut réserver un peu plus de place que son concurrent : le partitionnement est dit dynamique. Chaque instruction mise en attente (chaque entrée) va stocker le numéro du programme, attribué à l'instruction par l'unité de Fetch.

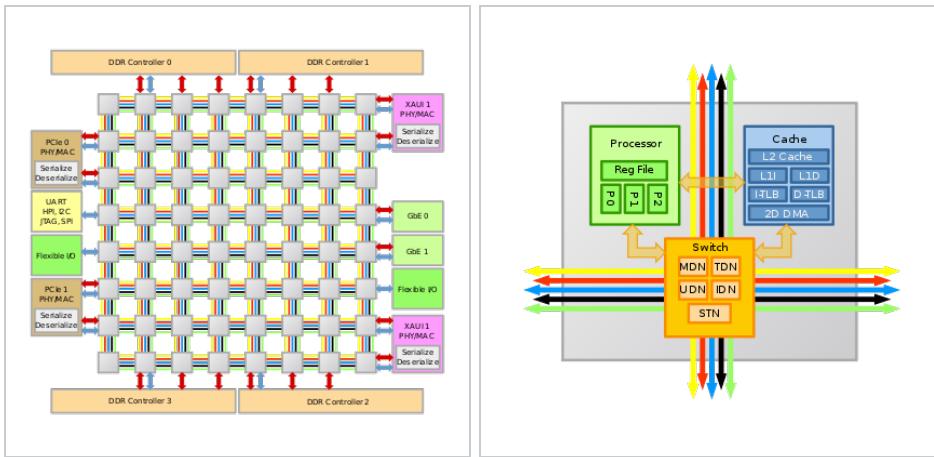


Instruction buffer et SMT.

Partitionnement de l'Instruction Buffer avec l'hyperthreading.

Architectures distribuées, NUMA et COMA

Les architectures distribuées sont des architectures qui utilisent non pas une mémoire partagée entre plusieurs processeurs, mais des mémoires séparées pour chaque processeur. Tous les processeurs sont reliés entre eux via un réseau, qui leur sert à échanger des données ou des ordres un peu particuliers. Dans la plupart des cas, les processeurs et mémoires sont localisées dans des ordinateurs séparés, reliés entre eux par un réseau local : on parle alors de **clusters**. De tels clusters sont monnaie courante dans les serveurs. Le **cloud computing** permet de relier entre des ordinateurs via internet, de manière à paralléliser les calculs dessus. Mais il est aussi possible de placer les processeurs et mémoires dans un même boîtier, sur la même puce de silicium. On parle alors de **d'architectures à tiles**. Ce nom vient du fait que chaque groupe processeur+RA+connexions réseau est appelé une tile. Un bon exemple serait l'architecture Tilera.



Architecture Tile64 du Tilera.

Tile de base du Tile64.

Types d'architectures distribuées

Il existe plusieurs types d'architectures distribuées selon la manière dont les processeurs gèrent les communications entre mémoires locales. On distingue ainsi :

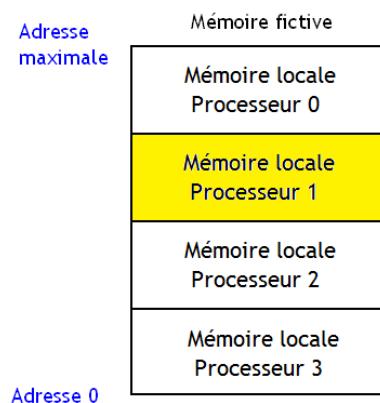
- les architectures à passage de mémoire ;
- les architectures Non Uniform Memory Access ;
- les architectures Cache Only Memory Access.

Architectures à passage de messages

Sur les **architectures à passage de messages**, chaque mémoire est dédiée à un processeur. Les processeurs peuvent accéder à la mémoire d'un autre via le réseau local. Il va de soi que les communications entre les différents processeurs peuvent prendre un temps relativement long, et que ceux-ci sont loin d'être négligeables. Avec une organisation de ce genre, la qualité et les performances du réseau reliant les ordinateurs est très important pour les performances.

NUMA

La seconde classe d'architectures est celle des **architectures NUMA**. Comme avec les architectures distribuées, les processeurs possèdent chacun une mémoire locale réservée. Mais il y a une différence : chaque processeur voit toutes les mémoires locales rassemblées dans une seule grosse mémoire unique, un espace d'adressage global.



Lors des lectures ou écritures, les adresses utilisées seront des adresses dans l'espace d'adressage global, dans la grosse mémoire. Si l'adresse est dans la mémoire locale du processeur, l'accès est direct. Dans le cas contraire, le système d'exploitation prend la relève et prépare une communication sur le réseau. Bien tirer partie de ces architectures demande de placer le plus possible les données en mémoire locale : l'idée est d'éviter les accès au réseau, qui sont lents.

Architectures COMA

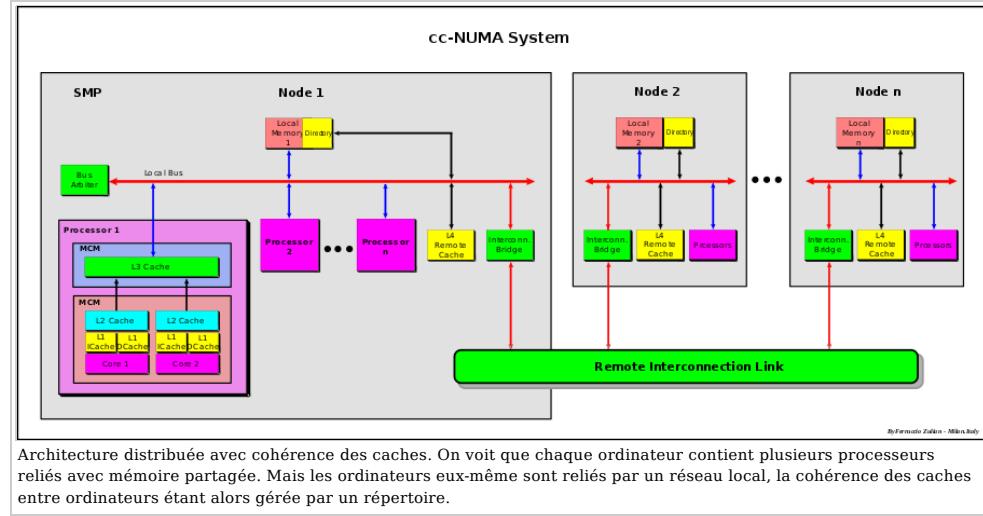
Sur les architectures NUMA, migrer les données à la main peut nuire aux performances : il se peut que des migrations inutiles soient faites, le processeur n'ayant pas besoin des données transférées. Pour éviter ces copies inutiles, le meilleur moyen est de les faire à la demande : quand un processeur a besoin d'un bloc de mémoire (d'une donnée), celui-ci est migré dans sa mémoire locale. En faisant ainsi, chaque mémoire locale se comporte comme un cache, mais sans qu'il y ait de mémoire principale. On obtient alors une **architecture COMA**.

Ces architectures font cependant face à quelques problèmes. Premièrement, il faut pouvoir localiser la mémoire qui contient le bloc demandé à partir de l'adresse de ce bloc. Ensuite, on doit gérer la migration de blocs dans une mémoire locale pleine. Sur les architectures Simple-COMA, ces opérations sont déléguées au système d'exploitation. Seules les architectures Flat-COMA et Hierarchical-COMA effectuent ces opérations en matériel, en modifiant les circuits de cohérence des caches. La différence entre ces deux architectures tient dans le réseau d'interconnexion utilisé pour relier les processeurs.

Cohérence des caches à base de répertoires

Sur ces architectures, les protocoles à base d'espionnage de bus deviennent inefficaces : on doit utiliser d'autres méthodes. Une première méthode, qui fonctionne assez mal, est d'utiliser une hiérarchie de bus : certains bus connectent un nombre limité de processeurs entre eux, ces bus étant reliés par d'autres bus, et ainsi de suite. Mais la méthode la plus simple est d'utiliser un protocole de cohérence des caches à base de répertoires. Pour rappel, un répertoire est simplement une table de correspondance qui mémorise, pour chaque ligne de cache, toutes les informations nécessaires pour maintenir sa cohérence. Ce répertoire mémorise :

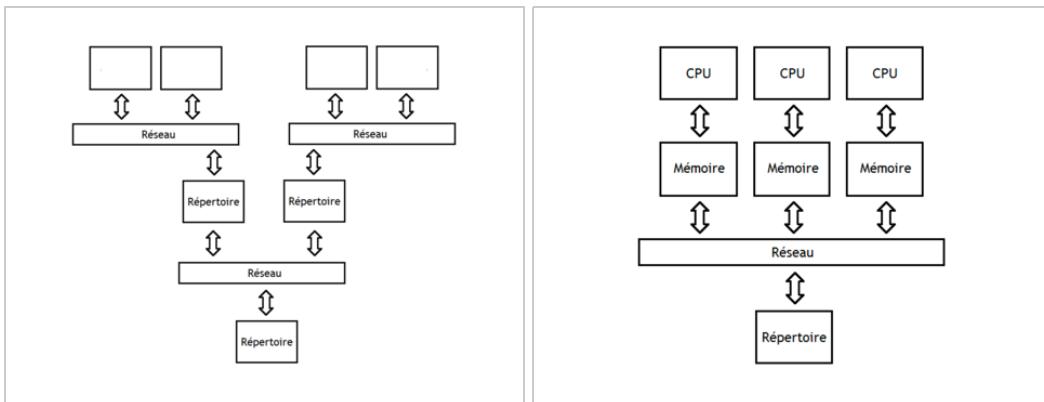
- l'état de la ligne de cache (exclusive, shared, invalid, empty, etc) ;
- la liste des processeurs dont le cache contient une copie de la ligne de cache.

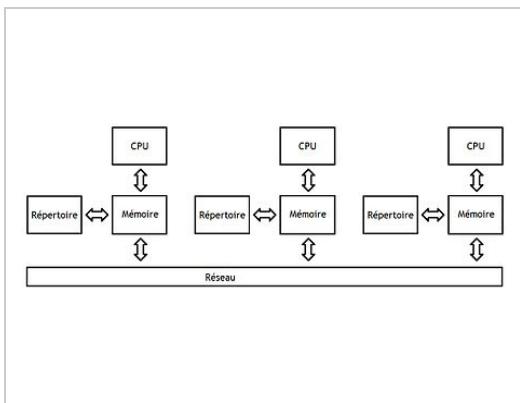


Localisation du nœud maison

Localiser les copies valides d'une ligne peut être assez compliqué : soit celle-ci est en mémoire locale, soit celle-ci est dans le cache d'un autre processeur (en état modified). On peut classer la méthode de localisation de la ligne de cache en trois grands catégories :

- Dans le cas hiérarchique, le répertoire est réparti dans un arbre. Chaque répertoire (nœud) indique quel sous-arbre il faut vérifier pour obtenir la donnée demandée. Avec cette méthode, le répertoire utilise peu de bits (de mémoire) pour mémoriser les informations nécessaires. Mais le nombre d'accès mémoire peut rapidement augmenter et dépasser celui obtenu avec les autres solutions.
- Dans le cas centralisé, on a un répertoire unique pour tous les processeurs. A chaque cache miss, le répertoire est consulté pour localiser la copie valide, afin de la charger dans le cache. Dans ces conditions, le répertoire doit répondre aux demandes de tous les processeurs, et devient rapidement un goulet d'étranglement quand les cache miss sont nombreux. En contrepartie, le nombre d'accès réseau nécessaire pour localiser une donnée est minimal : il suffit d'un seul accès réseau.
- Dans le cas plat, on a un répertoire pour chaque processeur. Le répertoire à interroger est celui lié à la mémoire locale qui contient une copie valide ou invalide de la donnée. Ce répertoire peut alors indiquer où se situe la copie valide de la donnée.





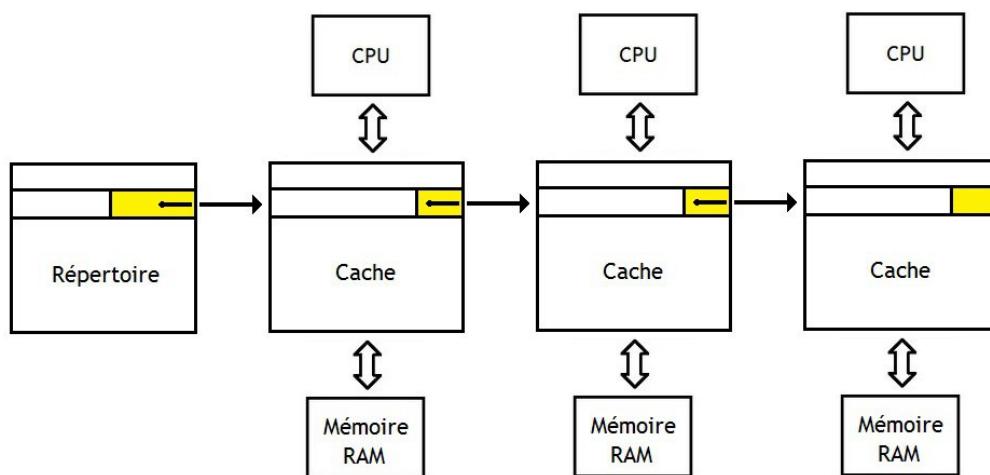
Cohérence des caches - Répertoire étalé

Listes des copies valides

Avec les répertoires plats, le répertoire doit mémoriser la liste des processeurs qui contiennent la donnée invalide : il faut que ces processeurs soient prévenus qu'ils doivent invalider la ligne de cache. Dans le cas le plus simple, le répertoire est une mémoire RAM, dont chaque entrée (chaque byte) mémorise les informations pour une ligne de cache : on parle de répertoire basé sur une mémoire. Une autre solution consiste à utiliser des bits de présence : le n ième bit de présence indique si le n ième processeur a une copie de la donnée dans son cache. Une autre solution consiste à numérotter les processeurs et à mémoriser la liste des numéros des processeurs dans le répertoire.

Toutes ces méthodes posent problème quand le nombre de processeur augmente trop. Ce problème peut être résolu en n'autorisant qu'un nombre maximal de copies d'une ligne de cache : soit un invalide automatiquement une copie quand ce nombre est dépassé, soit on interdit toute copie une fois ce maximum atteint. Une autre solution consiste à autoriser les copies en trop, mais toute invalidation de ces copies sera envoyée à tous les processeurs, même s'ils n'ont pas de copie de la donnée invalide.

Ces solutions sont cependant assez lourdes, et des alternatives existent. Ces alternatives consistent à mémoriser la liste des copies dans les caches eux-mêmes. Le répertoire n'identifie, pour chaque ligne de cache, qu'un seul processeur : plus précisément, il identifie la ligne de cache du processeur qui contient la copie. A l'intérieur d'une ligne de cache, la copie suivante (si elle existe) est indiquée dans les bits de contrôle. On parle de répertoires à base de caches.



Architecture à parallélisme de données

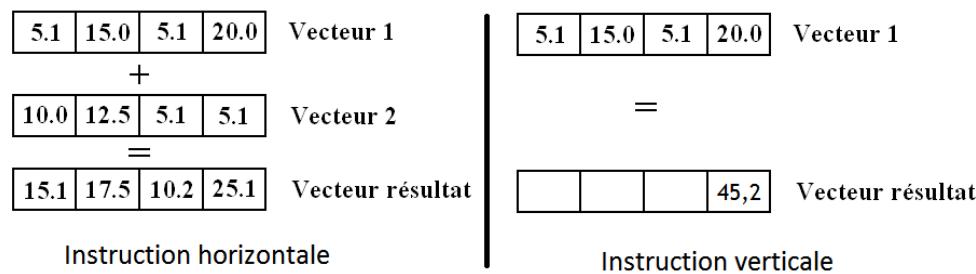
Nous allons maintenant aborder le parallélisme de données, qui consiste à traiter des données différentes en parallèle. De nombreuses situations s'y prêtent relativement bien : traitement d'image, manipulation de sons, vidéo, rendu 3d, etc. Mais pour exploiter ce parallélisme, il a fallu concevoir des processeurs adaptés. Une solution simple est d'utiliser plusieurs processeurs qui exécutent la même instruction, chacun sur des données différentes. Cette solution a autrefois été utilisée sur certains supercalculateurs, comme les Thinking machines CM-1 et CM-2. Ces ordinateurs possédaient environ 64000 processeurs minimalistes, qui exécutaient tous la même instruction au même cycle d'horloge. Mais ce genre de solution est vraiment quelque chose d'assez lourd, qui ne peut être efficace et rentable que sur des grosses données, et sur des ordinateurs chers et destinés à des calculs relativement importants. N'espérez pas commander ce genre d'ordinateurs pour noël ! Ces architectures ont depuis été remplacées par des architecture qui exploitent le parallélisme de données au niveau de l'unité de calcul, celle-ci pouvant exécuter des calculs en parallèle. Nous allons aborder ces architectures dans ce chapitre.

Instructions SIMD

Certains processeurs fournissent des instructions capables de traiter plusieurs éléments en parallèle. Ces instructions sont appelées des **instructions vectorielles**. Ces instructions vectorielles travaillent sur un ou plusieurs nombres entiers ou flottants placés les uns à côté des autres, qui forment ce qu'on appelle un vecteur. Quand on exécute une instruction sur un vecteur, celle-ci traite ces entiers ou flottants en parallèle, simultanément.

Instructions SIMD

Les instructions SIMD sont difficilement utilisables dans des langages de haut niveau et c'est donc au compilateur de traduire un programme avec des instructions vectorielles. Les transformations qui permettent de traduire des morceaux de programmes en instructions vectorielles (déroulage de boucles, strip-mining) portent le nom de vectorisation. Ces instructions vectorielles peuvent être rassemblées en deux grands groupes : les horizontales et les verticales. Les instructions horizontales travaillent en parallèle sur les éléments qui sont "à la même place" dans deux vecteurs. Les instructions verticales vont réduire un vecteur en un simple nombre. Elle peuvent calculer la somme des éléments d'un vecteur, calculer le produit de tous les éléments d'un vecteur, renvoyer le nombre d'éléments nuls dans un vecteur, etc.

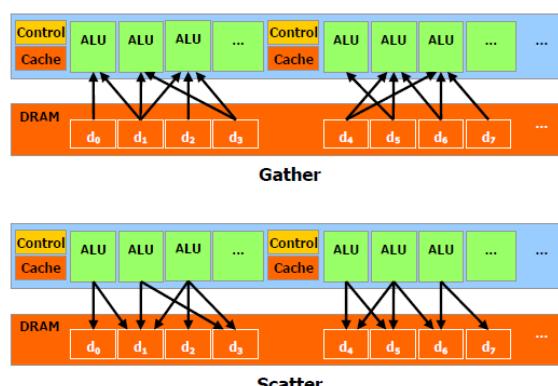


Certaines instructions sont chargées de copier des données de la RAM dans les registres vectoriels. Les modes d'adressages possibles pour ces instructions sont au nombre de trois :

- adressage contiguë ;
- adressage en stride ;
- adressage en scatter-gather.

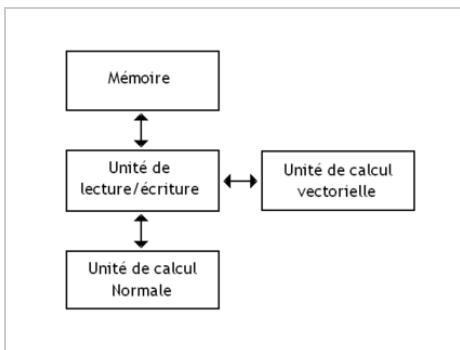
Généralement, les données d'un vecteur sont placées les unes à côté des autres en mémoire. L'instruction a juste à connaître l'adresse du vecteur en mémoire pour faire la copie dans les registres. Il arrive aussi que l'on accède à certains éléments séparés par une même distance. Par exemple, si on fait des calculs de géométrie dans l'espace, on peut très bien ne vouloir traiter que les coordonnées sur l'axe des x, sans toucher à l'axe des y ou des z. De tels accès sont ce qu'on appelle des accès en stride. L'instruction a juste à dire au processeur de toujours charger la donnée située x cases plus loin, le x étant souvent fourni via l'instruction (il est incorporé dans la suite de bits de l'instruction).

Les processeurs vectoriels incorporent aussi un autre type d'accès à la mémoire, le scatter-gather ! Cet accès demande d'avoir une liste d'adresses mémoires des données. Quand une instruction exécute un accès en scatter, les données présentes dans ces adresses sont rassemblées dans un vecteur, et enregistrée dans un registre vectoriel. Bien sûr, il existe aussi l'inverse : on peut écrire les données d'un vecteur dans des emplacements mémoires dispersés : c'est l'accès en gather.

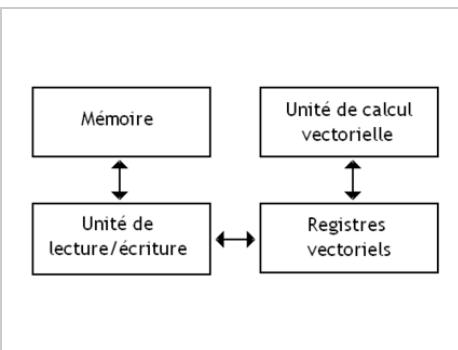


Accès mémoire

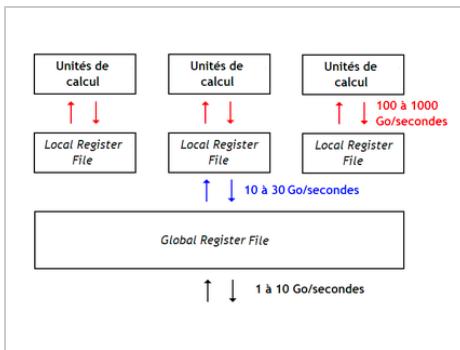
Sur certains processeurs assez anciens, les instructions vectorielles lisent et écrivent en mémoire RAM, sans passer par des registres : on parle de processeurs memory-memory. Les processeurs récents possèdent des registres vectoriels dédiés aux vecteurs. La taille d'un vecteur est fixée par le jeu d'instruction, ce qui fait que ces registres ont une taille fixe. Suivant la taille des données à manipuler, on pourra en placer plus ou moins dans un vecteur. Les processeurs de flux utilisent une hiérarchie de registres.



Processeur vectoriel mémoire-mémoire.



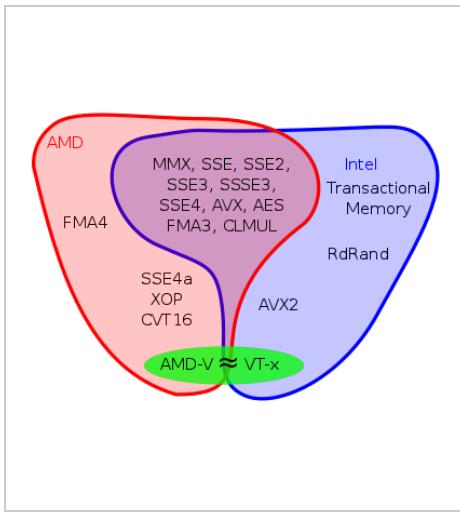
Processeur vectoriel à registres vectoriels.



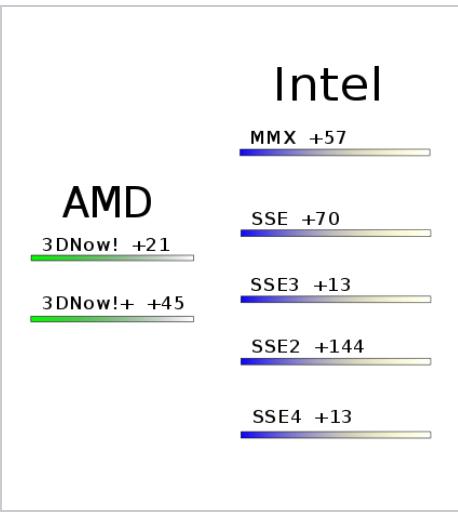
Processeur de flux.

Exemple avec les processeurs x86

On a donc vu un peu de théorie sur ces instructions SIMD, mais un peu de pratique ne ferait sûrement pas de mal. On va donc voir à quoi peuvent bien ressembler ces instructions avec des exemples concrets : on va étudier les instructions des processeurs x86, présents dans nos PC. Le jeu d'instruction de nos PC qui fonctionnent sous Windows est appelé le x86. C'est un jeu d'instructions particulièrement ancien, apparu certainement avant votre naissance : 1978. Depuis, de plus en plus d'instructions ont été ajoutées et rajoutées : ces instructions sont ce qu'on appelle des extensions x86. On peut citer par exemple les extensions MMX, SSE, SSE2, voir 3Dnow!. Les premières instructions SIMD furent fournies par une extension x86 du nom de **MMX**. Ce MMX a été introduit par Intel dans les années 1996 sur le processeur Pentium MMX et a perduré durant quelques années avant d'être remplacé par les extensions SSE. La même année, AMD sorti d'expansion 3DNow!, qui ajoutait 21 instructions SIMD similaires à celles du MMX. Celui-ci fut suivi du 3DNow! + quelques années plus tard. Le SSE, successeur du MMX, fut décliné en plusieurs versions avant d'être remplacé par le AVX. Une grande quantité de ces extensions x86 sont des ajouts d'instructions SIMD. Les processeurs PowerPC, ARM, SPARC, MIPS et bien d'autres, ont eux aussi des extensions de ce genre.



Extensions du jeu d'instruction x86, 2013.

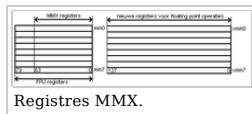


Chronologie des extensions x86 SIMD.

Commençons par étudier le MMX. Le MMX ajoutait pas mal d'instructions SIMD assez basiques, essentiellement des instructions arithmétiques : addition, soustraction, opérations logiques, décalages, rotations, mise à zéro d'un registre, etc. La multiplication est aussi supportée, mais avec quelques petites subtilités, via l'instruction PMULLW. Ce MMX introduisait 8 registres vectoriels, du nom de MM0, MM1, MM2, MM3, MM4, MM5, MM6 et MM7. Ceux-ci avaient une taille de 64 bits et ne pouvaient contenir que des nombres entiers : pas de nombres flottants à l'horizon. Il n'y a pas de registre d'état pour les instructions MMX. Pire : les instructions MMX ne mettent aucun registre d'état à jour et ne préviennent pas en cas d'overflow ou d'underflow si ceux-ci arrivent (pour les instructions qui ne travaillent pas en arithmétique saturée).

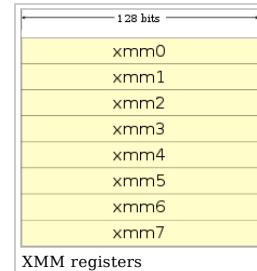
Ces registres MMX avaient cependant un léger défaut, qui a nui à l'adoption du MMX. Pour comprendre ce défaut, il faut savoir que les flottants sont gérés sur les processeurs x86 par une extension du jeu d'instruction x86 : le jeu d'instruction x87. Celui-ci définit 8 registres flottants de 80

bits, ainsi qu'à un registre d'état de 32 bits. Et c'est là qu'est le problème : chaque registre MMX correspondait aux 64 bits de poids faible d'un des 8 registres flottants de la x87 ! En clair : il était impossible d'utiliser en même temps l'unité de calcul flottante et l'unité MMX. Le pire, c'est que ce n'était pas un bug, mais quelque chose de voulu par Intel. En faisant ainsi, il n'y avait pas à sauvegarder 8 registres supplémentaires lorsqu'on appelait une fonction ou qu'on devait interrompre un programme pour en lancer un autre (changement de contexte, interruption, etc) : la sauvegarde des registres de la FPU x87 suffisait. Cela a sacrément gêné les programmeurs ou les compilateurs qui voulaient utiliser le jeu d'instruction MMX.



Dans les années 1999, une nouvelle extension SIMD fit son apparition sur les processeurs Intel Pentium 3 : le **Streaming SIMD Extensions**, abrégé SSE. Ce SSE fut ensuite complété, et différentes versions virent le jour : le SSE2, SSE3, SSE4, etc. Cette extension fit apparaître 8 nouveaux registres, les registres XMM. Sur les processeurs 64 bits, ces registres sont doublés et on en trouve donc 16. En plus de ces registres, on trouve aussi un registre d'état qui permet de contrôler le comportement des instructions SSE : celui-ci contient des bits qui permettront de dire au processeur que les instructions doivent arrondir leurs calculs d'une certaine façon, etc. Ce registre n'est autre que le registre MXCSR. Chose étrange, seuls les 16 premiers bits de ce registre ont une utilité : les concepteurs du SSE ont sûrement préféré laisser un peu de marge au cas où.

La première version du SSE contenait assez peu d'instructions : seulement 70. Le SSE première version ne fournissait que des instructions pouvant manipuler des paquets contenant 4 nombres flottants de 32 bits (simple précision). Je ne vais pas toutes les lister, mais je peux quand-même dire qu'en trouvait des instructions arithmétiques de base, avec pas mal d'opérations en plus : permutations, opérations arithmétiques complexes, instructions pour charger des données depuis la mémoire dans un registre. Petit détail : la multiplication est gérée plus simplement et l'on a pas besoin de s'embêter à faire mumuse avec plusieurs instructions différentes pour faire une simple multiplication comme avec le MMX.



On peut quand même signaler une chose : des instructions permettant de contrôler le cache firent leur apparition. On retrouve ainsi des instructions qui permettent d'écrire ou de lire le contenu d'un registre XMM en mémoire sans le copier dans le cache. Ces instructions permettent ainsi de garder le cache propre en évitant de copier inutilement des données dedans. On peut citer par exemple, les instructions MOVNTQ et MOVNTPS du SSE première version. On trouve aussi des instructions permettant de charger le contenu d'une portion de mémoire dans le cache, ce qui permet de contrôler son contenu. De telles instructions de prefetch permettent ainsi de charger à l'avance une donnée dont on aura besoin, permettant de supprimer pas mal de cache miss. Le SSE fournissait notamment les instructions PREFETCH0, PREFETCH1, PREFETCH2 et PREFETCHNNTA. Autant vous dire qu'utiliser ces instructions peut donner lieu à de sacrés gains si on s'y prend correctement ! Il faut tout de même noter que le SSE n'est pas seul "jeu d'instruction" incorporant des instructions de contrôle du cache : certains jeux d'instruction POWER PC (je pense à l'Altivec) ont aussi cette particularité.

Avec le **SSE2**, de nouvelles instructions furent ajoutées, permettant d'utiliser des nombres de 64, 16 et 8 bits dans chaque vecteur. Le SSE2 incorporait ainsi pas moins de 144 instructions différentes, instructions du SSE première version incluses. Ce qui commençait à faire beaucoup.

Puis, vient le **SSE3**, avec ses 13 instructions supplémentaires. Pas grand chose à signaler, si ce n'est que des instructions permettant d'additionner ou de soustraire tous les éléments d'un paquet SSE ensemble, des instructions pour les nombres complexes, et plus intéressant : les deux instructions MWAIT et MONITOR qui permettent de paralléliser plus facilement des programmes.

Le **SSE4** fut un peu plus complexe et fut décliné lui-même en 2 sous-versions. Le SSE4.1 introduit ainsi des opérations de calcul de moyenne, de copie conditionnelle de registre (un registre est copié dans un autre si le résultat d'une opération de comparaison précédente est vrai), de calcul de produits scalaires, de calcul du minimum ou du maximum de deux entiers, des calculs d'arrondis, et quelques autres. Avec le SSE4.2, le vice a été poussé jusqu'à incorporer des instructions de traitement de chaînes de caractères.

Enfin, la dernière extension en date est l'**AVX**. Avec celle-ci, on retrouve 16 registres nommés de YMM0 à YMM15, dédiés aux instructions AVX et d'une taille de 256 bits. Ces registres YMM sont partagés avec les registres XMM : les 128 bits de poids faible des registres YMM ne sont autre que les registres XMM. L'AVX complète le SSE et ses extensions, en rajoutant quelques instructions, et surtout en permettant de traiter des données de 256 bits. Son principal atout face au SSE, et que les instructions AVX permettent de préciser le registre de destination en plus des registres stockant les opérandes. Avec le SSE et le MMX, le résultat d'une instruction SIMD était écrit dans un des deux registres d'opérande manipulé par l'instruction : il fallait donc sauvegarder son contenu si on en avait besoin plus tard. Avec l'AVX, ce n'est plus le cas : on peut se passer des opérations de sauvegarde sans problème, ce qui supprime pas mal d'instructions.

	255	128	0
YMM0		XMM0	
YMM1		XMM1	
YMM2		XMM2	
YMM3		XMM3	
YMM4		XMM4	
YMM5		XMM5	
YMM6		XMM6	
YMM7		XMM7	
YMM8		XMM8	
YMM9		XMM9	
YMM10		XMM10	
YMM11		XMM11	
YMM12		XMM12	
YMM13		XMM13	
YMM14		XMM14	
YMM15		XMM15	

Processeurs vectoriels

Les instructions SIMD sont assez rudimentaires, et certains processeurs assez anciens allaient un peu plus loin : ces processeurs sont ce qu'on appelle des processeurs vectoriels. Ils incorporent diverses techniques que de simples instructions SIMD n'ont pas forcément.

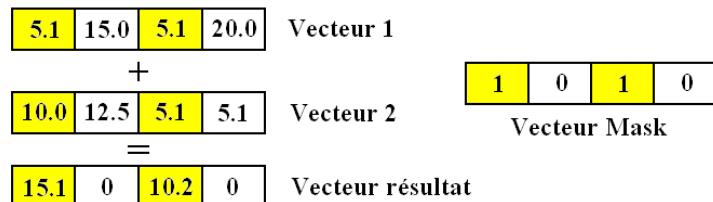
Vector Length Register

La première technique permet de gérer les tableaux dont la taille n'est pas un multiple d'un vecteur, en ajoutant un registre spécial : le **Vector Length Register**. Celui-ci indique combien d'éléments on doit traiter dans un vecteur. On peut ainsi dire au processeur : je veux que tu ne traite que les 40 premiers éléments présents d'un paquet. Quand on arrive à la fin d'un tableau, il suffit de configurer le Vector Length Register pour ne traiter que ce qu'il faut.

Instruction 1	Instruction 2	Instruction 3	
Vecteur de 64 éléments	Vecteur de 64 éléments	Vecteur de 20 éléments	Tableau de 148 éléments
64	64	20	Vector length register

Vector Mask Register

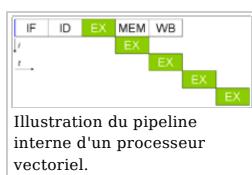
Si une boucle contient des branchements conditionnels, elle ne peut pas être vectorisée facilement : il est impossible de zapper certains éléments d'un vecteur suivant une condition. Pour résoudre ce problème, les processeurs vectoriels utilisent un registre de mask, ou Vector Mask Register. Celui-ci stocke un bit pour chaque donnée présente dans le vecteur à traiter, qui indique s'il faut ignorer la donnée ou non.



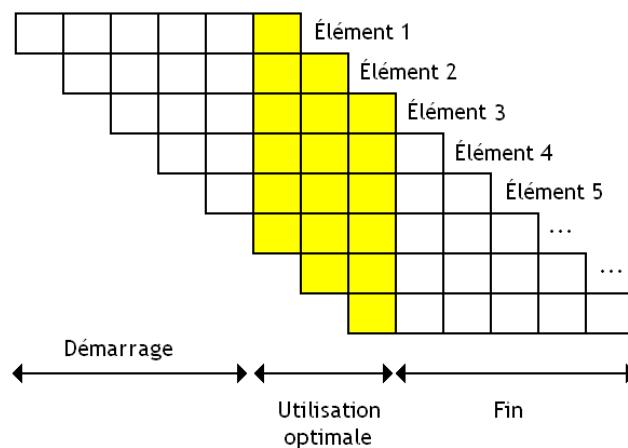
Micro-architecture

Mais surtout, les processeurs vectoriels ne font pas leurs calculs de la même manière que les processeurs SIMD, et n'accèdent pas à la mémoire de la même façon. Par exemple, pour permettre les accès en scatter-gather, les processeurs vectoriels sont souvent combinés à des mémoires multiports ou pipelinées. De plus, les processeurs vectoriels ne possèdent aucune mémoire cache pour les données (même s'ils ont des cache d'instruction).

La différence entre processeur vectoriel et SIMD tient dans la façon dont sont traités les vecteurs : les instructions SIMD traitent chaque élément en parallèle, alors que les processeurs vectoriels pipelinent ces calculs ! Par pipeliner, on veut dire que l'exécution de chaque instruction est découpée en plusieurs étapes indépendantes : au lieu d'attendre la fin de l'exécution d'une opération avant de passer à la suivante, on peut ainsi commencer le traitement d'une nouvelle donnée sans avoir à attendre que l'ancienne soit terminée.



Avec une unité de calcul pipelinée découpée en N étages, on peut gérer plusieurs opérations simultanées : autant qu'il y a d'étapes différentes. Mais ce nombre maximal d'opérations met un certain temps avant d'être atteint : l'unité de calcul met un certain temps avant d'arriver à son régime de croisière. Durant un certain temps, elle n'a pas commencé à traiter suffisamment d'éléments pour que toutes les étapes soient occupées. Ce temps de démarrage est strictement égal du nombre d'étapes nécessaires pour effectuer une instruction. La même chose arrive vers la fin du vecteur, quand il ne reste plus suffisamment d'éléments à traiter pour remplir toutes les étapes.

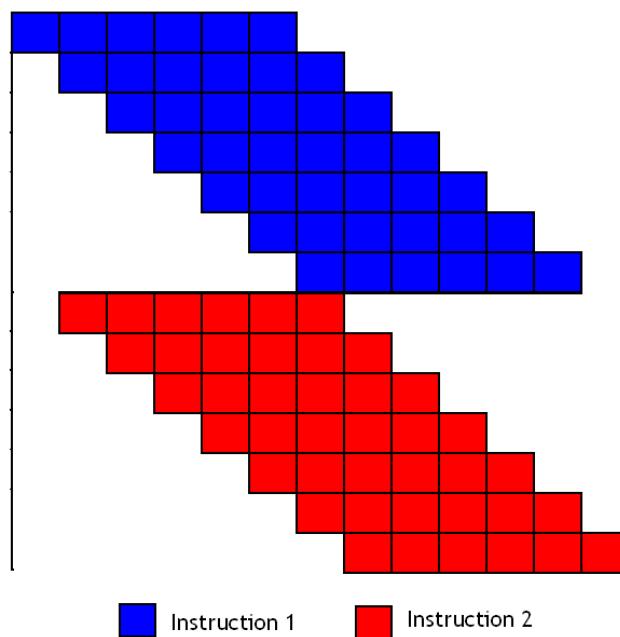


Pour amortir ces temps de démarrage et de fin, certains processeurs démarrent une nouvelle instruction sans attendre la fin de la précédente : deux instructions peuvent se chevaucher pour remplir les vides. Par contre, il peut y avoir des problèmes lorsque deux instructions qui se suivent manipulent le même vecteur. Il faut que la première instruction aie fini de manipuler un élément avant que la suivante ne veuille commencer à le modifier. Pour cela, il faut avoir des vecteurs suffisamment grands qui contiennent plus d'éléments qu'il n'y a d'étapes pour effectuer notre instruction.

Certains de nos processeurs vectoriels utilisent plusieurs ALU pour accélérer les calculs. En clair, ils sont capables de traiter les éléments d'un vecteur dans des ALU différentes. Vu que nos ALU sont pipelinées, rien n'empêche d'exécuter plusieurs instructions vectorielles différentes dans une seule ALU, chaque instruction étant à une étape différente. Mais d'ordinaire, les processeurs vectoriels comportent plusieurs ALUs spécialisées : une pour les additions, une pour la multiplication, une pour la division, etc.

Cette technique du pipeline peut encore être améliorée dans certains cas particuliers. Imaginons que l'on aie trois paquets : A, B et C. Pour chaque ième élément de ces paquets, je souhaite effectuer le calcul $A_i + B_i \times C_i$. Mon processeur ne disposant pas d'instruction permettant de faire en une fois ce calcul, je dois utiliser deux instruction vectorielles : une d'addition, et une autre de multiplication. On pourrait penser que l'on doit effectuer d'abord la multiplication des paquets B et C, stocker le résultat dans un paquet temporaire, et effectuer l'addition de ce tableau avec le paquet A. Mais en rusanant un peu, on s'aperçoit qu'on peut simplifier le tout et utiliser notre pipeline plus correctement. Une fois que le tout premier résultat

de la multiplication du premier vecteur est connu, pourquoi ne pas démarrer l'addition immédiatement après, et continuer la multiplication en parallèle ? Après, tout les deux calculs ont lieu dans des ALUs séparés. Il s'agit de ce qu'on appelle le Vector Chaining.



Pour ce faire, on doit modifier notre pipeline de façon à ce que le résultat de chaque étape d'un calcul soit réutilisable au cycle d'horloge suivant. La sortie de l'unité de multiplication doit être connectée à l'entrée de l'ALU d'addition. Un processeur implémentant le chaining a toutes ses unités de calcul reliées entre elles de cette façon : la sortie d'une unité est reliée aux entrées de toutes les autres.

Le noyau d'un système d'exploitation

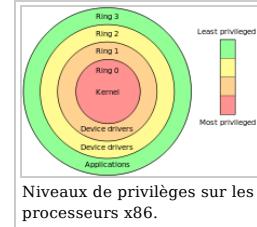
À tout début de l'informatique, les logiciels n'étaient pas compatibles sur une large gamme de matériel. Avec le temps, les informaticiens ont inventé des techniques d'abstraction matérielle qui permettent à un programme de s'exécuter sur des ordinateurs avec des matériels très différents. Il ont pour cela fabriqué . Rendre les programmes indépendants du matériel a demandé de revoir l'organisation des logiciels, qui ont dû être découpés en plusieurs programmes séparés :

- les **programmes systèmes** gèrent la mémoire et les périphériques ;
- les **programmes applicatifs** ou applications déléguent la gestion de la mémoire et des périphériques aux programmes systèmes.

Anneaux mémoire

L'OS doit garantir que seuls les programmes systèmes ont accès aux périphériques ou aux registres de gestion de la mémoire virtuelle. Pour cela, les processeurs actuels incorporent une technique : les **anneaux mémoires**. Dans les grandes lignes, le processeur gère deux niveaux de priviléges : un mode noyau pour les programmes systèmes, où les instructions d'accès aux périphériques peuvent s'exécuter, et un mode utilisateur pour les applications, où ces opérations sont interdites.

Ces anneaux mémoire/niveaux de priviléges, sont gérés en partie par le processeur. Celui-ci contient un registre qui précise s'il est en espace noyau ou en espace utilisateur. A chaque accès mémoire ou exécution d'instruction, le processeur vérifie si le niveau de privilège permet l'opération demandée. Lorsqu'un programme effectue une instruction interdite en mode utilisateur, une exception matérielle est levée. Généralement, le programme est arrêté sauvagement et un message d'erreur est affiché. Un programme ne peut changer d'anneau mémoire au cours de son exécution, sous certaines conditions relativement drastiques, afin d'éviter que tout programme s'arrogue des droits d'accès aux périphériques sans restrictions.



Sur certains processeurs, on trouve des niveaux de priviléges intermédiaires entre l'espace noyau et l'espace utilisateur. Les processeurs de nos PC actuels contiennent 4 niveaux de priviléges. Le système Honeywell 6180 en possédait 8. À l'origine, ceux-ci ont été inventés pour faciliter la programmation des pilotes de périphériques. Certains d'entre eux peuvent en effet gagner à avoir des niveaux de priviléges intermédiaires entre celui d'une simple application et celui d'un OS. Mais force est de constater que ceux-ci ne sont pas vraiment utilisés, seuls les espaces noyau et utilisateur étant pertinents. Il en est de même pour beaucoup de méthodes de protection mémoire. Une des raisons à cet état de fait est tout simplement la compatibilité entre architectures matérielles différentes. Par exemple, les premières versions de Windows NT n'utilisaient que deux anneaux de priviléges sur les processeurs x86, en partie parce que d'autres jeu d'instructions supportés par Windows n'avaient que deux niveaux de priviléges. Ce n'est pas la seule raison, la facilité de programmation de l'OS devant aussi être prise en compte.

Appels systèmes

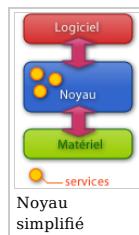
Tous les programmes systèmes sont des routines d'interruptions, fournies par l'OS ou les pilotes, qui permettent d'exploiter les périphériques. Sur les PC actuels, où le BIOS fournit des routines de base, l'OS doit modifier le vecteur d'interruption avec les adresses de ses routines. On dit qu'il détourne/détourne l'interruption. Les applications peuvent appeler à la demande ces routines via une interruption logicielle : ils effectuent ce qu'on appelle un **appel système**. Tout OS fournit un ensemble d'appels systèmes de base, qui servent à manipuler la mémoire, gérer des fichiers, etc. Par exemple, Linux fournit les appels systèmes open, read, write et close pour manipuler des fichiers, les appels brk, sbrk, pour allouer et désallouer de la mémoire, etc. Évidemment, ceux-ci ne sont pas les seules : Linux fournit environ 380 appels systèmes distincts. Ceux-ci sont souvent encapsulés dans des bibliothèques et peuvent s'utiliser comme simples fonctions.

L'appel système se charge automatiquement de switcher le processeur dans l'espace noyau. Cette commutation n'est cependant pas gratuite, de même que l'interruption qui lui est associée. Ainsi, les appels systèmes sont généralement considérés comme lents, très lents. Divers processeurs incorporent des techniques pour rendre ces commutations plus rapides, via des instructions spécialisées (SYSCALL/SYSRET et SYSENTER/SYSEXIT d'AMD et Intel), ou d'autres techniques (call gate de Intel, Supervisor Call instruction des IBM 360, etc).

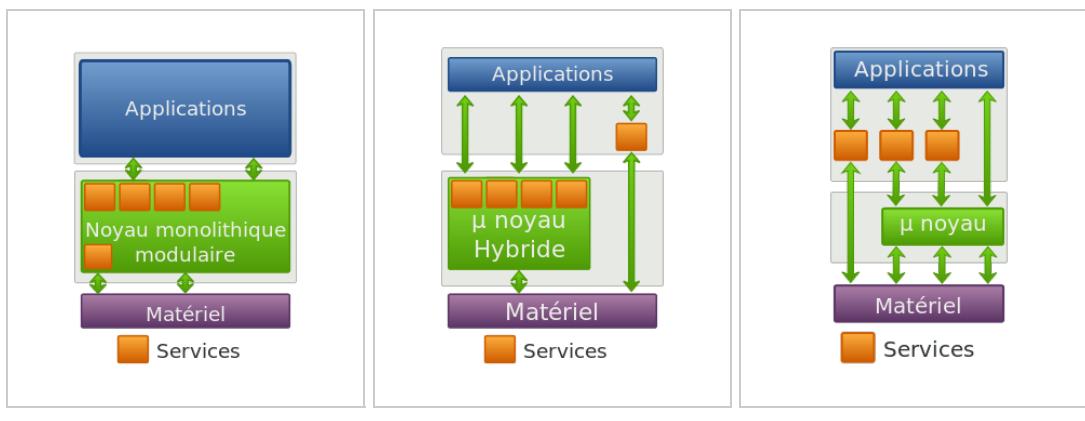
Le noyau du système d'exploitation

La portion du système d'exploitation placée dans l'espace noyau est ce qu'on appelle le **noyau du système d'exploitation**, le reste de l'OS étant composé d'applications qui servent à afficher une interface graphique ou une ligne de commande, par exemple. Reste que beaucoup de programmes de l'OS peuvent être placés indifféremment dans le noyau ou dans l'espace utilisateur. Mais cela a un coût : exécuter un appel système est très lent, changer de niveau de privilège étant assez coûteux. Conserver de bonnes performances impose de diminuer la quantité d'appels systèmes, et donc de placer un maximum de choses en espace noyau. Mais cela se fait au prix de la sécurité, toute erreur de programmation dans le noyau entraînant un écran bleu. On peut ainsi classer les noyaux en plusieurs types, selon l'accent mis sur les performances ou la sécurité.

- Les **noyaux monolithiques** placent un maximum de programmes systèmes dans l'espace noyau. Leurs performances sont donc excellentes, vu que les appels systèmes à faire sont peu nombreux. Par contre, leur fiabilité est plutôt faible, la majorité des erreurs de programmation de l'OS se trouvant dans le noyau, source de beaucoup d'écrans bleus.
- Les **micro-noyaux** préfèrent au contraire placer le moins de choses dans l'espace utilisateur. Leur sécurité est excellente, la majorité des erreurs de programmation n'entraînant pas un crash, mais simplement l'affichage d'un message d'erreur. Mais le nombre d'appels système est nettement plus important que pour les noyaux monolithiques, ce qui source de performances relativement faibles.
- Les **noyaux hybrides**, sont un intermédiaire entre les deux précédents.
- Certaines versions relativement extrêmes de noyaux monolithiques, où tout l'OS est placé dans l'espace noyau, portent le nom de **noyaux mégalithiques**. De même, on trouve des versions extrêmes de micro-noyaux, où tout l'OS est composé de bibliothèques logicielles exécutées en espace utilisateur, à l'exception du minimum vital. On parle alors d'**exokernel** ou de **nanokernel**.



Pour information, les systèmes Unix et Linux sont basés sur des noyaux monolithiques, tandis que les systèmes Windows utilisent des micro-noyaux.



La multiprogrammation

Les tout premiers OS datent des années 1950, et ceux-ci ont commencé à devenir de plus en plus utilisés à partir des années 60-70. Évidemment, ces OS étaient relativement simples. Notamment, ceux-ci ne pouvaient pas exécuter plusieurs programmes en même temps. Il était ainsi impossible de démarrer à la fois un navigateur internet, tout en écoutant de la musique. Dit autrement, ces systèmes d'exploitation ne permettaient de ne démarrer qu'un seul programme à la fois. On appelait ces OS des OS **mono-programmés**.

Mais cette solution avait un problème. Lors de l'accès à un périphérique, le processeur doit attendre que le transfert avec le périphérique ait cessé et est inutilisé durant tout ce temps. Pour certains programmes qui accèdent beaucoup aux périphériques, il est possible que le processeur ne soit utilisé que 20% du temps, voire moins. Une solution à cela est d'exécuter un autre programme pendant que le programme principal accède aux périphériques. Un OS qui permet cela est un OS **multiprogrammé**. Le nombre de programme pouvant être chargés en mémoire simultanément est appelé le taux de multiprogrammation. Plus celui-ci est élevé, plus l'OS utilisera le processeur à ses capacités maximales.

Les OS actuels vont plus loin et permettent d'exécuter plusieurs programmes "simultanément", même si aucun programme n'accède aux périphériques. Les OS de ce genre sont des **OS multi-tâche**. L'apparition de la multiprogrammation a posé de nombreux problèmes, notamment au niveau du partage de la mémoire et du processeur. Dans ce qui va suivre, nous allons voir comment ces OS gèrent la mémoire et les commutations entre programmes (appelés processus sur ces OS). Pour manipuler plusieurs processus, l'OS doit mémoriser des informations sur eux. Ces informations sont stockées dans ce qu'on appelle un Process Control Block, une portion de la mémoire.

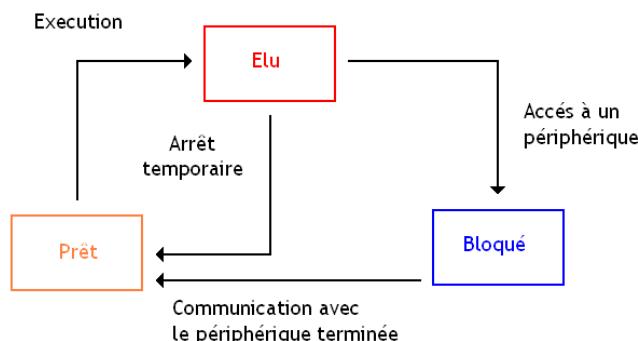
Allocation du processeur

Le système d'exploitation utilise une technique très simple pour permettre la multiprogrammation sur les ordinateurs à un seul processeur : l'ordonnancement. Cela consiste à constamment switcher entre les différents programmes qui doivent être exécutés : en switchant assez vite, on peut donner l'illusion que plusieurs processus s'exécutent en même temps.



Avec cette technique, chaque processus peut être dans (au moins) trois états :

- Élu : un processus en état élu est en train de s'exécuter sur le processeur.
- Prêt : celui-ci a besoin de s'exécuter sur le processeur et attend son tour.
- Bloqué : celui-ci n'a pas besoin de s'exécuter (par exemple, parce que celui-ci attend une donnée en provenance d'une entrée-sortie).



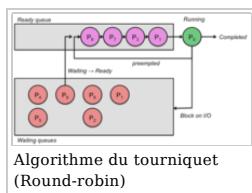
Les processus en état prêt sont placés dans une **file d'attente**, le nombre de programme dans cette liste a une limite fixée par le système d'exploitation. Lorsque l'O.S décide de switcher de processus, il doit choisir un processus dans la file d'attente et le lancer sur le processeur. Pour comprendre comment faire, il faut se souvenir qu'un processus manipule un ensemble de registres processeur, aussi appelé **contexte d'exécution du processus**. Pour qu'un processus puisse reprendre où il en était, il faut que son contexte d'exécution redevienne ce qu'il était. Pour cela, ce contexte d'exécution est sauvegardé quand il est interrompu et restauré lors de l'ordonnancement. On appelle cela une **commutation de contexte**. Avec cette méthode de sauvegarde du contexte, le système d'exploitation doit fatalement mémoriser tous les contextes des processus dans une liste appelée **table des processus**, elle-même très liée à la file d'attente.

Reste que pour utiliser notre processeur au mieux, il faut faire en sorte que tous les processus aient leur part du gâteau. C'est ce qu'on appelle l'**ordonnancement**. Il existe deux grandes formes d'ordonnancements : l'ordonnancement collaboratif, et l'ordonnancement préemptif. Dans le premier cas, c'est le processus lui-même qui décide de passer de l'état élu à l'état bloqué ou prêt, par l'intermédiaire d'un appel système. Dans le second, c'est le système d'exploitation qui stoppe l'exécution d'un processus. L'avantage du dernier cas est que les processus ne peuvent plus monopoliser le processeur.

L'ordonnancement préemptif se base souvent sur la technique du **quantum de temps** : chaque programme s'exécute durant un temps fixé une bonne fois pour toute. En clair, toutes les « x » millisecondes, le programme en cours d'exécution est interrompu et l'ordonnanceur exécuté. Ainsi, il est presque impossible qu'un processus un peu trop égoïste puisse monopoliser le processeur au dépend des autres processus. La durée du quantum de temps doit être grande devant le temps mis pour changer de programme. En même temps, on doit avoir un temps suffisamment petit pour le cas où un grand nombre de programmes s'exécutent simultanément. Généralement, un quantum de temps de 100 millisecondes donne de bons résultats.

Comme on vient de le voir, l'ordonnancement est un processus relativement complexe. Cependant, nous n'avons pas encore abordé les algorithmes qui permettent de choisir quel programme faire passer de l'état prêt à l'état élu. Pour faire simple, nous allons en voir trois, qui sont relativement importants, et très proches de ceux utilisés dans les OS actuels.

- **L'algorithme du tourniquet** exécute chaque programme l'un après l'autre, dans l'ordre. Cet algorithme est redoutablement efficace et des OS comme Windows ou Linux l'ont utilisé durant longtemps. Mais cet algorithme a un défaut : le choix du quantum de temps doit être calibré au mieux et n'être ni trop court, ni trop long.



- **L'algorithme des files d'attentes multiple-niveau** donne la priorité aux programmes rapides ou qui accèdent souvent aux périphériques. Cet algorithme utilise plusieurs files d'attentes, auxquelles ont donné des quantums de temps différents. Tout programme démarre dans la file d'attente avec le plus petit quantum de temps, et change de file d'attente suivant son utilisation du quantum. Si un programme utilise la totalité de son quantum de temps, c'est le signe qu'il a besoin d'un quantum plus gros et doit donc changer de file. La seule façon pour lui de remonter d'un niveau est de ne plus utiliser complètement son quantum de temps. Ainsi, un programme va se stabiliser dans la file dont le quantum de temps est optimal (ou presque).
- **L'ordonnancement par loterie** repose sur un ordonnancement aléatoire. Chaque processus se voit attribuer des tickets de loterie : pour chaque ticket, le processus a droit à un quantum de temps. À chaque fois que le processeur change de processus, il tire un ticket de loterie et le programme qui a ce ticket est alors élu. Évidemment, certains processus peuvent être prioritaires sur les autres : ils disposent alors de plus de tickets de loterie que les autres. Dans le même genre, des processus qui collaborent entre eux peuvent échanger des tickets. Comme quoi, il y a même moyen de tricher avec de l'aléatoire...

Création et destruction de processus

Les processus peuvent naître et mourir. La plupart des processus démarre quand l'utilisateur demande l'exécution d'un programme, n'importe quel double-clic sur une icône d'exécutable en est un bon exemple. Mais, le démarrage de la machine en est un autre exemple. En effet, il faut bien lancer le système d'exploitation, ce qui demande de démarrer un certain nombre de processus. Après tout, vous avez toujours un certain nombre de services qui se lancent avec le système d'exploitation et qui tournent en arrière-plan. A l'exception du premier processus, lancé par le noyau, les processus sont toujours créés par un autre processus. On dit que le processus qui les a créés est le processus parent. Les processus créés par le parent sont appelés processus enfants. On parle aussi de père et de fils. Certains systèmes d'exploitation conservent des informations sur qui a démarré quoi et mémorisent ainsi qui est le père ou le fils pour chaque processus. L'ensemble forme alors une **hiérarchie de processus**.

Sur la majorité des systèmes d'exploitation, un appel système suffit pour créer un processus. Mais certains systèmes d'exploitation (les unixoides, notamment) ne fonctionnent pas comme cela. Sur ces systèmes, la création d'un nouveau processus est indirecte : on doit d'abord copier un processus existant avant de remplacer son code par celui d'un autre programme. Ces deux étapes correspondent à deux appels systèmes différents. Dans la réalité, le système d'exploitation ne copie pas vraiment le processus, mais utilise des optimisations pour faire comme si tout se passait ainsi.

Si les processus peuvent être créés, il est aussi possible de les détruire quand ils ont terminé leur travail ou s'ils commettent une erreur/faute lors de leur exécution. Dans tous les cas, le processus père est informé de la terminaison du processus fils par le système d'exploitation. Le processus fils passe alors dans un mode appelé mode zombie : il attend que son père prenne connaissance de sa terminaison. Lorsque le processus se termine, le système récupère tout ce qu'il a attribué au processus (PCB, espaces mémoire, etc.), mais garde une trace du processus dans sa table des processus. C'est seulement lorsque le père prend connaissance de la mort de son fils qu'il supprime l'entrée du fils dans la table des processus. Si un processus père est terminé avant son fils, le processus qui est le père de tous qui « adopte » le processus qui a perdu son père et qui se chargera alors de le « tuer ».

Communication inter-processus

Comme on l'a vu plus haut, les processus sont chargés dans des espaces mémoire dédiés où seuls eux peuvent écrire. On parle d'**isolation des processus**. Il faut cependant noter que tous les systèmes d'exploitation n'implémentent pas cette isolation des processus, MS-DOS en étant un exemple. Faire communiquer des processus entre eux demande d'utiliser des mécanismes de communication inter-processus. La méthode la plus simple consiste à partager un bout de mémoire entre processus, mais il existe d'autres méthodes que nous allons aborder.

Avec isolation des processus

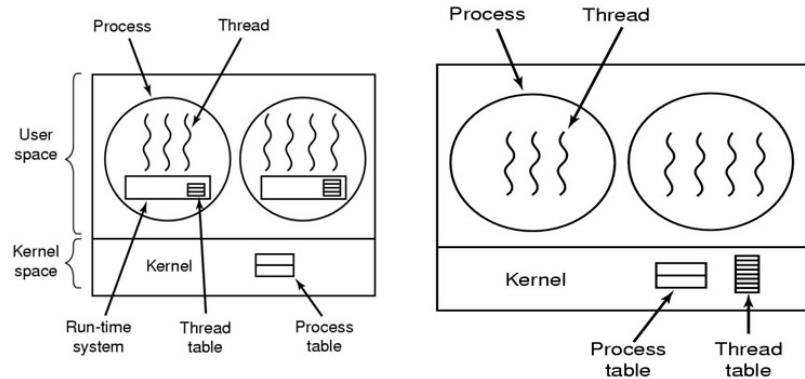
La première méthode est appelée l'**échange de messages**. Il permet aux processus de s'échanger des **messages**, des blocs de données de taille variable ou fixe selon l'implémentation. Mais le contenu n'est pas standardisé, et chaque processus peut mettre ce qu'il veut dans un message. Le tout est que le processus qui reçoit le message sache l'interpréter. Pour cela, le format des données, le type du message, ainsi que la taille totale du message, est souvent envoyé en même temps que le message.

Dans le cas le plus simple, il n'y a pas de mise en attente des messages dans un espace de stockage partagé entre processus. On parle alors de **passage de message**. Cela fonctionne bien quand les deux processus ne sont pas sur le même ordinateur et communiquent entre eux via un réseau local ou par Internet. Mais certains O.S permettent de mettre en attente les messages envoyés d'un processus à un autre, le processus récepteur pouvant consulter les messages en attente quand celui-ci est disponible. Les messages sont alors mis en attente dans diverses structures de mémoire partagée. Celles-ci sont appelées, suivant leur fonctionnement, des **tubes** ou des **files de messages**. Ces structures stockent les messages dans l'ordre dans lequel ils ont été envoyés (fonctionnement en file ou FIFO). La différence entre les deux est que les tubes sont partagés entre deux processus, un pouvant lire et un autre écrire, alors qu'une file de message peut être partagée entre plusieurs processus qui peuvent aussi bien lire qu'écrire.

Sans isolation des processus

Il est possible de rassembler plusieurs programmes dans un seul processus. Ces programmes n'étant pas isolés, ils se partagent le même morceau de mémoire et peuvent ainsi communiquer entre eux via une mémoire partagée. Les programmes portent alors le nom de **threads**. Ces threads doivent aussi être ordonnancés. On fait ainsi la distinction entre threads proprement dits, gérés par un ordonnancement préemptif, et les fibers, gérés par un ordonnancement collaboratif. Le changement de contexte entre deux threads est beaucoup plus rapide que pour les processus, vu que cela évite de devoir faire certaines manipulations obligatoires avec les processus. Par exemple, on n'est pas obligé de vider le contenu des mémoires caches sur certains processeurs. Généralement, les threads sont gérés en utilisant un ordonnancement préemptif, mais certains threads utilisent l'ordonnancement collaboratif (on parle alors de **fibers**).

Les **threads utilisateurs** sont des threads qui ne sont pas liés au système d'exploitation. Ceux-ci sont gérés à l'intérieur d'un processus, par une bibliothèque logicielle. Celle-ci s'occupe de la création et la suppression des threads, ainsi que de leur ordonnancement. Le système d'exploitation ne peut pas les ordonner et n'a donc pas besoin de mémoriser les informations des threads. Par contre, chaque thread doit se partager le temps alloué au processus lors de l'ordonnancement : c'est dans un quantum de temps que ces threads peuvent s'exécuter. Les **threads noyaux** sont gérés par le système d'exploitation, qui peut les créer, les détruire ou les ordonner. L'ordonnancement est donc plus efficace, vu que chaque thread est ordonné tel quel. Il est donc nécessaire de disposer d'une table des threads pour mémoriser les contextes d'exécution et les informations de chaque thread.



Les systèmes de fichiers

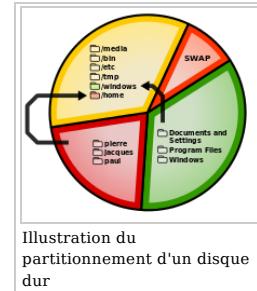
Le **système de fichiers** est la portion du système d'exploitation qui s'occupe de la gestion des mémoires de masse. Il prend en charge le stockage des fichiers sur le disque dur, le rangement de ceux-ci dans des répertoires, l'ouverture ou la fermeture de fichiers/répertoires, et bien d'autres choses encore. La gestion des partitions est aussi assez liée au système d'exploitation.

Partitions

Avant d'installer un système d'exploitation sur un disque dur, celui-ci doit être partitionné, du moins sur les PC actuels. Partitionner un disque dur signifie le diviser en plusieurs morceaux, qui seront considérés par le système d'exploitation comme autant de disques durs séparés. Les informations sur les partitions sont mémorisées dans une **table des partitions**, qui stocke la taille et le premier secteur de chaque partition, ainsi que quelques informations complémentaires.

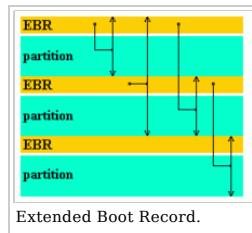
Master Boot Record

Sur les PC actuels, on a vu que cette table est stockée dans le Master Boot Record. Celui-ci peut contenir 4 partitions. Les informations de chaque partition sont codées sur 16 octets, de la manière indiquée ci-dessous. La partition correspond évidemment à un bloc de secteurs consécutifs. Vu que la taille d'une partition est codée sur 32 bits, la taille d'une partition ne peut pas dépasser les 4 Giga-secteurs. Sur les mémoires de masse qui ont des secteurs de 512 Kio, cela correspond à une capacité totale de 2,2 Téraoctets.



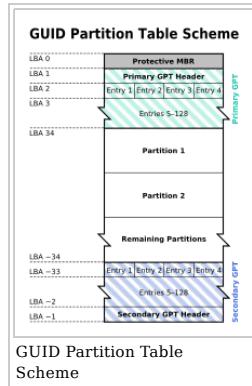
Flag qui indique si la partition est bootable ou non	Adresse CHS du début de la partition	Type de la partition	Adresse CHS de la fin de la partition	Adresse LBA du début de la partition	Taille de la partition en nombre de secteurs
1 octets	3 octets	1 octets	3 octets	4 octets	4 octets

Afin de dépasser la limite de 4 partitions, il est possible de subdiviser une partition en sous-partitions, appelées **partitions logiques**. Pour cela, la partition subdivisée doit être définie comme une partition dite étendue dans le MBR (le type de la partition doit avoir une valeur bien précise). Les informations sur une partition logique sont mémorisées dans un secteur situé au début de la partition logique, dans une structure appelée **Extended Boot Record**. Celui-ci a une structure similaire au MBR, à quelques détails près. Premièrement, seules les deux premières entrées de la table des partitions sont censées être utilisées. La première indique l'adresse de la prochaine partition logique (ce qui fait que les EBR forment une liste chaînée), tandis que la seconde sert pour les informations de la partition logique proprement dite. De plus, la place normalement réservée au chargement de système d'exploitation est remplie de zéros.



Tables GUID

Afin de faire face aux limitations de ce système, le MBR est progressivement remplacé par des **table de partitionnement GUID**. Celle-ci permet de gérer des partitions de plus de 2,2 Téra-octets, sans problèmes. Avec elle, la table des partitions peut contenir 32 partitions différentes. La table des partitions contient donc 32 entrées, une pour stocker les informations sur chaque partition. Ces entrées sont précédées par un en-tête, qui contient des informations générales permettant au système de fonctionner. Cet en-tête est lui-même précédé par un MBR tout ce qu'il y a de plus normal, pour des raisons de compatibilité.



Les adresses de chaque partition sont des adresses LBA codées sur 8 octets (64 bits), ce qui permet de gérer des partitions plus grandes qu'avec le MBR. Voici à quoi ressemble une entrée de la table des partitions.

Type de la partition	Identifiant GUID	Première adresse LBA	Dernière adresse LBA	Attributs de la partition	Nom de la partition
16 octets	16 octets	8 octets	8 octets	8 octets	72 octets

Fichiers

Les données sont organisées sur le disque dur sous la forme de **fichiers**. Ce sont généralement de simples morceaux d'une mémoire de masse, sur lesquels un programme peut écrire ce qu'il veut. Le système de fichiers attribuent plusieurs caractéristiques à chaque fichier.

- Chaque fichier reçoit un **nom de fichier**, qui permet de l'identifier et de ne pas confondre les fichiers entre eux. L'utilisateur peut évidemment

renommer les fichiers, choisir le nom des fichiers, et d'autres choses dans le genre.

- En plus du nom, le système d'exploitation peut mémoriser des informations supplémentaires sur le fichier : la date de création, la quantité de mémoire occupée par le fichier, et d'autres choses encore. Ces informations en plus sont appelées des **attributs de fichier**.
- Il existe des normes qui décrivent comment les données doivent être rangées dans un fichier, suivant le type de données à stocker : ce sont les **formats de fichiers**. Le format d'un fichier est indiqué à la fin du nom du fichier par une **extension de fichier**. Généralement, cette extension de nom commence par un ".", suivi d'une abréviation. Par exemple, le .JPEG, le .WAV, le .MP3 ou le .TXT sont des formats de fichiers. Ces formats de fichiers ne sont pas gérés par le système d'exploitation. Tout ce que peut faire l'OS, c'est d'attribuer un format de fichier à une application : il sait qu'un .PDF doit s'ouvrir avec un lecteur de fichiers .PDF, par exemple.

Clusters

Pour rappel, toutes les mémoires de masse sont découpées en secteurs de taille fixe, qui possèdent tous une adresse. Les systèmes de fichiers actuels ne travaillent pas toujours sur la base des secteurs, mais utilisent des **clusters**, des groupes de plusieurs secteurs consécutifs. L'adresse d'un cluster est celle de son premier secteur. La taille d'un cluster dépend du système de fichiers utilisé, et peut parfois être configuré. Par exemple, Windows permet d'utiliser des tailles de clusters comprises entre 512 octets et 64 kilooctets.

La taille idéale des clusters dépend du nombre de fichiers à stocker et de leur taille. Pour de petits fichiers, il est préférables d'utiliser une taille de cluster faible, alors que les gros fichiers ne voient pas d'inconvénients à utiliser des clusters de grande taille. En effet, un fichier doit utiliser un nombre entier de clusters. Illustrons la raison par un exemple. Si un fichier fait 500 octets, il utilisera un cluster complet : seuls les 500 octets seront utilisés, les autres étant tout simplement inutilisés tant que le fichier ne grossit pas. Si le cluster a une taille de 512 octets, seuls 2 octets seront gâchés. Mais avec une taille de cluster de 4 kilooctets, la grosse majorité du cluster sera inoccupé. La même chose a lieu quand les fichiers prennent un faible nombre de clusters. Par exemple, un fichier de 9 kilooctets n'utilisera pas d'espace inutile avec des clusters de 512 octets : le fichier utilisera 18 clusters au complet. Mais avec une taille de clusters de 4 Ko, trois secteurs seront utilisés, dont le dernier ne contiendra qu'1 Ko de données utiles, 3 Ko étant gâchés.

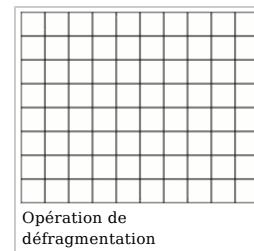
En comparaison, le débit en lecture et écriture sera nettement amélioré avec des clusters suffisamment gros. Pour comprendre pourquoi, il faut savoir que le débit d'un disque dur est le produit de deux facteurs : la taille d'une unité d'allocation (ici, un cluster) multiplié par le nombre d'**opérations disque par secondes**. Ces opérations disque correspondent à une lecture ou écriture de cluster. Chaque opération d'entrée-sortie utilise un peu de processeur, vu qu'il s'agit d'un appel système. Cela peut se résumer par la formule suivante, avec D le débit du HDD, IOPS le nombre d'opérations disque par secondes, et $T_{cluster}$ la taille d'un cluster : $D = IOPS \times T_{cluster}$. Pour un débit constant, augmenter la taille d'un cluster diminue le nombre d'opérations disque nécessaires. Le débit maximal est donc plus stable, beaucoup de disques durs ayant une limite au nombre d'opérations disque qu'ils peuvent gérer en une seconde. Expérimentalement, on constate que le débit en lecture ou écriture est meilleur avec une grande taille de cluster, du moins pour des accès à de gros fichiers, dont la lecture/écriture demande d'accéder à des secteurs/cluster consécutifs.

Allocation des secteurs

Le système d'exploitation doit mémoriser pour chaque fichier, quels sont les secteurs du HDD qui correspondent (leur adresse). Cette liste de secteurs est stockée dans une table de correspondance, située au tout début d'une partition : la **Master File Table**. Celle-ci contient, pour chaque fichier, son nom, ses attributs, ainsi que la liste de ses clusters. Avec un système de fichiers de ce type, chaque partition est décomposée en quatre portions :

- le secteur de boot proprement dit, qui contient le chargeur de système d'exploitation ;
- la **Master File Table**, qui contient la liste des fichiers et leurs clusters ;
- et enfin, les fichiers et répertoires contenus dans le répertoire racine.

Reste que lors de la création ou de l'agrandissement d'un fichier, l'OS doit lui allouer un certain nombre de secteurs. Pour cela, il existe diverses méthodes d'allocation. La première méthode est celle de l'**allocation contiguë**. Elle consiste à trouver un bloc de secteurs consécutifs capable d'accepter l'ensemble du fichier. Repérer un fichier sur le disque dur demande simplement de mémoriser le premier secteur, et le nombre de secteurs utilisés. Cette méthode a de nombreux avantages, notamment pour les performances des lectures et des écritures à l'intérieur d'un fichier. C'est l'ailleurs la voie royale pour le stockage de fichiers sur les CD-ROM ou les DVD, où les fichiers sont impossible à supprimer. Mais sur les disques durs, les choses changent. A force de supprimer ou d'ajouter des fichiers de taille différentes, on se retrouve avec des blocs de secteurs vides, coincés entre deux fichiers proches. Ces secteurs sont inutilisables, vu que les fichiers à stocker sont trop gros pour rentrer dedans. Une telle situation est ce qu'on appelle la **fragmentation**. Le seul moyen pour récupérer ces blocs est de compacter les fichiers, pour récupérer l'espace libre : c'est l'opération de défragmentation.



Sur d'autres systèmes de fichiers, les clusters d'un fichier ne sont donc pas forcément consécutifs sur le disque dur. Le système d'exploitation doit garder, pour chaque fichier, la liste des clusters qui correspondent à ce fichier. Cette liste est ce qu'on appelle une liste chaînée : chaque cluster indique quel est le cluster suivant (plus précisément, l'adresse du cluster suivant). L'accès à un fichier se fait cluster par cluster. L'accès à un cluster bien précis demande de parcourir le fichier depuis le début, jusqu'à tomber sur le cluster demandé. Avec cette méthode, la défragmentation reste utile sur les HDD, vu que l'accès à des secteurs consécutifs est plus rapide.

On peut améliorer la méthode précédente, en regroupant toutes les correspondances (cluster -> cluster suivant) dans une table en mémoire RAM. Cette table est appelée la FAT, pour **File Allocation Table**. Avec cette méthode, l'accès aléatoire est plus rapide : parcourir cette table en mémoire RAM, est plus rapide que de parcourir le disque dur. Malheureusement, cette table a une taille assez imposante pour des disques durs de grande capacité : pour un disque dur de 200 gibioctets, la taille de la FAT est de plus de 600 mégobits.

Une autre méthode consiste à ne pas stocker les correspondances (cluster -> cluster suivant), mais simplement les adresses des clusters les unes à la suite des autres : on obtient ainsi un **i-nœud**. C'est cette méthode qui est utilisée dans les systèmes d'exploitation UNIX et leurs dérivés (Linux, notamment).

Répertoires

Sur les premiers systèmes d'exploitation, on ne pouvait pas ranger les fichiers : tous les fichiers étaient placés sur le disque dur sans organisation. De nos jours, tous les systèmes d'exploitation gèrent des **répertoires**. Ceux-ci sont représentés sur le disque dur par des fichiers, qui contiennent des liens vers les fichiers contenus dans le répertoire. Le fichier répertoire mémorise aussi toutes les informations concernant les fichiers : leur nom, leur extension, leur taille, leurs attributs, etc. Ces informations sont rarement stockées dans le fichier lui-même. Ainsi, quand vous ouvrez un répertoire ou que vous consultez les propriétés d'un fichier, ce fichier n'est jamais ouvert : ces informations sont récupérées dans le répertoire.

Codage des répertoires

Un répertoire contient une liste de fichier. Pour chaque fichier, il mémorise son nom, ses attributs, ainsi que la liste des clusters qui lui sont attribués. Les attributs sont stockés dans une structure de taille fixe, le nombre des attributs par fichier étant connu à l'avance. Mais la liste des clusters et les noms des fichiers n'ont pas de taille fixée à l'avance. Cela ne pose pas de problème pour la liste des clusters, contrairement à ce qu'on a pour les noms. Généralement, les répertoires mémorisent les noms des fichiers qu'ils contiennent, avec leurs attributs. Ceux-ci sont donc placées dans un bloc de taille fixe, un en-tête. Reste que la gestion des noms de fichiers peut se faire de plusieurs manières.

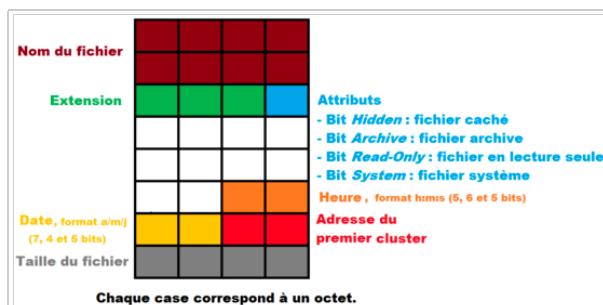
Avec la première méthode, on limite la taille des noms de fichiers. On peut alors réservé une taille suffisante dans le fichier répertoire pour stocker ce nom. Par exemple, sur les premiers systèmes de fichiers FAT, les noms de fichiers étaient limités à 8 caractères (plus 3 pour l'extension de fichier). Le répertoire allouait 8 octets pour chaque nom de fichier, plus 3 octets pour l'extension. La taille utilisée était alors limitée. Cette

méthode, bien que simple d'utilisation, a tendance à gâcher de la mémoire si de grands noms de fichiers sont utilisés. Et si de petits noms de fichier, l'espace utilisé sera faible, mais les utilisateurs pourront se sentir limités par la taille maximale des noms de fichier.

On peut aussi utiliser des noms de fichiers de taille variable. La solution la plus simple consiste à placer ceux-ci à la suite de l'en-tête (qui contient les attributs). Cette méthode a cependant un désavantage lors de la suppression du fichier : elle génère de la fragmentation assez facilement. Pour éviter cela, on peut placer les noms de fichier de taille variable à la fin du fichier répertoire, chaque en-tête pointant vers le nom de fichier adéquat (Linux).

Hiérarchie de répertoires

Sur les premiers systèmes d'exploitation, tous les fichiers étaient placés dans un seul et unique répertoire. Quand le nombre de fichier était relativement faible, cela ne posait pas trop de problèmes. L'avantage de cette méthode était que le système d'exploitation était plus simple et qu'il n'avait pas à gérer une arborescence de répertoire sur le disque dur. Mais l'inconvénient se faisait sentir quand les fichiers devenaient de plus en plus nombreux. Cette organisation est encore utilisée sur les baladeurs et appareils photo numériques.

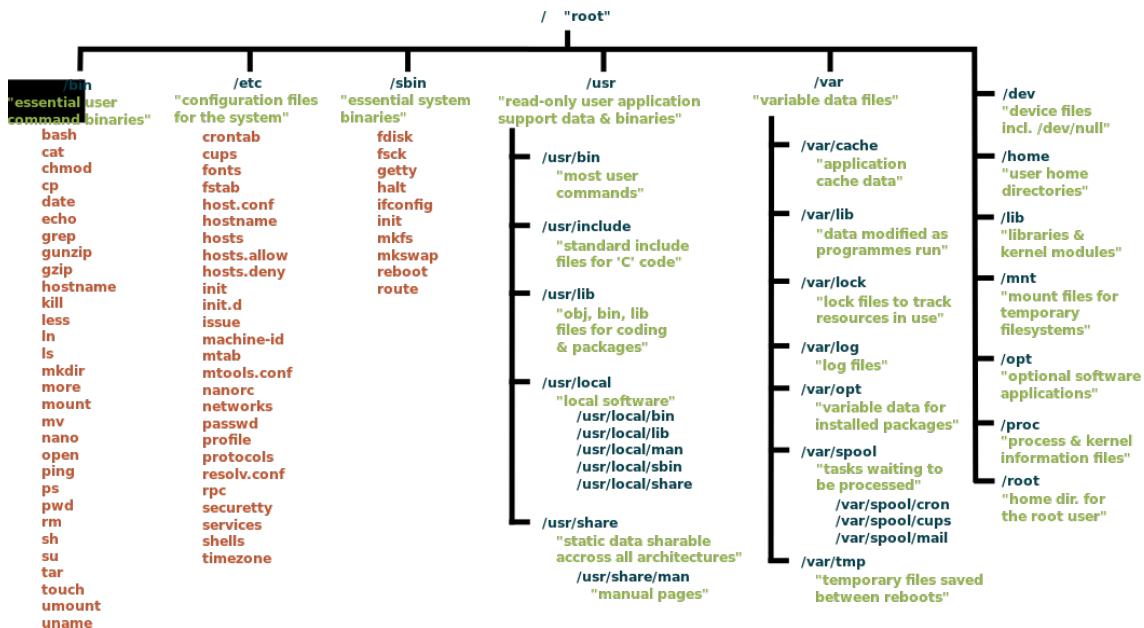


Ce schéma illustre l'entrée d'un fichier (dans un répertoire) telle qu'elle est codée avec les systèmes de fichier FAT 12 et FAT 16. La FAT-32 utilise un système quasiment identique, sauf pour l'adresse du premier cluster, qui est codée sur 4 octets.

De nos jours, les répertoires sont organisés en une **hiérarchie de répertoire** : un répertoire peut contenir d'autres répertoires, et ainsi de suite. Évidemment, cette hiérarchie commence par un répertoire maître, tout en haut de cette hiérarchie : ce répertoire est appelé la **racine**. Sous Windows, on trouve une seule et unique racine par partition ou disque dur : on en a une sur C :, une autre sur D :, etc. Sous Linux, il n'existe qu'une seule racine, chaque partition étant un répertoire accessible depuis la racine : ces répertoires sont appelés des **points de montage**. Sur Windows, ce répertoire est noté : soit \, soit nom_lecteur :\ (C:\, D:\). Sous Linux, il est noté : / .

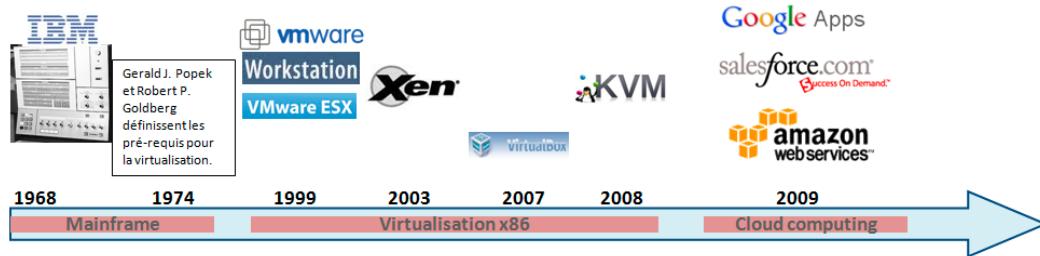
Reste que savoir où se trouve un fichier demande de préciser quel est le chemin qu'il faut suivre en partant du répertoire maître : on doit préciser qu'il faut ouvrir tel répertoire, puis tel autre, et ainsi de suite jusqu'au fichier. Ce chemin, qui reprendre le chemin de la racine jusqu'au fichier, est appelé le **chemin d'accès absolu**. Sous Windows, les noms de répertoires sont séparés par un \ : C:\Program Files\Internet Explorer. Sous Linux, les noms de répertoires sont séparés par un / : /usr/bin/courrier. Les **chemins d'accès relatifs** sont similaires, sauf qu'ils ne partent pas de la racine : ils partent d'un répertoire choisi par l'utilisateur, nommé répertoire courant. Dans les grandes lignes, l'utilisateur ou un programme choisissent un répertoire et lui attribuent le titre de répertoire courant. Généralement, chaque processus a son propre répertoire de travail, qu'il peut changer à loisir. Pour cela, le programme doit utiliser un appel système, chdir sous Linux.

Linux a une particularité : il normalise la place de certains répertoires bien précis du système d'exploitation. L'arborescence des répertoires est en effet standardisé par le **Filesystem Hierarchy Standard**, la dernière version datant de juin 2015.



Virtualisation et machines virtuelles

Grâce aux différentes technologies de virtualisation, il est possible de lancer plusieurs systèmes d'exploitation en même temps sur le même ordinateur. Il est ainsi possible de passer d'un O.S à un autre relativement rapidement, suivant les besoins, avec un simple raccourci clavier. Vous avez certainement déjà eu l'occasion d'utiliser VMWARE ou un autre logiciel de virtualisation. Ces technologies sont aussi beaucoup utilisées sur les serveurs, entre autres.

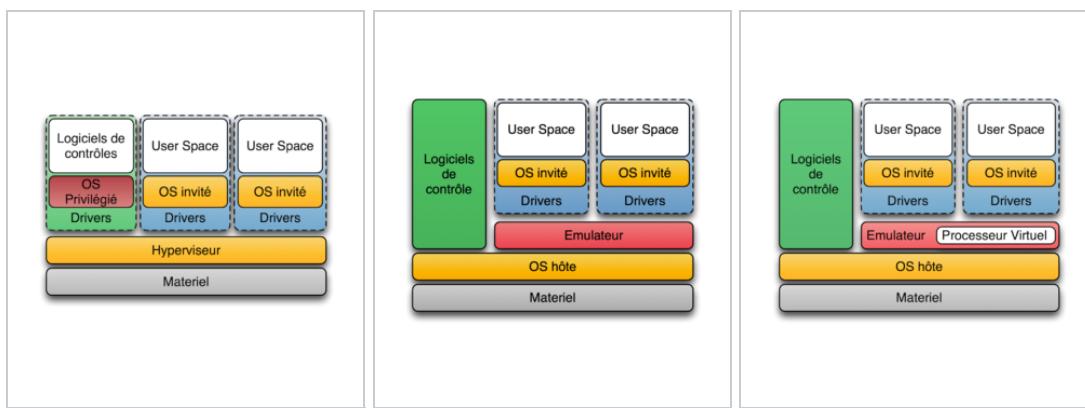


Machine virtuelle

Ces logiciels implémentent ce qu'on appelle des **machines virtuelles**. Il s'agit d'une sorte de faux matériel, garantie par un logiciel. Un logiciel qui s'exécute dans une machine virtuelle aura l'impression de s'exécuter sur un matériel et/ou un O.S différent du matériel sur lequel il est en train de s'exécuter.

Ces machines virtuelles sont utilisés aussi bien pour les techniques de virtualisation que sur ce qu'on appelle les **émulateurs**, des logiciels qui permettent de simuler le fonctionnement d'anciens ordinateurs ou consoles de jeux. Mais les techniques de virtualisation fonctionnent un peu différemment de l'éulation de consoles de jeux. En effet, un émulateur a besoin d'émuler le processeur, qui est généralement totalement différent entre le matériel émulé et le matériel sur lequel s'exécute la V.M (virtual machine, machine virtuelle). Ce n'est pas le cas avec la virtualisation, le jeu d'instruction étant généralement le même.

Le logiciel de virtualisation est généralement limité à un logiciel appelé **hyperviseur**. Celui-ci gère plusieurs machines virtuelles, une par O.S lancé sur l'ordinateur. Il existe différents types d'hyperviseurs, qui sont généralement appelées sous les noms techniques de virtualisation de type 1 et 2. Dans le premier cas, l'hyperviseur s'exécute directement, sans avoir besoin d'un O.S sous-jacent. Dans le second cas, l'hyperviseur est un logiciel applicatif comme un autre, qui s'exécute sur un O.S bien précis. Les émulateurs utilisent systématiquement un hyperviseur de type 2 amélioré, capable de simuler le jeu d'instruction du processeur émulé.



Hyperviseur de type 1

Hyperviseur de type 2

Emulateur

Une méthode annexe à la virtualisation de type 2 consiste à modifier l'O.S à virtualiser de manière à ce qu'il puisse communiquer directement avec l'hyperviseur. On parle alors de **paravirtualisation**. Avec cette méthode, les appels systèmes des O.S sont remplacés par des appels systèmes implémentés par l'hyperviseur.

Certains processeurs disposent de technologies matérielles d'accélération de la virtualisation. Sur les processeurs x86, on peut citer l'Intel VTx ou l'AMD-V.

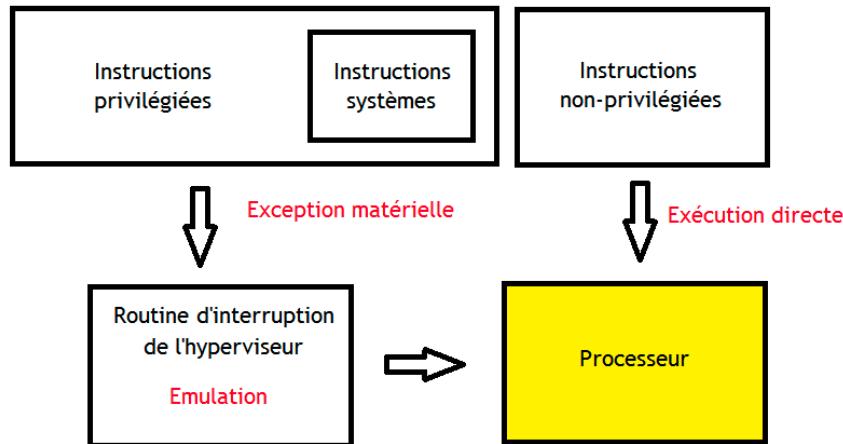
Fonctionnement/implémentation

On peut se demander comment les technologies de virtualisation et les émulateurs font pour simuler une machine virtuelle. Pour être considéré comme un logiciel de virtualisation, un logiciel doit remplir trois critères :

- L'équivalence : l'O.S virtualisé et les applications qui s'exécutent doivent se comporter comme s'ils étaient exécutés sur le matériel de base, sans virtualisation.
- L'efficacité : La grande partie des instructions machines doit s'exécuter directement sur le processeur, afin de garder des performances correctes. Ce critère n'est pas respecté par les émulateurs matériels, qui doivent simuler le jeu d'instruction du processeur émulé.
- Le contrôle des ressources : tout accès au matériel par l'O.S virtualisé doit être intercepté par la machine virtuelle et intégralement pris en charge par l'hyperviseur.

Remplir ces trois critères est possible sous certaines conditions, établies par la théorie de la virtualisation de Popek et Goldberg. Ceux-ci ont réussi à établir plusieurs théorèmes sur la possibilité de créer un hyperviseur pour une architecture matérielle quelconque. Ces théorèmes se basent sur la présence de différents types d'instructions machines. Pour faire simple, on peut distinguer les **instructions systèmes**, qui agissent sur le matériel. Il s'agit d'instructions d'accès aux entrées-sorties, aussi appelées instructions sensibles à la configuration, et d'instructions qui reconfigurent le processeur, aussi appelées instructions sensibles au comportement. Certaines instructions sont exécutables uniquement si le processeur est en mode noyau. Si ce n'est pas le cas, le processeur considère qu'une erreur a eu lieu et lance une exception matérielle. Ces instructions sont appelées des **instructions privilégiées**. On peut considérer qu'il s'agit d'instructions que seul l'O.S peut utiliser. A côté, on trouve des instructions **non-privilégiées** qui peuvent s'exécuter aussi bien en mode noyau qu'en mode utilisateur.

La théorie de Popek et Goldberg dit qu'il est possible de virtualiser un O.S à une condition : que les instructions systèmes soient toutes des instructions privilégiées. Si c'est le cas, virtualiser un O.S signifie simplement le démarrer en mode utilisateur. Quand l'O.S exécutera un accès au matériel, il le fera via une instruction privilégiée. Ce faisant, celle-ci déclenchera une exception matérielle, qui sera traitée par une routine d'interruption spéciale. L'hyperviseur n'est ni plus ni moins qu'un ensemble de routines d'interruptions, chaque routine simulant le fonctionnement du matériel émulé. Par exemple, un accès au disque dur sera émulé par une routine d'interruption, qui utilisera les appels systèmes fournis par l'OS pour accéder au disque dur réellement présent dans l'ordinateur. Cette méthode est souvent appelée la méthode trap-and-emulate.



Mais le critère précédent n'est pas respecté par beaucoup de jeux d'instructions. Par exemple, ce n'est pas le cas sur les processeurs x86, qu'ils soient 32 ou 64 bits. Diverses instructions systèmes ne sont pas des instructions privilégiées. La solution est simplement de traduire à la volée les instructions systèmes (privilégiées ou non-privilégiées) en appels systèmes équivalents. Cette technique est appelée la **réécriture de code**.



Vous avez la permission de copier, distribuer et/ou modifier ce document selon les termes de la **licence de documentation libre GNU**, version 1.2 ou plus récente publiée par la Free Software Foundation ; sans sections inaltérables, sans texte de première page de couverture et sans texte de dernière page de couverture.

Récupérée de « https://fr.wikibooks.org/w/index.php?title=Fonctionnement_d'un_ordinateur/Version_imprimable&oldid=527662 »

Dernière modification de cette page le 28 septembre 2016, à 16:06.
Les textes sont disponibles sous licence Creative Commons attribution partage à l'identique ; d'autres termes peuvent s'appliquer.
Voyez les termes d'utilisation pour plus de détails.