Architecture des circuits
Architecture des ordinateurs

Protopoly

Florent de Dinechin
avec des figures de
R. Bergasse, N. Bonifas, N. Brunie, P. Boutillier,
A. Derouet-Jourdan, J. Detrey, A. Friggeri, B. Grenet, F. Givors,
T. Jolivet, C. Keller, E. Lassalle, P.E. Meunier,
P. Robert, G. Salagnac, O. Schwander, T. Risset, P. Vannier

INSA | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Table des matières

(*) à peine survolé en cours, l'examen ne portera pas dessus, mais je vous encourage à le lire.

Chapitre 1

Introduction

L'informatique c'est la science du traitement de l'information.

Un ordinateur est une *machine universelle de traitement de l'information*. Universelle veut dire : qui peut réaliser toute transformation d'information *réalisable*.

Il existe des calculs non réalisables, et des calculs non réalisables en temps raisonnable. C'est pas que ce n'est pas important, mais on ne s'y intéresse pas dans ce cours.

Un bon bouquin de support de ce cours est *Computer Organization & Design, the hardware/software interface*, de Patterson et Hennessy. Il y a aussi *Architecture de l'Ordinateur*, 74ème édition (au moins) par Tanenbaum, il existe en français. Mon préféré du moment est *Computer Architecture and Organisation* de Murdocca et Heuring, il devrait arriver un jour à la bibliothèque.

1.1 Historique du calcul mécanique

néolithique Invention du système de numération unaire, de l'addition et de la soustraction (des moutons)
(en latin, *calculus* = petit caillou)

antiquité Systèmes de numération plus évolués :

systèmes alphabétiques (Égypte, Grèce, Chine) :
chaque symbole a une valeur numérique fixe et indépendante de sa position.
Les chiffres romains sont un mélange de unaire et d'alphabétique.

systèmes à position (Babylone, Inde, Mayas) :
c'est la position du chiffre dans le nombre qui donne sa puissance de la base.

Les bases utilisées sont la base 10 (Égypte, Inde, Chine), la base 20 (Mayas), la base 60 (Sumer, Babylone), ...

Ces inventions sont guidées par la nécessité de faire des calculs

Essayez donc de décrire l'algo de multiplication en chiffres romains...Par contre la base 60 c'est bien pratique, pourquoi ?

≈**-1000** Invention du calculateur de poche (l'abaque ou boulier) en Chine.

+1202 Le génial système de numération indo-arabe arrive en Europe par l'Espagne.

1623 Sir Francis Bacon (Angleterre) décrit le codage des nombres dans le système binaire qu'il a inventé dans sa jeunesse.

1623 Wilhelm Schickard (Tübingen) invente la roue à chiffres qui permet de propager les retenues.

1624 Il construit la première calculette "occidentale"

1645 Blaise Pascal en fabrique une mieux qui peut propager les retenues sur les grands nombres.

1672 Gottfried Wilhelm Leibniz construit une machine à calculer 4 opérations

1679 Leibniz encore développe l'arithmétique binaire (De Progressione Dyadica) mais sans l'idée que cela pourrait servir à quelque chose.

1728 Première utilisation connue des cartes perforées (Falcon ?)

1741 Jacques de Vaucanson invente la mémoire de programme dans le contexte des métiers à tisser. Il utilise des rouleaux de fer-blanc perforés.

1808 Joseph Marie Jacquard utilise du carton perforé, c'est moins cher.

1822 Charles Babbage se lance dans sa *difference engine*, qui sert à calculer des polynômes.

1833 Karl Friedrich Gauß et Wilhelm Weber, à Göttingen, sont les précurseurs de l'internet en s'envoyant à distance sur un fil électrique du texte codé par un genre de code Morse.

1833 Babbage laisse tomber car il a une meilleur idée, l'*analytical engine*, programmable par cartes perforées, inconstruisible avec les moyens de l'époque.

1854 Georges Boole (Angleterre) met en place la logique mathématique désormais dite booléenne.

1854 Christopher L. Sholes (USA) : première machine à écrire utilisable.

1928 Ackermann montre qu'il y a des fonctions entières qui ne peuvent être calculées par un nombre fini de boucles.

1900-1935 Développements en électronique : tube cathodique, tube à vide 3 électrodes, enregistrement sur support magnétique.

1936 Thèse d'Alan M. Turing sur une machine abstraite universelle. Début de la théorie de la calculabilité.

1936 Konrad Zuse (Allemagne) construit le premier calculateur binaire (Z1 : mécanique, Z2 : à relais)

1937 John V. Atanasoff (USA) a l'idée d'utiliser le binaire pour des calculateurs.

1941 Le Z3 de Zuse est le premier calculateur (presque) universel programmable
- 1400 relais pour la mémoire, 600 pour le calcul
- 64 mots de mémoire
- arithmétique binaire en virgule flottante sur 22 bits
- programmation par ruban perforé de 8 bits
- une seule boucle

1946 L'ENIAC est le premier calculateur *électronique*, mais à part cela c'est un boulier.

1939-1945 C'est la guerre, tout le monde construit des calculateurs, surtout pour la balistique et la cryptographie.

1949 Turing et von Neumann, ensemble, construisent le premier ordinateur universel électronique, le Manchester Mark I. L'innovation c'est la mémoire partagée programme et donnée.

Depuis on n'a pas fait tellement mieux. Enfin si, mais les innovations sont difficiles à expliquer *avant* le cours d'ASR...

1.2 Objectifs du cours

On va construire un ordinateur moderne, en essayant de se concentrer sur les techniques qui sont indépendantes de la technologie (en principe on pourra construire notre ordinateur en utilisant l'électronique actuelle, mais aussi en Lego, ou bien avec des composants moléculaires opto-quantiques à nanotubes de carbone).

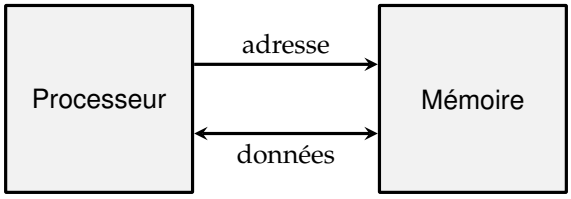


FIGURE 1.1 – S'il te plait, dessine-moi un ordinateur

C'est quoi un ordinateur moderne ? C'est la figure 1.1.

Et il y a une idée capitale cachée dans ce dessin, et que personne n'avait eu avant la bande à von Neumann : *C'est la même mémoire qui contient le programme et les données.*

C'est génial car cela permet par exemple

— le *système d'exploitation* : un programme qui prend un paquet de données sur un disque dur, les met en mémoire, et ensuite décide que ces données forment un programme, et exécute ce programme.

— le *compilateur* qui prend un texte (des données) et le transforme en un programme...

La mémoire est un ensemble de cases mémoires "type-agnostiques" : dans chaque case on met une information qui peut être interprétée n'importe comment. On attrape les cases par leur adresse.

Le processeur réalise le *cycle de von Neumann*

1. Lire une case mémoire d'adresse `PC` (envoyer l'adresse à la mémoire, et recevoir en retour la donnée à cette adresse).
2. Interpréter cette donnée comme une instruction, et l'exécuter
3. Ajouter 1 à `PC`
4. Recommencer

PC c'est pour *program counter*, bande de gauchistes.

La fréquence de votre ordinateur favori, c'est la fréquence à laquelle il exécute ce cycle.

1.3 La conception d'ASR est hiérarchique

Selon le bon vieux paradigme *diviser pour régner*, on est capable de construire l'objet très complexe qu'est votre PC par assemblage d'objets plus simples. Au passage, on utilise différents formalismes (ou différentes abstractions) pour le calcul (arithmétique binaire, fonctions booléennes, etc), pour la maîtrise des aspects temporels (mémoires, automates, etc), pour les communications (protocoles)... La figure 1.2 décrit cet empilement.

Dans ce cours on va avoir une approche de bas en haut *(bottom-up)*, mais avec des détours et des zig-zags.

Maintenant qu'on a une recette pour faire des systèmes complexes, voyons les limites pratiques à cette complexité.

1.4 Quelques ordres de grandeur

1.4.1 L'univers est notre terrain de jeu

Il y a quelques limites aux ordinateurs qu'on saura construire, par exemple :

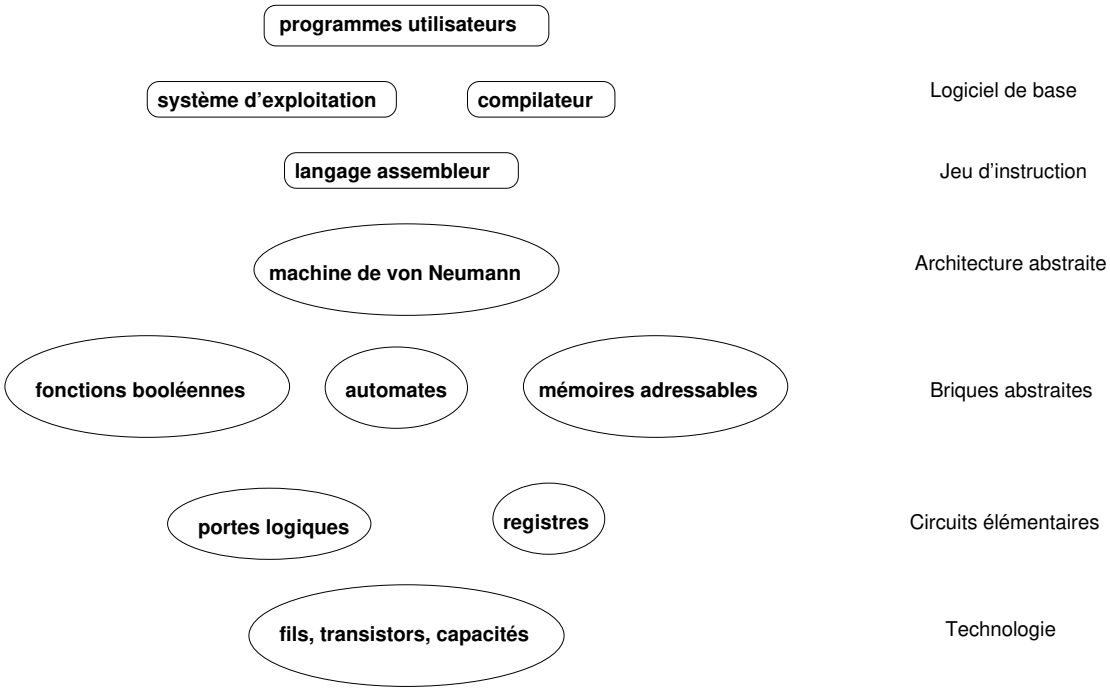


FIGURE 1.2 – *Top-down* ou *bottom-up* ?

— $\frac{\text{Masse de l'univers}}{\text{Masse du proton}} \approx 10^{78}$
(ce qui limite le nombre de transistors sur une puce. Actuellement on est à 10^{10}, il y a de la marge).
— Vitesse de la lumière : 3.10^8m/s
— Distance entre deux atomes : $\approx 10^{-10}$ m.
Taille de la maille du cristal de silicium : $\approx 5 \cdot 10^{-10}$ m
(Actuellement, un fil ou un transistor font quelques dizaines d'atomes de large : en 2013, on vend de processeurs "en technologie 22 nm", donc une quarantaine de mailles de cristal. Le problème c'est plus les couche d'isolant utilisées, actuellement entre 2 et 3 atomes d'épaisseur)
— Par conséquent, le temps minimum physiquement envisageable pour communiquer une information est de l'ordre de $\approx 10^{-18}$s.
— On a fait plus de la moitié du chemin, puisque nos transistors commutent à des fréquences de l'ordre de 10^{12} Hz.
— Actuellement, un signal n'a pas le temps de traverser toute la puce en un cycle d'horloge.
— Actuellement, on manipule des charges avec une résolution correspondant à 200 électrons. Cela permet à votre téléphone portable de gérer des fréquences au Hz près dans les 600MHz

1.4.2 La technologie en 2014

Les ordinateurs actuels sont construits en assemblant des transistors (plein [1]). La figure 1.4.2 montre les progrès de l'intégration des circuits électroniques. Il y a une loi empirique, formulée par un dénommé Moore chez Fairchild dans les années 60 (après il a rejoint Intel) et jamais démentie depuis, qui décrit l'augmentation exponentielle de la quantité de transistors par puce (la

1. A partir de 2004 on a produit plus de transistors par an dans le monde que de grains de riz. Je ne sais pas si c'est une bonne nouvelle.

taille de la puce restant à peu près constante, de l'ordre de $1cm^2$).

La formulation correcte de la loi de Moore est : le nombre de transistors qu'on sait intégrer sur une puce économiquement viable double tous les deux ans.

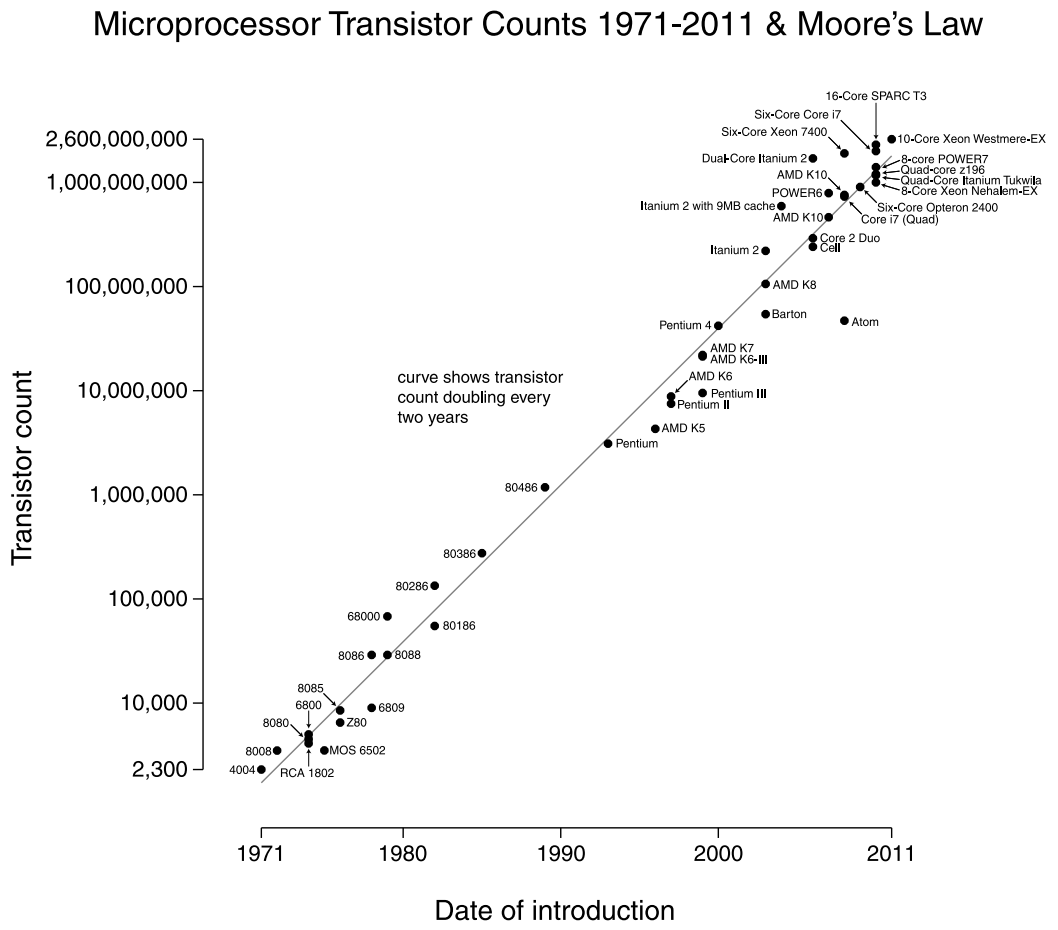


FIGURE 1.3 – La "loi" de Moore (©Wikipedia/creative commons)

Pour illustrer le chemin accompli : le premier processeur comptait 2300 transistors. De nos jours on pourrait mettre 2300 processeurs 32-bit complets dans le budget de transistors d'un pentium.

Ce doublement tous les deux ans se traduit par exemple directement par un doublement de la mémoire qu'on peut mettre dans une puce. Mais ce n'est pas tout : les transistors étant plus petits, ils sont aussi plus rapides (j'expliquerai peut-être avec les mains pourquoi, admettez en attendant). La puissance de calcul des processeurs peut donc augmenter donc plus vite que ce facteur deux tous les deux ans (les processeurs sont aussi de plus en plus complexes, ce qui tire dans l'autre sens).

Enfin, c'était vrai jusqu'aux années 2000 : en réduisant la taille du transistor, on en mettait plus sur la puce, ils étaient plus rapides, et ils consommaient moins. Avec les dernières générations (dites sub-microniques, c'est-à-dire en gros que le transistor fait moins d'un micron), cela se passe moins bien : on réduit toujours le transistor, cela permet toujours d'en mettre plus par puce, mais ils sont de moins en moins plus rapides et consomment de plus en plus (en 2007, sur les 150W que

consommait le processeur d'un PC de gamerz, il y en avait déjà 50 qui partaient en courants de fuite). Une raison facile à comprendre est qu'on arrive à des fils tellement petit que les électrons passent par effet tunnel d'un fil à l'autre. Il y a d'autres raisons.

En pratique, les experts ne donnent pas 10 ans de plus à la loi de Moore pour le silicium [2]. Heureusement il y a de la marge de progression après la fin de la loi de Moore, et c'est de l'architecture [3] (le sujet de ce cours).

Autre évolution de la technologie : l'investissement (en terme de megadollars) pour passer d'une génération technologique à la suivante double aussi avec chaque technologie... Cela se traduit par des regroupement de firmes pour partager ces coûts, et à ce train, il n'y aura plus qu'un seul fabricant de circuits intégrés dans dix ans.

Tout ceci pour dire que d'ici à ce que les plus brillants d'entre vous fassent à leur tour un cours d'architecture en Licence, la technologie sera sans doute en train de changer profondément. Vers quoi, je ne sais pas. On cherche une techno qui permette d'utiliser les 3 dimensions, et pas juste 2. Et peut-être de la transmission d'information par la lumière, qui a plein d'avantages en principe sur les déplacements d'électrons. Mais en tout cas je vais essayer de ne pas trop perdre du temps sur la technologie actuelle. Par contre je la connais assez pour répondre à vos questions.

Un dernier aspect technologique : les transistors de plus en plus petits sont de moins en moins fiables, et il faut le prévoir quand on fait des circuits. Par exemple, on évaluait en 2007 que les rayons cosmiques font statistiquement changer un bit par puce et par mois... Et cela ne s'arrange pas : plus on intègre, moins il faut d'énergie pour faire changer un bit. Ce sont des "soft errors", la puce n'est pas endommagée. Les mémoires actuelles sont munies de dispositifs d'autocorrections.

1.4.3 Calcul contre stockage et déplacement de données

D'abord, la consommation électrique est un problème, et pas que pour les téléphones portables. Le *supercomputing center* de l'UCSD consomme 1/4 de l'électricité du campus. Allez chercher sur internet combien il faut d'énergie pour faire une addition avec une calculette et avec la barre Google.

En techno 45 nm,

— 0.05 pJ pour faire une addition 32 bits
— 8 pJ pour bouger une donnée de 32 bits d'1mm sur la puce
— 320 pJ pour sortir le même mot de la puce.

Pour 50W, on peut

— faire 10^{15} additions 32 bits par seconde
— lire de la DRAM à 160Go/s, soit "seulement" 4.10^{10} mots 32bits par seconde...

D'ailleurs, sur une puce de 2cm par 2cm, on case

— 450 000 additionneurs 32 bits, ou
— 128Mo de mémoire rapide (SRAM)

Conclusion : le calcul n'est pas ce qui est coûteux. Un algorithmicien qui ne compte que les additions est un peu à côté de la plaque.

1.4.4 Ya pas que les PC dans la vie

Juste trois exemples :

Vous croyez que le processeur le plus vendu au monde est la famille x86 (pentium etc) ? Eh bien non, c'est la famille ARM, un processeur 32 bits conçu pour coûter pas cher et consommer peu d'énergie, et qui équipe la plupart des téléphones portables (sans compter les GameBoy, les organiseurs, les machines à laver, etc). Eh oui, l'arrière grand-mère de mes enfants, en Russie profonde, en est à son troisième *mobilnik* et n'aura jamais de PC. La performance d'un ARM est

2. Cela fait presque 10 ans que cette phrase est dans le poly et qu'elle reste vraie.
3. `http://www.theregister.co.uk/2016/09/22/the_evolution_of_moores_law_suggests_hardware_is_eating_software/`

comparable à celle du Pentium cinq ans avant (l'écart se réduit), sauf pour la consommation qui est divisée par 1000.

Une vieille télé Philips de 2002 contient une puce (Viper2) qui doit traiter 100 GOps (10^{11} opérations par seconde). Elle contient 4 microprocesseurs programmables, 250 RAMs, 60 autres blocs de calculs divers, 100 domaines d'horloge différents, et en tout 50Mtransistors tournant à 250 MHz (source : M. Duranton, Philips, Euromicro DSD 2006).

Un mobile 3G doit calculer entre 35 et 40 Gops (milliards d'opérations par seconde) pour traiter un canal de 14.4 Mbps, en mangeant moins d'1W. Pour la 4G on parle de centaines de GOps

Avec tout cela, votre Pentium tout neuf est ridicule, avec ses 90W il fait à peine quelques Gops. Bien sûr, on ne parle pas des mêmes opérations : le téléphone portable mouline de petits nombres de quelques bits, alors que le Pentium travaille sur de gros nombres de 32 ou 64 bits. De plus, le Pentium est généraliste. Cela se traduit par le fait qu'il dépense 6% de son énergie à calculer, et le reste à déplacer les données et les instructions à traiter (source : Dally et al dans *IEEE Computer* de juillet 2008, et cela n'a fait qu'empirer depuis).

Première partie

L’information et comment on la traite

Chapitre 2

Coder l'information

2.1 Généralités

2.1.1 Information et medium

La notion d'information est une notion abstraite. On appelle medium un support physique de l'information, que se soit pour la stocker (CD, journal) ou la transmettre (fil, onde radio, écran télé).

Le medium a un coût, l'information aussi, il faut bien distinguer les deux. Quand on achète un CD, on paye 1 euro le medium, et 10 euros l'information. Quand on le pirate, on économise le coût de l'information, mais si on le grave on n'économise pas le coût du medium. Quand on convertit un CD légalement acheté en mp3 sur son disque dur, on a dupliqué l'information, et on n'est pas deux fois plus riche pour autant. C'est une notion que l'industrie du disque a du mal à intégrer.

Formellement, l'information est un bien non rival. Les économistes définissent un bien *rival* comme un bien dont la jouissance par l'un exclut la jouissance par un autre (une baguette de pain, une fois qu'lle est mangée par l'un, ben y en a plus pour l'autre). Toute l'économie s'étant construite sur des biens rivaux, y intégrer une économie de l'information n'est pas évident, et à ce jour cette question n'est pas réglée.

2.1.2 Information analogique

L'information peut être discrète ou continue. La nature est pleine d'informations continues, que l'on sait enregistrer sur des supports analogiques (disque vinyle, photo argentique, etc), et transmettre par fil, par radio, etc. Les machines qui traitent ce genre d'information sont dites analogiques. Les traitements les plus courants sont *amplification* et *filtrage*.

Quelques exemples de *calculateurs analogiques* plus généraux :

— les horloges astronomiques ;
— Mr Thompson/Lord Kelvin a construit à base de roues dentées une machine à prédire les marées dans chsais plus quel port ;
— dans les années 50, il y avait à Supelec une "salle de calcul analogique" où l'on pouvait brancher des amplis ops entre eux pour simuler, plus vite, des phénomènes physiques. Par exemple, on a pu ainsi simuler tout le système de suspension de la Citroen DS avant de fabriquer les prototypes.

Ces derniers temps, on s'est rendu compte que c'est plus propre et plus facile de *discrétiser* l'information d'abord et de la traiter ensuite avec un calculateur numérique.

Jusqu'aux années 90, il ne restait d'analogique dans les circuits intégrés que les partie qui devaient produire des ondes à hautes fréquence : les téléphones mobiles de la première génération se composaient d'une puce numérique, et d'une puce analogique branchée sur l'antenne. De nos jours on peut faire un téléphone en une seule puce : les transistors actuels commutent tellement

vite qu'on peut tout faire en numérique. Un avantage supplémentaire est que les composants analogiques ne peuvent pas être miniutarisés autant qu'on veut : dans la surface typique d'une inductance, on peut mettre 4 processeurs ARM complets en techno de 2013.

2.1.3 Information numérique

Si elle est discrète, l'unité minimale d'information est le *bit* (contraction de *binary digit*), qui peut prendre juste deux valeurs, notées 0 ou 1 (mais vous pouvez les appeler vrai et faux, ou $\emptyset$ et $\{\emptyset\}$, ou yin et yang si cela vous chante) [1].

Pour coder de l'information utile on assemble les bits en vecteurs de bits. Sur n bits on peut coder 2^n informations différentes.

Une information complexe sera toujours un vecteur de bits. Le code est une relation entre un tel vecteur et une grandeur dans un autre domaine.

Le précepte qui reviendra tout le temps, c'est : **un bon code est un code qui permet de faire facilement les traitements utiles**.

2.1.4 Coder le temps

Le temps physique est continu, on peut le discrétiser. Parmi les instruments de mesure du temps, les plus anciens utilisent une approche analogique du temps continu (clepsydre, sablier). Les plus récents utilisent une approche numérique/discrète : horloge à balancier, à quartz... Ici aussi, le passage au monde discret (par le balancier) permet ensuite de transformer cette information sans perte : les roues dentées de l'horloge ont un rapport qui est une pure constante mathématique, ce qui permet de construire un dispositif dans lequel une heure fait toujours exactement 3600 secondes. Il ne reste dans une telle horloge qu'un seul point de contact avec le monde analogique : le balancier, qui donne la seconde. Une montre à quartz c'est pareil.

Plus pratiquement, on codera toujours un *instant* par un *changement* d'information. C'est vite dit mais il faut s'y arrêter longtemps.

2.2 Coder des nombres entiers

Il y a plein de manières de coder les entiers en binaire. Toute bijection de $\{0,1\}^n$ dans un ensemble d'entiers fait l'affaire.

Suivons le précepte ci-dessus. Sur les entiers, on aime avoir la relation d'ordre, et faire des opérations comme addition, multiplication...

Le codage en unaire fut sans doute le premier. On y représente un nombre n par un tas de n cailloux (ou batons, ou "1"). Le gros avantage de ce codage est que l'addition de deux nombres est réalisable par une machinerie simple qui consiste à mélanger les deux tas de cailloux. La comparaison peut se faire par une balance si les cailloux sont suffisamment identiques, sinon vous trouverez bien quelque chose. Plus près de nous, j'ai lu récemment un papier sérieux proposant des circuits *single electron counting* : il s'agit de coder n par n électrons dans un condensateur. La motivation est la même : l'addition de deux nombres consiste à vider un condensateur dans l'autre. La comparaison se ramène à une différence de potentiel.

Le gros problème du codage unaire est de représenter de grands nombres. Dans l'exemple du *single electron counting*, compte tenu des imperfections des composants utilisés, les nombres de doivent pas dépasser la douzaine si on veut être capable de les discriminer. Notre précepte nous dit que ce codage n'est pas bon pour coder les grands nombres.

On a inventé les codages de position comme par exemple celui que vous avez appris en maternelle. La version la plus simple est le codage binaire : un tuple $(x_0, x_1, ...x_n)$ représente par exemple un entier $X = \sum_{i=0}^{n-1} 2^i x_i$. Commençons par ceux-ci.

1. En français, "bit" est un peu un gros mot, c'est fâcheux. J'ai donc proposé à l'Académie Française de le remplacer par la contraction équivalente de *chiffre binaire*, ce qui donne *chibre*. Bizarrement, l'adoption officielle de ce terme traîne un peu.

2.2.1 Numération de position pour les entiers naturels

à savoir! Soit $\beta \in \mathbf{N}$, $\beta > 1$, une base. Tout $n \in \mathbf{N}$ peut être représenté de manière unique par sa *représentation positionnelle en base* β :

$$(x_{p-1}x_{p-2}\cdots x_1x_0)_\beta := \sum_{i=0}^{p-1} x_i\beta^i.$$

Les $x_i \in \{0, 1, \ldots, \beta-1\}$ sont les *chiffres* de l'écriture de n en base β, p est le nombre de chiffres nécessaires pour écrire de l'entier naturel n. On attribue un symbole à chaque chiffre : chiffres 0 et 1 en binaire, chiffres de 0 à 9 en décimal, chiffres de 0 à F en hexadécimal.

Changement de base par sommation des puissances

La définition permet directement d'effectuer des changements de base : si on connaît l'écriture $(x_{p-1}\ldots x_0)_\beta$ de n, et qu'on veut connaître cette écriture en base γ, il suffit de convertir les chiffres x_i en base γ, puis de calculer $\sum_{i=0}^{p-1} x_i\beta^i$ en base γ.

Changement de base par divisions euclidiennes

Une autre technique de changement de base est celle des divisions euclidiennes successives. Rappelez et justifiez cette méthode

Le reste dans la division euclidienne d'un entier naturel n par β donne le chiffre de poids faible dans la représentation de n en base β. En effet,

$$\begin{aligned} n &= x_{p-1}\cdot\beta^{p-1} + x_{p-2}\cdot\beta^{p-2} + \cdots + x_2\cdot\beta^2 + x_1\cdot\beta^1 + x_0, \\ &= \underbrace{\left(x_{p-1}\cdot\beta^{p-2} + x_{p-2}\cdot\beta^{p-3} + \cdots + x_2\cdot\beta^1 + x_1\right)}_{\text{quotient}}\cdot\beta + \underbrace{x_0}_{\text{reste}}, \end{aligned}$$

avec $0 \leq x_0 < \beta$.

En d'autres termes, $n \mod \beta = x_0$. On peut donc obtenir tous les chiffres de l'écriture d'un entier naturel n par des divisions euclidiennes successives, en s'arrêtant au premier quotient nul : on obtient les chiffres de poids faibles d'abord.

2.2.2 Parenthèse pratique : superposition des codages

Lorsqu'on construit un calculateur, on est souvent appelé à superposer des codages. Pas de grande théorie ici, mais une application du précepte (le bon codage c'est celui qui permet le bon traitement) à plusieurs niveaux de granularité. Quelques exemples :

— le boulier code des grands nombres en base 10, et chaque chiffre est représenté en unaire. C'est un codage très bien adapté à la technologie d'il y a 2000 ans : on peut reconnaître les chiffres d'un coup d'œil ; on peut les additionner d'un mouvement du doigt ; etc.

— Les calculettes bon marché fonctionnent également en base 10, avec chaque chiffre codé en binaire.

— Mon papier sur *single electron counting* reconnait que cela ne fonctionne que pour un nombre limité d'électrons dans un condensateur, et utilise ces condensateurs comme chiffres dans une représentation de position classique.

2.2.3 Codage des entiers relatifs

Si on veut gérer des entiers relatifs, on peut ajouter un bit qui contient l'information de signe. C'est ce que vous avez appris en 6ème. Un défaut de cette technique est que vous avez du apprendre deux algorithmes complètement différents pour l'addition et la soustraction.

Nous introduisons ici un codage un peu plus zarbi, mais qui permet une implémentation très simple de la soustraction dès lors qu'on a l'addition. Je commence par le présenter en décimal pour vous en convaincre.

En décimal : complément à 10 sur trois chiffres

On va travailler en base 10 avec des nombres à 3 chiffres. Au lieu de représenter les nombres de 000 à 999, on va dire que

— les codes 000 à 499 représentent les entiers positifs habituels
— 999 représente -1. Remarque : c'est 1000-1.
— 998 représente -2. Remarque : c'est 1000-2.
— ...
— 501 représente -499. Remarque : c'est 1000-499.
— 500 représente -500 (vous avez compris)

Et maintenant, pourquoi c'est un bon codage (on répète en choeur le précepte) :

— Le signe n'est plus explicite, mais il suffit de regarder le chiffre de gauche pour le connaitre : de 0 à 4, on a affaire à un positif, de 5 à 9 c'est un négatif. Quand on transposera tout cela en binaire, le bit de gauche sera directement le bit de signe.
— Notre algo d'addition du CE1 marche pour les négatifs comme pour les positifs, à condition d'ignorer la retenue qui sort. Illustration : calculons 17 + (-2). On envoie un étudiant au tableau pour qu'il pose 017 + 998. Après un laps de temps plus ou moins long, il trouve 1015. On ignore la retenue sortante, c'est-à-dire qu'on reste sur 3 chiffres, et on a bien calculé 15.
— Pour calculer abc-xyz on calcule abc+(1000-xyz). Comme 1000-xyz c'est encore compliqué, on calcule en fait **abc+(999-xyz)+1**. Et là tout devient simple :
  — La soustraction 999-xyz se fait chiffre à chiffre (pas de propagation de retenue). On obtient le code pqr.
  — Le calcul abc+pqr+1, c'est l'algo d'addition du CE1, le +1 consistant juste à ajouter systématiquement une retenue à la colonne de droite, qui n'en avait pas.

  Et donc on a bien ramené la soustraction à l'addition à peine modifiée. Et elle marche pour les positifs comme pour les négatifs (illustration en envoyant un autre étudiant d'excellence calculer 17-42 puis -17 - 42).

En binaire : complément à 2 sur p chiffres

Transposons tout ceci en binaire sur 4 bits (Figure 2.1) :

— Les codes 0000 à 0111 représentent les entiers 0 à 7 comme si de rien n'était
— Le code 1111 représente -1 (remarque : c'est 10000_2 -1)
— ...

Remarque : ignorer la retenue sortante c'est travailler modulo 10000_2, c'est pourquoi la figure est un disque.

Et en binaire tout est plus simple :

— Le signe est directement donné par le bit de gauche.
— pour calculer abcd-xyzt on calcule abcd+(1111-xyzt)+1.
— la soustraction chiffre à chiffre est un *non* logique.
— et l'algo d'addition du CE1 marche très bien.

Remarque Un autre point de vue est de dire que le bit le plus à gauche a le poids qu'il aurait en binaire classique, mais *avec un signe négatif*. Par exemple 1111 représente (-8)+4+2+1=-1. Ceci s'explique, encore une fois, par le modulo.

Généralisation : en complément à 2 sur p bits, le vecteur de bits $(c_{p-1}c_{p-2}\dots c_1c_0)$ représente la valeur $-c_{p-1}2^{p-1} + \sum_{i=0}^{p-2} c_i 2^i$.

Autrement dit, le bit le plus à gauche a un poids négatif (-2^{p-1}). Vérifiez que $n \geq 0$ ssi $c_{p-1} = 0$, et $n < 0$ ssi $c_{p-1} = 1$.

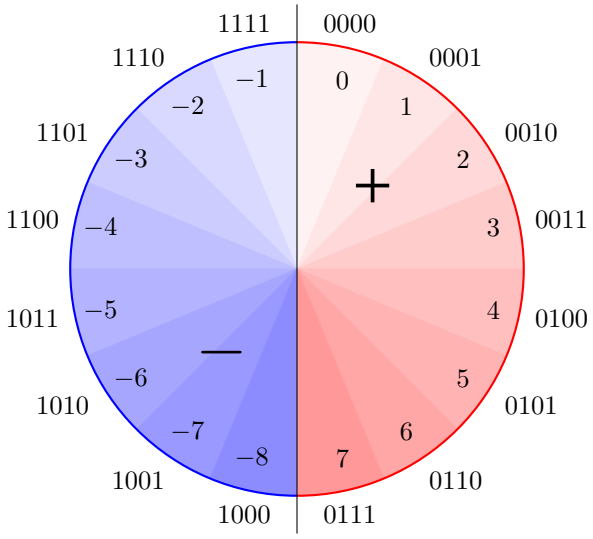


FIGURE 2.1 – Illustration du complément à 2 sur 4 bits (crédit : Benoît Lopez)

Important Comme le bit de gauche est différent des autres, il ne faut jamais parler juste de complément à 2, mais toujours préciser lourdement "sur p bits". C'est différent du binaire non signé, où la taille importe peu.

Conséquences pratiques Avec un type entier signé dans votre langage favori, si ce type est implémenté par du complément à 2 (typiquement sur 16, 32 ou 64 bits) l'addition de deux positifs peut vous donner un résultat négatif... Si vous n'y prenez pas garde, votre boucle `for` risque de devenir une boucle infinie.

Détection des dépassement de capacité dans l'addition L'idée est en gros la suivante :
- Si les deux entrées ont des signes différents, il ne peut y avoir de dépassement de capacité.
- Si elles ont le même signe, il y aura un dépassement si et seulement si le signe du résultat n'est pas le signe des entrées.

Et pourquoi cela s'appelle complément à 2? Eh bien je n'ai jamais su. Je crois que c'est en réaction au complément à 1 (que vous irez wikipedier vous-même).

2.2.4 Et pour aller plus loin

... On peut utiliser une représentation de position avec des *chiffres négatifs*. Par exemple, du décimal avec les chiffres $\{-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5\}$. En notant plutôt le chiffre -5 par $\overline{5}$, on compte comme ceci : $1, 2, 3, 4, 5, 1\overline{4}, 1\overline{3}, 1\overline{2}, 1\overline{1}, 10, 11, 12$...

Vos algorithmes d'addition et de multiplication marchent toujours, ils sont basés sur la formule $\sum_{i=0}^{p} x_i 10^i$.

Gros avantage : fini les tables de 6, 7, 8 et 9 ! C'est une idée de Cauchy, dans une note publiée dans les Comptes-Rendus de l'Académie des Sciences intitulée *Sur les moyens d'éviter les erreurs de calculs numériques*. On remarque que, pour la beauté de la symétrie, on a 11 chifres, plus 10. Du coup, le code est devenu *redondant* : 5 peut aussi s'écrire $1\overline{5}$. Comme chaque fois qu'on a plus de liberté, cela peut être un avantage ou un inconvénient.

Plus près de nous, le diviseur des processeurs intel modernes utilisent en interne une représentation de position en base 16 avec bien plus que 16 chiffres, signés. La motivation est que cela permet de calculer la division plus vite (toujours le précepte) mais c'est bien trop pointu pour être détaillé ici.

2.3 Coder du texte

Second exemple : le texte. On a un alphabet de 26 lettres, plus les chiffres, plus les majuscules et la ponctuation. Tout cela tient en

moins de $128 = 2^7$ caractères, donc on peut coder chaque caractère sur 7 bits. C'est ce que fait le code ASCII.

On l'a déjà dit : un bon code est un code qui permet de faire facilement les traitements utiles. Par exemple, pour le texte,

— il faut que l'ordre alphabétique se retrouve dans les codes, considérés comme des entiers ;
— idem pour les chiffres ;
— il faut qu'on puisse passer de la majuscule à la minuscule en ne changeant qu'un bit ;

ASCII a été inventé par des Américains. Il a été étendu de diverses manières pour d'autres langues, en passant à 8 bits, ce qui permet 128 caractères supplémentaires. Par exemple, il existe deux codes populaires pour le Russe, chacun cohabitant avec les caractères de l'anglais :

— un code qui, lorsqu'on annule le bit 8 [2], projette chaque lettre russe sur la lettre anglolatine qui lui correspond le mieux phonétiquement ;
— un code qui est dans l'ordre alphabétique de l'alphabet cyrillique.

La norme Unicode a tenté d'unifier dans un seul code, sur 16 bits, l'ensemble des écritures de la planète. En pratique, certains caractères nécessitent deux codes de 16 bits, et on peut l'utiliser à travers un encodage sur 8 bits (appelé UTF8) compatible avec le bon vieil ASCII. Bref, c'est une usine à gaz, à l'image de la complexité des écritures des langages de l'humanité. Vous en verrez peut-être des morceaux en TD.

2.4 Coder les images et les sons

On décompose une image en pixels, et on code pour chaque pixel sa couleur. Une couleur est la somme de trois composantes, par exemple vert, bleu, rouge. En codant chaque composante sur 8 bits, on peut déjà coder plus de couleurs que ce que l'œil humain peut distinguer.

Cela donne des quantités d'information astronomiques pour chaque image. On verra comment on peut *compresser* cette information.

Pour le son, on peut le regarder à l'oscilloscope et discrétiser la courbe obtenue. La nouveauté est qu'il faut discrétiser dans le temps et dans l'amplitude.

Avec tout cela, on sait même coder des vidéos.

Remarque : dans un fichier de dessin vectoriel (CorelDraw, XFig, SVG *(scalable vector graphics)*, un standard du web), on ne code pas des images, mais des figures construites à partir de primitives telles que point, droite, cercle... C'est en général plus compact, et on peut zoomer à volonté sans voir apparaître de vilains gros pixels.

Exercice : sur votre linux, ouvrez `/usr/share/icons` et promenez vous dedans (cela dépend des install). Vous trouverez des icones de différentes tailles, et des icones en SVG (dans des répertoires appelés `scalable` ou **symbolic**). Ouvrez les (par exemple dans un navigateur web) et zoomez dessus.

A ma connaissance il n'y a pas de format de fichier vectoriel pour les dessins animés.

2.5 Codes détecteurs d'erreur

On peut ajouter à chaque octet un bit de parité qui vaut 1 s'il y a un nombre impair de 1 dans l'octet, et 0 sinon. A la réception d'un tel paquet de 9 bits, on peut vérifier cette parité. Cela permet de détecter une erreur de transmission d'un bit, mais cela ne permet pas de le corriger.

2. Cela arrivait aux temps héroiques de l'internet : votre message pouvait passer par un ordinateur anglo-saxon ou configuré comme tel, qui ne ressentait pas le besoin de transmettre plus de 7 bits par octet. On en trouve par exemple des traces dans le protocole `ftp` *file transfer protocol* qui offre toujours deux modes, "ASCII" et "binary".

2.6 Codes correcteurs d'erreur

Le *code auto-correcteur de Hamming* permet de corriger un bit erroné dans une information de 4 bits (i_3, i_2, i_1, i_0). On code cette information au moyen de 3 bits supplémentaires p_2, p_1 et p_0 : le codage est $(i_3, i_2, i_1, p_2, i_0, p_1, p_0)$, où :

— p_0 est le bit de parité paire du sous-mot (i_3, i_1, i_0, p_0),
— p_1 est celui du sous-mot (i_3, i_2, i_0, p_1), et
— p_2 est celui de (i_3, i_2, i_1, p_2).

À la réception de $(i_3, i_2, i_1, p_2, i_0, p_1, p_0)$, on vérifie ces trois parités, et on définit t_k par : $t_k = 0$ si p_k est correct et $t_k = 1$ sinon.

Par magie, (t_2, t_1, t_0) est alors le rang dans $(i_3, i_2, i_1, p_2, i_0, p_1, p_0)$ du bit erroné, écrit en binaire : 000 si l'information est correcte, 001 si p_0 est erroné, 010 si p_1 est erroné, 011 si i_0 est erroné, 100 si p_2 est erroné, 101 si i_1 est erroné, 110 si i_2 est erroné ou 111 si i_3 est erroné.

On parle ici du code $(7, 4)$: 7 bits transmis pour 4 bits d'information effective. On peut généraliser ces idées.

Il y a bien sûr des limitations : le code de Hamming ne permet de corriger qu'un bit d'erreur, pas deux bits simultanés. Et cela coûte presque autant que ce qu'on veut corriger.

Les codes correcteurs d'erreur sont utilisés en particulier dans

— Les barrettes de mémoire avec ECC *(error correcting code)* dans le nom, ce qui explique qu'elles coûtent plus cher.
— La transmission de données, autrefois sur fil, actuellement par radio.

2.7 Compression d'information

Notions : compression avec perte ou sans perte.

Algos : on espère que vous en verrez quelques uns en cours d'algo.

Chapitre 3

Transformer l'information : circuits combinatoires

On a déjà mentionné la préhistoire analogique. Remarquez que le calcul analogique à Supelec utilisait déjà l'approche hiérarchique : il fallait se ramener à des amplis ops.

Mais désormais on manipule de l'information sous sa plus simple expression : codée en binaire.

Et pour transformer de l'information binaire, on va utiliser les outils offerts par l'algèbre booléenne.

Vérifiez que vous comprenez la différence entre *algèbre*, *calcul*, *expression* et *circuit* booléens.

— L'*algèbre* booléenne va définir un certain nombre d'*opérations* et de *propriétés*.
— On utilisera les *opérations* de l'algèbre pour construire des *expressions* booléennes, et on utilisera les propriétés de l'algèbre pour manipuler ces expressions.
— La définition la plus simple d'une *fonction* booléenne sa table de vérité. Elle donne, pour chaque valeur possible des entrées, les valeurs de sortie correspondantes.
— Mais on peut également définir une *fonction* booléenne par une *expression* booléenne. Pour une fonction donnée, il existe une infinité d'expressions booléennes : on passe de l'une à l'autre par des identités de l'*algèbre*, comme $\overline{a \vee b} = \overline{a} \wedge \overline{b}$.
— Dans une technologie donnée, on sait implémenter certaines opérations de l'algèbre par des *opérateurs*, qu'on appelle alors volontiers des *portes logiques*.
— En assemblant ces portes, on peut construire des *circuits logiques* qui implémentent une expression, donc une fonction booléenne donnée. C'est ce qu'on fera en 3.2, en se posant notamment des questions de coût et de performance.

3.1 Algèbre booléenne

3.1.1 Définitions et notations

L'algèbre booléenne définit

— un ensemble $\mathbb{B}$ à deux éléments, muni de
— une opération unaire involutive (la négation)
— et deux opérations binaires de base (ET et OU) commutatives, associatives, ayant chacun un élément neutre, et distributives l'une par rapport à l'autre [1].

On utilise des mélanges variables des notations suivantes :

— Les valeurs sont notées (vrai, faux) – ou une traduction dans la langue de votre choix –, ou $(0, 1)$, ou $(\emptyset, \{\emptyset\})$, ou $(\top, \bot)$.

1. Un peu plus loin, on voit qu'une seule opération suffirait, mais c'est tout de même plus confortable avec ces trois-là.

— L'opération unaire est notée NON – ou une traduction dans la langue de votre choix –, ou $\neg$, ou C (complément ensembliste), ou $\overline{\text{toto}}$.
— Les opérations binaires sont notées (OU, ET) – ou une traduction dans la langue de votre choix –, ou $(+, .)$, ou $(\vee, \wedge)$, ou $(\cup, \cap)$.

Je m'économise d'écrire les tables de vérité.

3.1.2 Expression booléenne

Lorsqu'on écrit des expressions booléennes, on peut utiliser des variables booléennes, les deux constantes, et des parenthèses. L'opérateur unaire est prioritaire sur les deux opérateurs binaires, qui ont une priorité équivalente : $\neg a \vee b == (\neg a) \vee b$.

Je déconseille la notation utilisant (+,.). Elle présente l'intérêt d'économiser des parenthèses, puisqu'on adopte alors la priorité usuelle de . sur $+$. En revanche, l'arithmétique entière ou réelle a câblé dans votre cerveau des intuitions avec ces opérateurs qui seront fausses en booléen. Par exemple, les deux opérateurs booléens distribuent l'un sur l'autre. Vous serez familier de $(a + b).c = ac+bc$, mais vous oublierez que $ab+c = (a+c).(b+c)$ (démonstration : comme toujours pour prouver des identités booléennes, considérer $c = 0$ puis $c = 1$). D'autres pièges vous attendent si vous utilisez cette notation. Vous voila prévenus.

La notation à base de surlignage pour la négation est pratique, car elle permet d'économiser la plupart des parenthèses sans ambiguité. Exemple :

$$x \wedge y = \overline{\overline{x} \vee \overline{y}}$$

3.1.3 Dualité

L'algèbre booléenne est parfaitement symétrique : pour toute propriété, il existe une autre propriété déduite en échangeant 1 avec 0 et $\vee$ avec $\wedge$. Cette idée pourra souvent économiser du calcul. Encore une fois, elle est particulièrement contre-intuitive avec la notation $(+, .)$ à cause des priorités héritées de l'algèbre sur les réels.

3.1.4 Quelques propriétés

$0 \wedge x = 0$	$1 \vee x = 1$	(élément absorbant)
$1 \wedge x = x$	$0 \vee x = x$	(élément neutre)
$\overline{x \wedge y} = \overline{x} \vee \overline{y}$	$\overline{x \vee y} = \overline{x} \wedge \overline{y}$	(lois de Morgan)

3.1.5 Universalité

On peut ramener nos trois opérations booléennes à une seule, le non-et, par application des règles suivantes :

— Négation : $\overline{x} = \overline{x \wedge x}$
— Ou : $x \vee y = \overline{\overline{x} \wedge \overline{y}} = \overline{\overline{x \wedge x} \wedge \overline{y \wedge y}}$
— Et : $x \wedge y = \overline{\overline{x \wedge y}} = \overline{\overline{x \wedge y} \wedge \overline{x \wedge y}}$

On dira que l'opération non-et est universelle. Par dualité, non-ou aussi. Cela n'est pas sans nous interpeler profondément dans notre relation à la transcendance de l'Univers, croyez-vous ? Eh bien figurez vous que les portes de base de notre technologie CMOS seront justement NON-ET et NON-OU. Nous voila convaincus qu'elles ne sont pas plus mauvaises que ET et OU. Plus précisément, non seulement elles forment un ensemble universel, mais en plus, pour implémenter une fonction donnée, il faudra un nombre équivalent de portes, qu'on se ramène à des NON-ET/NON-OU ou bien à des ET/OU.

3.1.6 Fonctions booléennes

Une fonction booléenne est une fonction d'un vecteur de bits vers un vecteur de bits :

$$f : \mathbb{B}^n \to \mathbb{B}^m$$

En général (pas toujours) on considérera une telle fonction comme un vecteur de fonctions simples $f_i : \mathbb{B}^n \to \mathbb{B}$, et on étudiera séparément chaque f_i.

On peut définir une fonction booléenne de diverses manières :

— par extension (en donnant sa table de vérité) – exemple une table de sinus,
— par intention (en donnant des propriétés qu'elle doit vérifier, l'unicité n'est alors pas garantie) – exemple la somme de deux chiffres BCD, voir ci-dessous,
— par une expression algébrique – exemple, la fonction NON-ET.

On peut donner une définition d'une fonction par sa table, mais avec des "don't care" pour les sorties. Exemple : la fonction d'addition de deux chiffres BCD (pour *binary-coded decimal*). Il s'agit des chiffres décimaux de 0 à 9 codés en binaire sur 4 bits. Les codes entre 10 et 15 n'étant jamais utilisés, on se fiche (don't care) de ce que fait la fonction dans ce cas. Formellement, cela revient à donner une définition intentionnelle de la fonction en question, et plusieurs fonctions booléennes différentes feront l'affaire.

3.2 Circuits logiques et circuits combinatoires

3.2.1 Signaux logique

Définition : un signal logique, c'est un dispositif physique pouvant transmettre une information binaire d'un endroit à un autre.

C'est un cas particulier, on peut considérer des signaux qui transmettent des informations non binaires. On définira aussi plus tard des signaux "trois états".

On dessinera un signal par un trait, ou éventuellement plusieurs. Cela colle avec la technologie la plus courante, dans laquelle un signal sera implémenté par un fil métallique. La valeur du signal sera codée par le potentiel électrique de ce fil. La transmission du signal se fera par un courant électrique.

La notion de signal logique est plus générale que cela : vous pourrez à la fin de ce cours réaliser un ordinateur complet en lego [2], dans lequel un signal binaire sera implémenté par une tige pouvant prendre deux positions.

3.2.2 Circuits logiques

Définition : un *circuit logique*, c'est un dispositif physique destiné à manipuler des informations binaires.

Un circuit logique se présente, vu de l'extérieur, comme une boîte noire avec des signaux d'entrée et des signaux de sortie.

3.2.3 Portes de base

Définition : une *porte logique* est un circuit logique implémentant une opération booléenne élémentaire. Les figures 3.1 et 3.2 montrent les dessins standardisés pour les principales portes de base.

Exercice : construisez-les en Lego. Tout ensemble *universel* (au sens du 3.1.5) de portes de base fera en pratique l'affaire. Si on en a plus on aura plus de liberté pour optimiser, voir plus bas.

2. En googlant un peu vous trouverez des tentatives dans ce sens

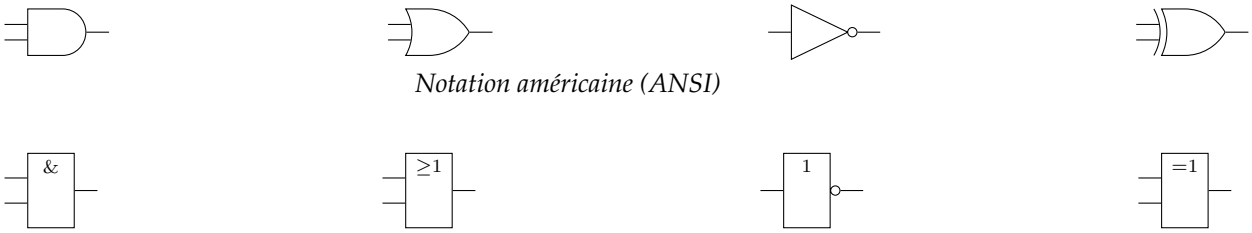


FIGURE 3.1 – Les dessins des portes ET, OU, NON et XOR (ou exclusif).

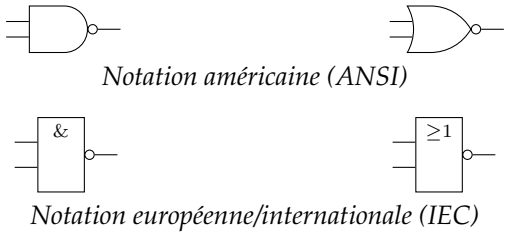


FIGURE 3.2 – Le non-et (NAND) et non-ou (NOR). En général, on peut mettre des petits ronds sur les entrées et les sorties pour dire qu'on les inverse.

3.2.4 Circuits combinatoires

Définition : un circuit combinatoire, c'est un dispositif physique implémentant une expression booléenne dans une certaine technologie.

Le circuit combinatoire est un cas particulier de circuit logique avec une grosse restriction : *il ne possède pas de mémoire du passé*. La sortie d'un circuit combinatoire est fonction uniquement de l'entrée, pas des entrées qu'il a pu avoir dans le passé. Par exemple, une GameBoy n'est pas un circuit combinatoire.

Exemples simples de circuit pas combinatoire : le bistable, figure 3.3.

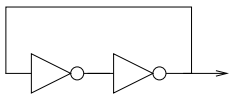


FIGURE 3.3 – Le bistable

Pour trouver un circuit combinatoire correspondant à une fonction booléenne donnée, on passe par un stade intermédiaire : on détermine une expression algébrique de la fonction qui aura une traduction directe en matériel. On verra cela en 3.4.

3.2.5 Circuits combinatoires bien formés

Étant donné un ensemble de portes de bases qui sont toutes des circuits combinatoires, un CCBF est formé par

— une porte de base
— un fil
— la juxtaposition de deux CCBFs posés l'un à côté de l'autre
— un circuit obtenu en connectant les sorties d'un CCBF aux entrées d'un CCBF
— un circuit obtenu en connectant entre elles deux entrées d'un CCBF.

On peut montrer qu'on obtient un graphe sans cycle de portes de bases.

Ce qu'on s'interdit :
— faire des cycles, car cela permettrait des situations mal définies comme le circuit instable de la figure 3.4,
— connecter des sorties entre elles (que se passerait si une sortie est 1 et l'autre 0 ?)

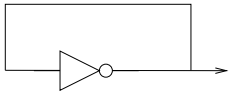


FIGURE 3.4 – Le pas-stable

Plus tard on lèvera ces deux interdictions dans des circonstances bien maîtrisées. Avec des circuits à cycles, on pourra construire des oscillateurs (en mettant des retards dans notre pas-stable) et des mémoires (le bistable en est une, mais dans laquelle on ne peut pas entrer d'information à cause de la seconde interdiction, donc il faudra le bricoler un peu).

3.2.6 Surface et délai d'un circuit combinatoire

Pour une fonction booléenne donnée, il y a une infinité dénombrable de manières de l'implémenter. Ce qui est intéressant, en architecture des ordinateurs, c'est de trouver la meilleure.

Exemple : construire un ET à quatre entrées à partir de NON-OU.

Meilleure selon quel critère ? La technologie amène avec elle ses métriques de qualité : taille, vitesse, consommation d'énergie, niveau d'émission électromagnétique...

Dans la notion la plus abstraite de circuit combinatoire, il n'y a pas plus de notion de temps que dans la notion d'expression booléenne. Par contre, toute réalisation physique d'un circuit combinatoire aura un certain *délai* de fonctionnement, noté τ, et défini comme le temps maximum entre un changement des entrées (à l'instant t) et la stabilisation des sorties dans l'état correspondant (à l'instant $t + \tau$). Les sorties du CC physique peuvent passer par des états dits transitoires pendant l'intervalle de temps $[t, t + \tau]$. C'est le cas du circuit de la figure 3.5. On apprendra à construire des circuits dans lesquels on garantit que ces états transitoires sont sans conséquence.

FIGURE 3.5 – Un circuit pas compliqué qui nous fait des misères transitoires

On peut définir (ou mesurer) le délai de chaque porte de base. Une fois ceci fait, on peut ramener le calcul du délai dans un CCBF à un problème de plus long chemin dans le graphe du CCBF. Ce plus long chemin est appelé *chemin critique* (en Australien : *critical path*). On aura une tendance malheureuse à identifier le chemin critique et le délai du chemin critique.

Dans la vraie vie, le délai d'une porte de base peut être différent suivant l'entrée et la sortie considérée. Dans le cas général, pour une porte de base à n entrées et m sorties, on aura une matrice $n \times m$ de délais. Chaque entrée de la matrice est elle-même le max des délais pour toutes les configurations des autres entrées. De plus, le délai pourra être différent sur front montant et sur front descendant, on aura alors une matrice $2n \times m$. Naturellement le calcul du délai d'un

CCBF devient un peu plus compliqué, mais cela reste parfaitement automatisable, et même vous sauriez faire.

Enfin, les authentiques gourous du circuit sauront retailler les transistors pour obtenir les délais qu'ils veulent. Le compromis sera : plus c'est rapide, plus c'est gros et plus cela consomme.

3.3 Au delà des portes de base

3.3.1 À deux entrées

Exercice : énumérons toutes les fonctions logiques à 2 entrées et une sortie, et donnons leur à chacune un petit nom.

3.3.2 À trois entrées

Dans un train électrique, on a une brique de base appelée *aiguillage*, qui en fonction d'une information binaire de *direction* (entrée du bas sur la figure), transmet l'information (le train) soit selon un chemin, soit selon l'autre.

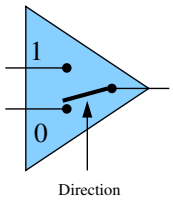


FIGURE 3.6 – L'aiguillage

En assemblant $1 + 2 + 4 + ... + 2^{k-1} = 2^k - 1$ aiguillage, on sait construire une gare de triage complète. Observez que l'adresse de la case d'arrivée est donnée en binaire sur les fils de direction.

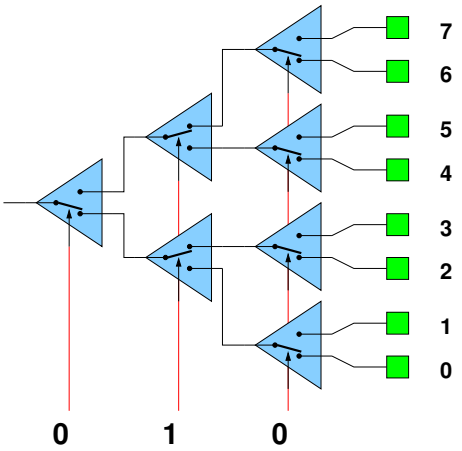


FIGURE 3.7 – Une gare de triage permet de construire une mémoire adressable

L'aiguillage, en technologie train électrique, marche dans les deux sens.

Si on parle de circuits logiques, on distinguera deux familles de circuit qui ont *la même construction sous forme d'abre binaire* : le **multiplexeur** (2^n en 1) et le **démultiplexeur** (1 vers 2^n).

Chacun a une entrée de sélection d'adresse (en général appelée *select* ou *sel*) de n bits. De plus,

— le multiplexeur (ou MUX) a aussi 2^n entrées, et une sortie. Il transmet vers sa sortie celle des 2^n entrées dont le numéro est donné en binaire sur l'entrée *sel*.
— le démultiplexeur (ou DEMUX) a aussi une entrée et 2^n sorties. Il transmet son entrée vers celle des 2^n sortues dont le numéro est donné en binaire sur l'entrée *sel*.

La figure 3.8 vous donne le dessin standard d'un multiplexeur 2 vers 1, à partir duquel vous saurez construire tous les autres (en suivant la construction de la figure 3.7. Le démultiplexeur, c'est le même dessin en retournant certaines flèches.

Exercices :

— Dessinez le DEMUX 1 vers 2 à coté de la figure 3.8.
— Construisez le MUX avec des portes ET, OU et NON.
— Construisez le MUX 8 vers 1 à partir du MUX 2 vers 1.

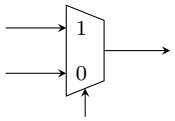


FIGURE 3.8 – Le multiplexeur 2 vers 1 est une porte logique à 3 entrées et une sortie. Dessinez à côté le démultiplexeur 1 vers 2.

Avec le multiplexeur on fabrique aussi le décaleur *(barrel shifter)*.

3.3.3 Pour le calcul sur les entiers

La base du calcul sur les entiers c'est l'addition : construisons un additionneur.

On part de l'algo d'addition en décimal, qu'on généralise un peu, et on le formalise mathématiquement :

Soit β un entier (la base, qui vaut 10 pour votre petite sœur mais vaudra 2 en binaire). Soit la fonction *table d'addition* TA :

$$\begin{array}{rcl} \{0,1\} \times \{0..\beta-1\} \times \{0..\beta-1\} & \to & \{0..1\} \times \{0..\beta-1\} \\ (r,x,y) & \mapsto & (r',s) \\ & & \text{tq} \quad \beta r' + s \;=\; r + x + y \end{array}$$

— cette fonction prend une "retenue entrante" (0 ou 1) et deux chiffres, et retourne leur somme ;
— Cette somme est comprise entre 0 et $2\beta-1$ (par exemple en décimal entre 0 et $9+9+1 = 19$). Elle s'écrit donc en base β sur deux chiffres :
  — Le chiffre de poids faible de la somme est appelé *somme modulo la base* ou juste *somme*
  — Le chiffre de poids fort de la somme est appelé *retenue* et vaut 0 ou 1 quelle que soit la base.

L'algorithme d'addition de deux nombres de n chiffres est alors

1: $c_{-1} = 0$
2: **for** $i = 0$ to n **do**
3: $\quad (c_i, s_i) = \text{TA}(c_{i-1}, x_i, y_i)$
4: **end for**

On peut le dérouler de plusieurs manière en matériel : algo séquentiel (utilisation du temps) ou parallèle (utilisation de l'espace). Cela aussi est une idée générale.

Pour construire l'additionneur binaire il suffit de considérer $\beta = 2$. Mais alors les trois entrées deviennent symmétriques (toutes entre 0 et 1). La porte obtenue s'appelle le *full adder*. La fonction booléenne correspondante a 3 entrées et deux sorties. En TP nous la construirons à partir de portes logiques.

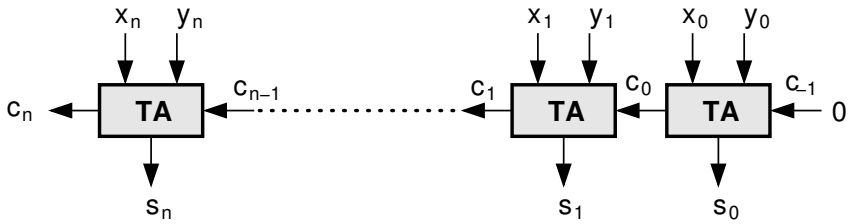


FIGURE 3.9 – L'algo d'addition déroulé dans l'espace. TA c'est "table d'addition".

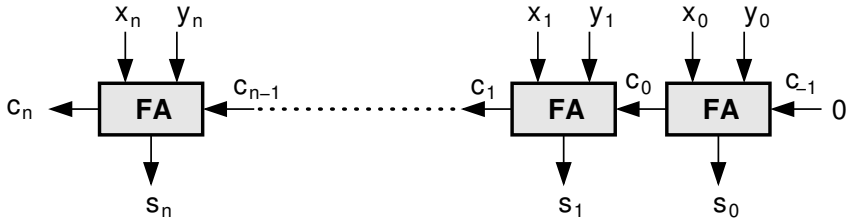


FIGURE 3.10 – Additionneur *binaire* à propagation de retenue.

Complément à 2

Pour calculer une soustraction, on peut contruire un soustracteur. Mais il ressemble à un additionneur : il propage des retenues, etc. On aimerait bien partager le même matériel entre addition et soustraction.

Pour cela, on utilise la notation en complément à 2, introduite à la section 2.2.3

Pour calculer 745-169, on peut calculer 745+1000-169 -1000. Le -1000 final, c'est juste d'ignorer la retenue sortante. Quant au +1000, on l'écrit +999+1, et notre soustraction devient :
745 + (999-169) +1.

Pourquoi c'est mieux ? parce que la soustraction 999-169 se fait chiffre à chiffre, sans propagation de retenue. Quant au 1 qu'on ajoute en plus de l'addition normale, c'est juste une retenue sur la première colonne.

Tout cela se transpose en binaire (en remplaçant 999 par 111, bien sûr), et l'on obtient l'architecture d'additionneur/soustracteur de la figure 3.11. Le complément, en binaire, est juste la négation (pour un booléen x, $\overline{x} = 1 - x$). La boîte $\boxed{\text{C}}$ est un XOR.

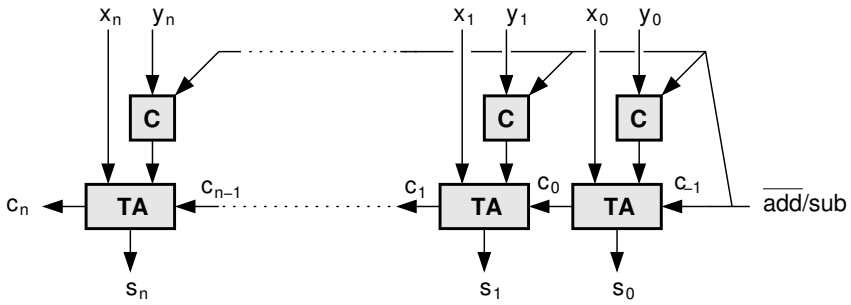


FIGURE 3.11 – Additionneur en complément à 2.

Maintenant, que se passe-t-il pour une soustraction qui donne un résultat négatif, par exemple 000 -001 ? En décimal on obtient (toujours en ignorant le 1 à gauche) 000 + 999-001 +1 = 999. Si donc on choisit de représenter -1 par 999 (et -2 par 998, etc), alors toute la mécanique précédente fonctionne encore. En fait, elle fonctionne modulo 1000.

Addition et calcul de l'opposé : définitions

à savoir! Soient $m = (c_m)_{\overline{2}}$ et $n = (c_n)_{\overline{2}}$ en complément à 2 sur p bits.

On dit qu'il y a dépassement de capacité dans une opération en complément à 2 sur p bits lorsque le résultat de l'opération ne peut pas être représenté sous cette même forme (il est soit trop petit, soit trop grand).

— En l'absence de dépassement de capacité, le codage de $m + n$ en complément à 2 sur p bits est le codage de l'entier naturel $(c_m + c_n) \mod 2^p$. Si m et n sont de même signe, il y dépassement ssi le signe du résultat calculé diffère du signe des opérandes ; s'ils sont de signes opposés, aucun dépassement n'est possible.

— En l'absence de dépassement de capacité, le codage de $-n$ en complément à 2 sur p bits est le même que celui de l'entier naturel $(\overline{c}_{p-1}\overline{c}_{p-2}\dots\overline{c}_1\overline{c}_0)_2 + 1 \mod 2^p$. Il y dépassement ssi le signe du résultat calculé est le même que celui de l'opérande.

3.3.4 Conclusion

Les gares de triage et les additionneurs sont des circuits construits par une approche algorithmique. Pour les gros circuits, c'est l'approche à privilégier : il est toujours mieux de faire marcher son intelligence, de diviser pour régner, etc.

Mais il existe aussi une approche automatique que nous allons voir maintenant. Vous pouvez l'appliquer à la main, ou la programmer. On verra qu'elle ne s'applique en pratique que pour de "petites" fonctions booléennes (par exemple pour construire un *full adder* ou un multiplexeur) à partir de portes logiques.

3.4 D'une fonction booléenne à un circuit combinatoire

3.4.1 Par les formes canoniques

Expansion de Boole : soit f une fonction booléenne à n entrées, alors

$$f(x_1, x2, ...x_n) = x_1.f(1, x_2, ..., x_n) + \overline{x_1}.f(0, x_2, ...x_n)$$

Démonstration : considérer $x_1 = 0$ puis $x_1 = 1$.

Exercice : donner l'expansion de Boole duale de celle-ci. J'ai perfidement utilisé la notation que j'ai dit que je n'aimais pas.

On appelle parfois un *monôme canonique* un ET de toutes les variables apparaissant chacune soit sous forme directe soit sous forme complémentée. On peut appeler *monal canonique* le dual du précédent.

Les deux variantes de l'expansion de Boole, appliquées récursivement, permettent d'obtenir automatiquement, à partir d'une fonction booléenne donnée par sa table de vérité, une expression booléenne de cette fonction sous forme dite canonique.

— Forme canonique disjonctive : OU (ou disjonction, ou somme mais je vous ai dit que je n'aimais pas) de monomes

— Forme canonique conjonctive : ET (ou conjonction) de monals (monaux ?) [3]

Exemple : fonction majorité à trois entrées.

Quelle forme préférer ? Considérons la forme conjonctive. Naturellement, on simplifie tous les monômes correspondant à une valeur 0 de la fonction. Donc cette forme sera préférée pour une fonction ayant une majorité de 0 dans sa table de vérité. Inversement, s'il y a une majorité de 1, la forme disjonctive aura moins de monaux rescapés que la forme conjonctive de monômes.

Exemple : la fonction majorité à 3 entrées, qui renvoie 1 si au moins deux des trois entrées sont à 1 et 0 sinon.

3. C'est sans doute pour cela qu'on préfère habituellement la disjonctive...

Simplification des formes canoniques Supposons pour anticiper qu'on veut se ramener à un nombre le plus petit possible de NON-ET et de NON-OU à moins de 4 entrées.

Il faut parenthéser notre forme canonique. Il y a un très grand nombre de parenthésages possibles (un amateur de combinatoire me dira combien ?).

Ensuite on peut appliquer des règles de réécriture. Lesquelles, et dans quel ordre ?

On peut privilégier la minimisation du nombre de portes (surface) ou la minimisation de la profondeur de l'arbre (délai). Dans les deux cas on sait comparer deux expressions pour choisir la meilleure. Par contre, pour obtenir la meilleure expression pour une fonction booléenne donnée, on ne sait pas faire tellement mieux que toutes les essayer (en commençant par essayer tous les parenthésages) : le problème est NP-fâcheux.

Heureusement il y a de bonnes heuristiques. Voyons maintenant l'une d'elles.

3.4.2 Arbres de décision binaire

Un arbre de décision (en néozélandais BDD pour *binary decision diagram*) est simplement un arbre d'évaluation de la fonction en considérant la valeur de chacune de ses variables l'une après l'autre. Un exemple est donné par la figure 3.12. Pour connaître la valeur de $f(x_1x_2x_3)$, on part de la racine, et à chaque nœud de profondeur i on part dans le sous-arbre gauche si $x_i = 0$, et dans le sous-arbre droite si $x_i = 1$. Arrivé à une feuille, l'étiquette de la feuille donne la valeur de la fonction.

Si vous voulez, c'est comme un arbre de détermination pour reconnaître les papillons ou les fleurs des champs, si vous avez pratiqué ces choses-là.

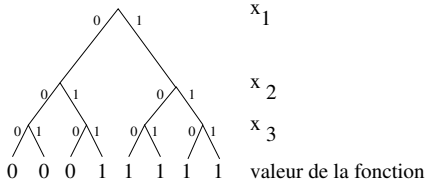


FIGURE 3.12 – Un arbre de décision binaire

La construction du BDD est triviale en partant de la table de vérité de la fonction. Faisons le au tableau.

Attention, l'arbre obtenu dépend de l'ordre des variables, c'est pourquoi on utilise souvent le terme OBDD (pour *ordered*).

L'OBDD a autant de feuilles que la table avait d'entrée, donc c'est une structure plutôt plus lourde que la table de départ. Son intérêt est de pouvoir ensuite être réduite pour obtenir une représentation plus compacte de la fonction. On applique pour cela les transformations suivantes :

— Si deux sous-arbres sont identiques, on les fusionne

— Si un nœud a deux fils identiques, on le court-circuite et on le supprime.

On obtient un arbre réduit (ROBDD), comme sur la figure 3.12, qu'il est ensuite facile de traduire en un circuit (exercice).

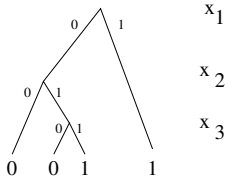


FIGURE 3.13 – Un arbre de décision binaire réduit

Tout ce processus est déterministe, et requiert au plus $O(2^n)$ en temps et en espace, donc pas plus que ce que demande la construction de la table de vérité. Donc il est tractable pour des n

assez grand (pour des fonctions jusqu'à 20 bits sans problème). C'est trop beau. Cela ne va pas durer.

En effet, il reste un problème : le facteur de réduction final dépend de l'ordre des variables choisi pour la construction de l'arbre. Or il y a pour n variables $n!$ permutations de ces variables. Laquelle d'entre elle va donner les simplifications les plus avantageuses ? Eh bien avant d'essayer, on ne sait pas. Nous voilà à nouveau contraints de bricoler des heuristiques, et tous les ans il y a des articles sur le sujet.

Remarque : l'OBDD, et même le ROBDD sont, pour un ordre des variables donné, des représentations canoniques. Elle sont aussi utilisées pour prouver l'équivalence de circuits (par exemple développé sur FPGA puis "recompilé" vers un ASIC).

3.5 Application : construction des circuits arithmétiques de base

3.5.1 Addition/soustraction binaire

Ah ben on a déjà tout vu en 3.3.3.

3.5.2 Multiplication binaire

En TD peut-être. Montrons juste qu'on sait poser la multiplication en binaire : cela suffit à définir les différentes fonctions booléennes nécessaires.

Evidemment c'est encore une construction algorithmique de circuit.

3.5.3 Division binaire

Même texte que pour la multiplication

3.6 Conclusion

On a une technique universelle pour transformer n'importe quelle fonction booléenne, donnée par sa table de vérité, en un circuit qui l'implémente. Cette technique s'adaptera facilement à une *bibliothèque de portes de base* pour une technologie donnée.

Toutefois, on se trouve confronté à des questions d'optimisation dont la complexité, dans le cas général, est exponentielle en le nombre d'entrées de la fonction.

Pour cette raison, faire de l'algorithmique intelligemment donne en général de meilleurs résultats que la technique universelle. C'est ce que l'on a vu pour construire les multiplexeurs et les opérateurs de calcul.

3.7 Annexe technologique contingente : les circuits CMOS

3.7.1 Transistors et processus de fabrication

Règles du jeu du CMOS :

— Un transistor est, en gros, un interrupteur, commandé par un fil de commande qui arrive sur sa "grille" (gate). Voir les figures 3.14 et 3.15 ci-dessous.

— On a deux types de transistor, le transistor normal (qui s'appelle NMOS pour N-type Metal Oxide Silicium), qui est passant quand on met la grille à 1 et ouvert sinon, et le transistor-dessiné-avec-un-rond-sur-la-grille (qu'on appelle PMOS mais je refuse de m'en souvenir), que c'est le contraire.

— Pour de sordides raisons électroniques, le NMOS marche bien (i.e. conduit bien le courant) quand il est relié à la masse (ou à un autre NMOS relié à la masse), alors que le PMOS c'est le contraire : il aime bien être relié à l'alimentation positive.

— du coup il faut les assembler de telle manière que les 1 logique produits par un circuit viennent à travers des PMOS, alors que les 0 logiques doivent être produits par des NMOS.

3.7.2 Portes de base

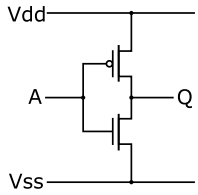


FIGURE 3.14 – Un dessin d'inverseur pompé sur Wikipedia. Vdd cela veut dire +5V (pour une certaine valeur de 5) et Vss cela veut dire la masse.

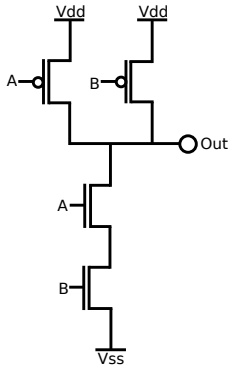


FIGURE 3.15 – A votre avis, c'est quoi comme porte ?

Illustration du fait que brancher deux sorties ensembles, c'est mal : ben, on voit bien que cela fait un joli court-circuit de l'alimentation à la masse, qui fait assez vite fondre les transistors sur le chemin...

3.7.3 Vitesse, surface et consommation

Sans entrer dans les détails, voici ce qui se passe dans un circuit composé de transistors branchés entre eux :

— La grille d'un transistor (gate) se comporte comme un condensateur : il faut charger et décharger ce condensateur pour le faire commuter. Les entrées d'une porte CMOS de base (inverseur, nand ou nor) sont toutes connectées à des grilles : pour faire basculer ces portes, il faut charger ces condensateurs.

— Par conséquent, lorsqu'on connecte une sortie à n entrées (fan-out de n), on divise le courant qu'est capable de produire la sortie entre les n entrées, et on multiplie donc par n le temps qu'il faut pour charger ces n entrées (donc faire basculer les transistors correspondants, donc faire passer l'information). Typiquement, dans un circuit, on accepte un fanout de 4 maximum, au-delà c'est mal.

— La connexion entre la source et le drain du transitor se comporte comme une résistance, assez faible quand le transistor est passant, et élevée quand il est bloqué.

— Plus la grille est large, plus elle présente une capacité en entrée élevée, c'est-à-dire plus il faut du courant pour la faire basculer, mais plus elle peut laisser passer de courant entre la source et le drain.
— Autrement dit, un transistor plus large est plus rapide du point de vue de sa sortie (il peut passer plus de courant pour charger plus vite les transistors suivants) mais plus lent du point de vue de son entrée (il faut plus de temps pour le charger pour le faire basculer).

Tout cela se mord la queue. Heureusement, une chouette technique (Logical Effort) permet de modéliser la vitesse d'un circuit CMOS en fonction de la taille des transistors, et ensuite d'optimiser cette vitesse en dérivant ce qu'il faut pour chercher quand la dérivée s'annule.

Chapitre 4

Memoriser l'information

Enchainement des calculs, accumulateur. Même à l'intérieur d'un calcul, on a des "retenues" : ce qu'on retient.

4.1 Vue abstraite des organes de mémorisation

4.1.1 Le registre ou mémoire ponctuelle

Voici le registre sur front, ou Flip-Flop [1] (on verra pourquoi ce nom rigolo un peu plus bas) :

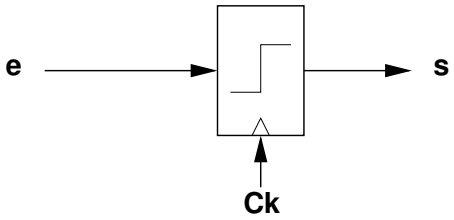


FIGURE 4.1 – Symbole pour un registre sur front montant

Il y a mémorisation de l'entrée e sur front montant (changement de valeur de 0 à 1) de l'entrée Ck.

Notez le triangle sur l'entrée d'horloge indique que ce signal porte une information temporelle, pas une donnée.

Une information d'instant sera toujours donnée par un changement de valeur, qu'on appelle un front (soit montant, soit descendant).

C'est pour cela que j'ai dessiné le front sur le registre. On peut évidemment aussi construire des registres sur front descendant.

Mettons 8 registres en parallèle, avec la même entrée d'horloge : on obtient un registre 8 bits. Par extension, les mémoires de travail dans le processeur sont appelées des registres. Exemples : le PC est un registre à l'intérieur du processeur ; le Pentium a 4 registres entiers et 8 registres flottants alors que l'Itanium a 128 de chaque.

4.1.2 Mémoires adressables

Une mémoire addressable c'est plein de registres juxtaposés. Chaque registre a une adresse, et on peut sélectionner un registre particulier par son adresse.

1. Dans la vraie vie (c'est-à-dire sur Wikipedia) vous trouverez souvent le terme "bascule D" pour ce registre. Je n'aime pas pour la raison détaillée dans la prochaine note de bas de page. Toutefois, vous voila prévenus.

Les opérations de base qu'on peut faire avec une mémoire adressable sont donc

— lire la donnée à l'adresse a. Dans ce cas il faut donner l'information d'adresse à la mémoire, et elle répond par l'information contenue dans la cellule visée.
— écrire une donnée d à l'adresse a. Dans ce cas on doit donner à la mémoire les informations d'adresse et de donnée, et un instant à partir duquel le contenu de la mémoire est changé.

On appelle aussi la mémoire adressable RAM *(random access memory)*, parce que c'est plus court. Parenthèse culturelle : *random* veut dire aléatoire, ce qui parfois veut dire "au hasard", mais pas ici. Ici *random* veut dire "au choix".

Moralement, la lecture d'une mémoire est une fonction combinatoire des entrées, alors que l'écriture demande un fil d'horloge (qui porte une information de temps : à quel moment précis le contenu de la mémoire va changer). C'est donc très différent.

En pratique toutefois, quand on construit une mémoire, on s'arrange pour offrir la même interface en lecture et en écriture. Typiquement on a un bus d'adresses, un bus de données, un fil d'horloge, et un signal $R/\overline{W}$. Au front montant de l'horloge,

— en cas de lecture (si $R/\overline{W} = 1$) le contenu de la case adressée est transféré sur le bus de données,
— en cas d'écriture (si $R/\overline{W} = 0$) le contenu du bus de données est transféré dans la case adressée.

4.1.3 Mémoires à accès séquentiel : piles et files

Pile ou LIFO *(last in, first out)* Les primitives sont :

— empiler(info)
ajoute un étage
— dépiler()
enlève un étage : destruction de la donnée dans la pile
— test pile vide

File ou FIFO *(first in, first out)* Les primitives sont :

— insérer(info)
— extraire() – avec destruction
— test file vide

On appele aussi la file : tampon ou *buffer*.

Remarque : les pile et files, conceptuellement, sont de capacité infinie... Si on les réalise, leur capacité sera sans doute finie, et il faudra ajouter le test "pile/file pleine".

4.1.4 Mémoires adressables par le contenu

Analogie avec le dictionnaire. On ne veut pas retrouver l'information qu'on connait déjà (le mot) mais une autre information qui y est associée (définition). En général, on a dans une mémoire adressable par le contenu des couples (clé, valeur).

Primitives :

— rechercher(clé)
— insérer(clé, valeur)
arbitrage si clé est déjà présent
— supprimer(clé)

Il y a des variations suivant les stratégies de remplacement, et en général en fonction de la manière dont on les construit.

Normalement vous verrez l'implémentation de ces choses là en logiciel. Je présenterai les implémentations matérielles quand j'en aurai besoin (elles sont plus rares).

4.2 Construction des mémoires

Remerciements : Toutes les figures compliquées de ce chapitre sont coupées d'un corrigé de TD rédigé par Jérémie Detrey. Des étudiants comme cela, on n'en fait plus.

4.2.1 Le verrou (latch) et le registre (Flip-Flop)

La figure 4.2 montre un petit circuit, construit à partir d'un multiplexeur, qui n'est absolument pas un circuit combinatoire bien formé (pourquoi ?).

On l'appelle *verrou*. Il recopie D (*data*) sur Q lorsque Keep vaut 0, et garde l'état précédent lorsque Keep vaut 1. Remarquez que ce n'est pas encore mon registre : il ne mémorise que la moitié du temps.

Et pourquoi la sortie s'appelle Q ? Je ne sais pas, elle s'appelle toujours comme ça [2].

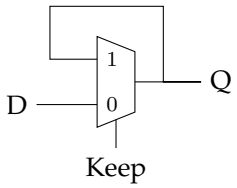


FIGURE 4.2 – Un verrou. Exercice : mettez des flèches sur les fils.

Sur la figure 4.3, on colle l'un derrière l'autre deux verrous fonctionnant sur les états différents de Keep. Sauf que du coup Keep est renommé Horloge (*clock* ou Ck).

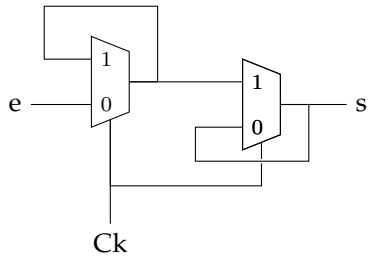


FIGURE 4.3 – Construction du flip-flop. Exercice : mettez des flèches sur les fils.

Convaincons-nous qu'on obtient le registre (ou Flip Flop) de la figure 4.1 p.39.

Le premier verrou est passant sur l'horloge basse, le second sur l'horloge haute. Ainsi, lorsque l'horloge vaut 0, la valeur de e est placée à l'entrée du second verrou (flip). Mais le second verrou ne la laisse pas passer. Jusqu'à ce que l'horloge passe de 0 à 1 (flop) : alors la valeur de e est mémorisée dans le premier verrou (et transférée à la sortie par le second verrou, qui est devenu passant). Lorsque l'horloge repasse à 0, c'est le second verrou qui mémorise à son tour la valeur (qui est toujours la valeur qu'avait e au front montant précédent). Le premier redevient passant mais on s'en fiche : on est revenu au début du paragraphe.

Ce flip-flop est une mémoire statique (l'information est stable, on peut alimenter une autre porte avec) mais volatile (elle disparaît quand on coupe l'alimentation).

2. Dans la vraie vie, c'est à dire dans les vieux bouquins recopiés sur Wikipedia, vous trouverez des circuits qui sont des verrous mais qui sont appelés bascule RS et bascule JK, et des variantes de flip-flop qui sont appelés bascule D. C'est historique mais je n'aime pas car cela prête à confusion : le mot "bascule" est pour cette raison banni de ce poly. Pour les 0.5% d'entre vous qui auront besoin de se frotter à la vraie vie, comprenez bien la différence entre un verrou et un flip-flop, et tout ira bien.

On aura souvent besoin d'y ajouter une entrée *reset* qui définit son état initial. Ce sera un multiplexeur sur l'entrée e.

On aura aussi souvent besoin d'y ajouter une entrée *clock enable* (ou ce). Comme son nom ne l'indique pas, ce sera également un multiplexeur sur l'entrée e. On verra dans la suite qu'il vaut mieux s'interdire de mettre de la logique sur l'horloge. *Parfois cette entrée s'appelle WE pour Write Enable. Parfois elle s'appelle Load.*

4.2.2 Autres technologies de point mémoire

On sait faire des points mémoire *dynamiques* : en stockant l'info juste dans une capacité. Dessin. Avantage : c'est plus petit. Inconvénient : la lecture est destructrice. De plus la capacité se décharge lentement, il faut la regénérer périodiquement.

Question : laquelle est la plus rapide ?

Réponse si vous avez suivi les transistors de la dernière fois : le point mémoire statique est plus rapide, puisqu'il peut fournir plus de courant (dans un point mémoire dynamique, on utilise le condensateur le plus petit possible).

On sait aussi faire des mémoires *non volatiles* : par exemple la surface d'un disque dur, avec des particules qui peuvent être aimantées dans un sens ou dans l'autre. Dans le CDRW on a deux états stables d'un alliage (amorphe ou cristallin).

Mémoire flash : on isole la grille d'un transistor, et on y piège des électrons par claquage. C'est une mémoire statique et non volatile, haute densité, à lecture non destructrice (puisque le transistor reste bloqué ou passant). Inconvénient : écriture relativement lente (haute tension), seulement 100 000 cycles d'écriture.

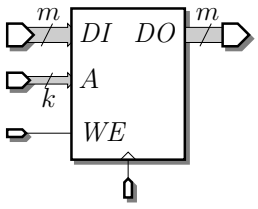
Actuellement, il y a pas mal d'agitation pour trouver la mémoire "idéale" :
— petit point mémoire
— rapide
— non volatile, 10 ans de rétention d'information
— résistante à 10^{15} cycles de lecture/écritures
— pas chère à construire

Aucune des solutions existantes (googlez MRAM, FeRAM, RRAM, etc.) ne réunit toutes ces qualités... mais il y a un gros gros marché pour le premier qui y arrive.

4.2.3 Mémoire adressable, première solution

L'interface d'une mémoire $2^k \times m$ comprend :
— un signal d'adresse A sur k bits,
— un signal de données entrantes DI sur m bits,
— un signal de données sortantes DO sur m bits,
— un signal d'horloge Clk, et
— un signal *write enable* WE.

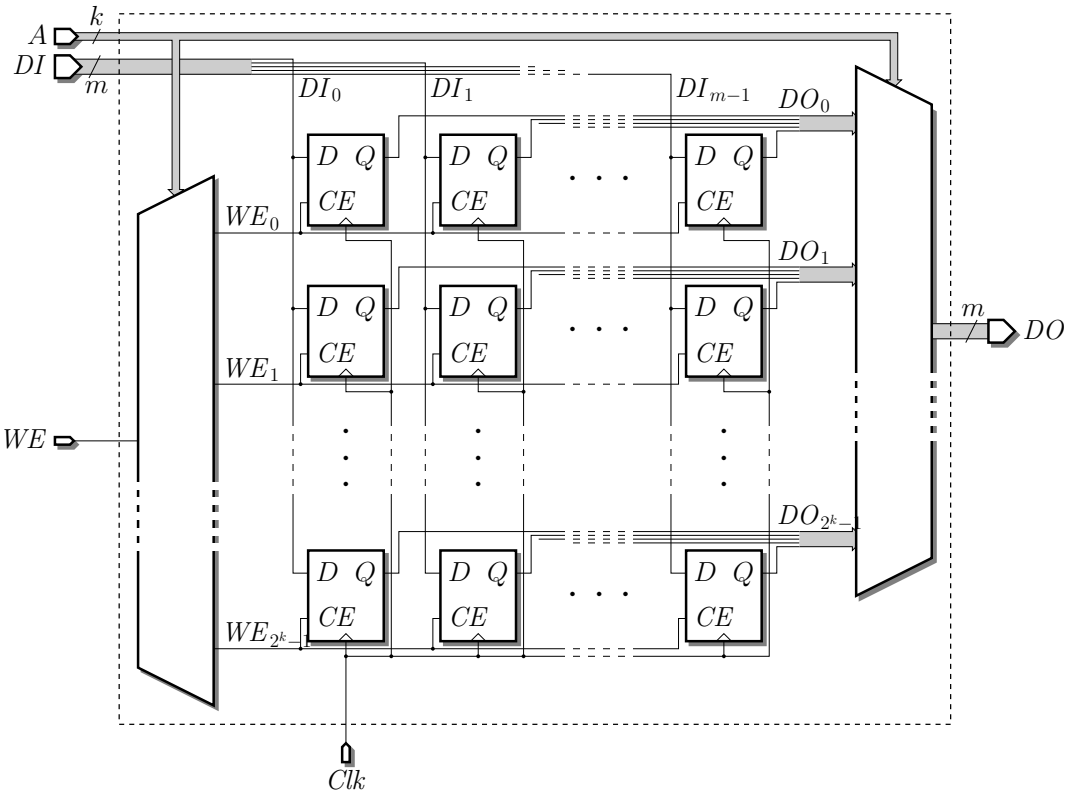
On la schématise de la manière suivante :



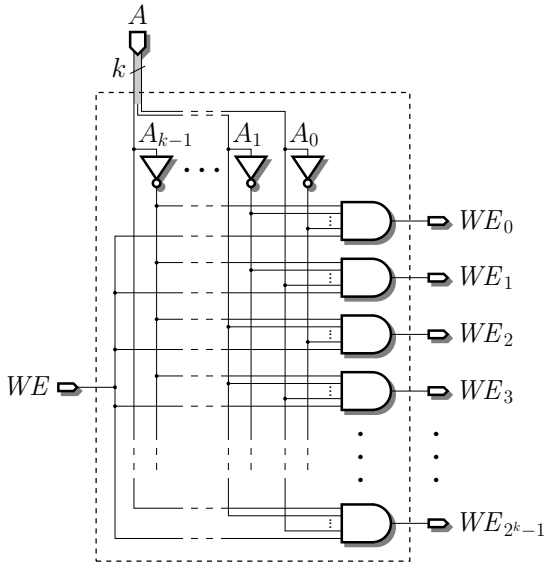
Elle repose sur une grille de 2^k lignes de m registres/flip-flops munis de *clock enable*. Chacun des m bits de DI est distribué aux 2^k registres correspondants, le choix du registre dans lequel la donnée sera écrite étant réalisé grâce aux CE.

En effet, en démultiplexant le signal WE selon l'adresse A, on obtient 2^k signaux WE_i, chacun distribué aux registres de la ligne correspondante.

Enfin, les 2^k sorties DO_i des lignes de registres sont multiplexée selon A pour sélectionner la valeur de la ligne demandée.

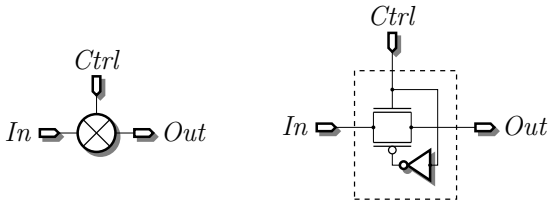


En détail ici, la construction du démultiplexeur pour WE :



4.2.4 Mémoire adressable, seconde solution

On se donne désormais aussi la porte trois-états (aussi appelée porte de transmission). Elle agit comme un interrupteur. Ci-dessous son schéma et sa réalisation à l'aide d'un inverseur et de deux transistors CMOS :

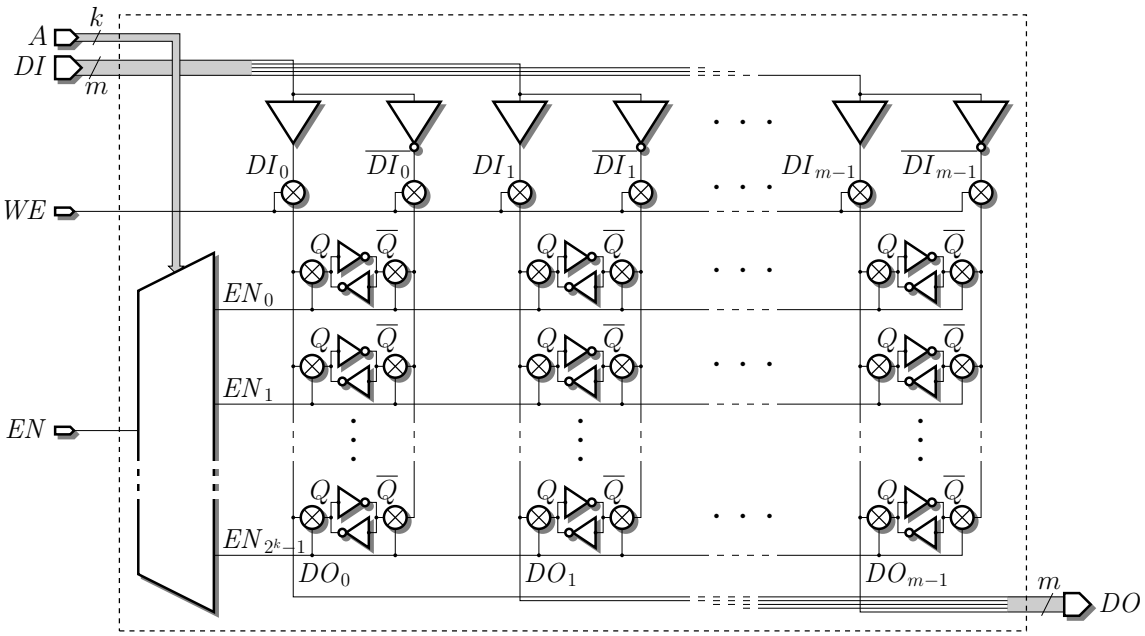


On se donne ce coup-ci une interface sensiblement différente pour la mémoire. On supprime le signal d'horloge *Clk* et on rajoute un signal *enable* noté EN qui contrôle le moment où les données sont lues ou écrites.

C'est ce signal EN qui va être démultiplexé selon l'adresse A en $2^k - 1$ lignes EN_i. De même que pour la version précédente, le signal EN_i de chaque ligne va être distribué aux m cellules mémoire composant cette ligne.

Pour éviter d'utiliser un multiplexeur pour sélectionner la bonne valeur de DO, on fait ici appel aux portes trois-états : on se donne m fils verticaux DO_j, correspondant chacun à un bit de DO. Grâce aux portes trois-états, on peut isoler ce fil DO_j de la sortie Q de chacun des éléments de mémorisation correspondant au bit de rang j, sauf pour la ligne i sélectionnée par le signal EN_i.

Pour pouvoir écrire dans un élément de mémorisation, on peut réutiliser ces fils verticaux DO_j et imprimer la valeur DI_j dessus, avec suffisamment de puissance pour écraser l'ancienne valeur Q contenue dans les cellules sélectionnées. Comme les éléments de mémorisation choisis ici sont de simples bistables, il faut aussi changer la valeur de $\overline{Q}$ en même temps que celle de Q. On dédouble donc les fils verticaux pour former les paires DI_j et $\overline{DI_j}$. L'amplification de ces deux signaux est réalisée avec de gros buffers / inverseurs. Enfin, le signal WE vient contrôler des portes trois-états sur ces fils pour les déconnecter des colonnes de mémorisation en cas de lecture.



Et pourquoi on a fait tout cela ? Eh bien comptez les transistors par point mémoire : on est passé de 18 (un flip-flop) à 8 (2 par inverseur, 2 par porte de transmission).

Par ailleurs, une mémoire dynamique va ressembler beaucoup à cette seconde solution.

Enfin, pour obtenir un circuit plus carré, on peut découper les fils d'adresse en deux moitiés, qui vont adresser lignes et colonnes.

4.2.5 Piles, files, etc

On n'a pas trouvé mieux qu'utiliser des RAM et des compteurs pour implémenter les piles et les files. Ceci est laissé en exercice pour le lecteur, qui peut toujours essayer de demander au prof si l'exercice lui semble trop difficile.

4.2.6 Disques (*)

Piste, secteur.

Temps d'accès (qq ms), débit (qq Mo/s).

Les disques sont de petits objets sensibles plein de pièces mouvantes et fragiles. Solution : disques RAID *(redundant array of inexpensive disks)*. Pour vous faire utiliser des codages redondants.

4.3 Une loi fondamentale de conservation des emmerdements

Comme l'illustre la table 4.1, une mémoire de grande capacité est lente ; une mémoire rapide est limitée en taille, et chère.

Remarque : ce n'est pas juste une question de compromis technologique (par exemple statique contre dynamique), c'est aussi une question de complexité algorithmique :

- *plus votre mémoire a une grande capacité, plus elle est étendue dans l'espace, et donc plus déplacer certains de ses bits vers la sortie (même à la vitesse de la lumière) prendra du temps ;*
- *pour une mémoire de capacité n, le décodage d'adresse utilise $\log_2 n$ étages d'aiguillages : plus votre mémoire a une grande capacité, plus l'accès sera lent.*

Type	temps d'accès	capacité typique
Registre	0.1 ns	1 à 128 mots (de 1 à 8 octets)
Mémoire vive	10 - 100 ns	4 Goctets
Disque dur	10ms	100Goctets
Archivage	1mn	(illimité)

TABLE 4.1 – Hiérarchie mémoire simplifiée d'un ordinateur de 2010. Voir la table 12.1 page 105 pour la version complète.

Remarques :

- Cette table4.1 est tout entier victime de la loi de Moore.
- Le coût de chaque étage est grosso modo équivalent. Le coût par octet est donc exponentiel par rapport à la vitesse d'accès.

On verra plus tard (au chapitre 12) comment construire des mécanismes qui donnent l'illusion que que toute la mémoire vive est aussi rapide que les registres (mémoire cache) et aussi grande que son disque (mémoire d'échange ou *swap*).

Chapitre 5

Circuits séquentiels synchrones

On a vu au chapitre 3 comment implémenter des fonctions booléennes sous forme de circuits combinatoires. Je ré-insiste lourdement sur le fait que ce sont des fonctions au sens mathématique ($f : x \mapsto f(x)$), c'est-à-dire sans mémoire. Leur sortie ne dépend que de l'entrée, pas de ce qui s'est passé avant.

On a vu au chapitre 4 comment construire des mémoires.

Dans ce chapitre, nous allons fusionner tout cela pour construire des circuits plus généraux, mêlant calcul (combinatoire) et mémorisation. On appellera de tels circuits des circuits séquentiels.

5.1 Quelques exemples de circuits séquentiels

Voici des exemples de circuits qui sont séquentiels et non pas uniquement combinatoires :

— Un compteur, obtenu en reliant un additionneur et un registre de n bits, comme illustré sur la Figure 5.1. Dans le cycle de von Neumann, le PC avance grâce à un circuit similaire.

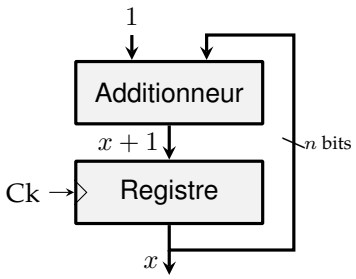


FIGURE 5.1 – Compteur automatique de moutons pour s'endormir plus vite (patent pending). Il a déjà compté jusqu'à x moutons et s'apprête à compter le $x+1$-ième.

— Votre montre à quartz numérique[1] est pleine de compteurs : un pour les secondes, un pour les minutes, etc. C'est un circuit séquentiel.
— Le circuit caché derrière votre digicode. Il doit se souvenir des chiffres déjà tapés ; il a une mémoire, c'est un circuit séquentiel.
— la mémoire adressable du chapitre précédent a évidemment une mémoire
— Votre téléphone, votre console de jeux, votre PC sont des circuits séquentiels.

Bref, la classe des circuits séquentiels est la plus générale.

1. Une vachté de chouette idée...

5.2 Restriction aux circuits séquentiels synchrones

Dans tout ce cours, on va faire une grosse restriction sur les circuits séquentiels : **ils ne devront avoir qu'une seule horloge**. Plus précisément, **les entrées d'horloge de tous les registres binaires devront utiliser le même signal d'horloge**. La classe de circuits obéissant à cette restriction est la classe des *circuits séquentiels synchrones*.

Remarque : il y a n registres binaires cachés dans la Figure 5.1, et ils utilisent bien tous le même signal d'horloge.

La raison de cette restriction est qu'elle rend les circuits infiniment plus facile à concevoir, comprendre, vérifier et fabriquer. La preuve : vous allez apprendre à faire tout cela dans les pages qui suivent.

Ce n'est pas une restriction pédagogique : l'écrasante majorité des circuits manipulés dans l'industrie sont des circuits synchrones. En effet, l'industrie aussi veut comprendre les circuits qu'elle fabrique, pour fabriquer des circuits qui marchent. Sans cette restriction, on arrive vite à des circuits séquentiels indébuggables, illisibles, imprévisibles, et en général mal élevés (ce n'est pas forcément évident, alors croyez-moi sur parole).

On observe au passage que les circuits séquentiels contiennent des parties qui sont des circuits combinatoires (par exemple l'additionneur sur la Figure 5.1). On continue à s'interdire, dans ces parties combinatoires, ce qui était interdit jusque là (brancher deux sorties ensembles, faire un cycle). Par contre, on se permet des cycles dans les circuits séquentiels, **à condition qu'il y ait au moins un registre sur chaque cycle**. Remarquez que c'est le cas de la la Figure 5.1.

Un circuit séquentiel qui n'est pas synchrone *On peut construire un compteur uniquement en branchant entre eux n flip-flops. Je le dessinerai au tableau uniquement, pour être sûr que vous ne le retenez pas. Il n'est pas synchrone bien qu'il n'ait qu'une seule horloge en entrée, parce que les entrées d'horloge de certains registres n'utilisent pas cette horloge. Il est plus économique que le compteur de la Figure 5.1, mais il a aussi des défauts :*

- *Il n'est pas du tout évident que c'est un compteur en le regardant (alors que la Figure 5.1 se comprend vite).*
- *Comment le transformer en un compteur qui revient à 0 quand il atteint 60 (pour votre montre à quartz numérique) ? Sur la Figure 5.1, on ajoute un comparateur à 60 et un multiplexeur et c'est dans la poche.*

Désormais, on se permettra de dire "circuit synchrone" au lieu de "circuit séquentiel synchrone" qui est, du reste, redondant.

5.3 Correction et performance

Observons ce qui se passe lors d'une transition sur l'horloge dans un circuit synchrone, par exemple notre compteur.

Partons d'un front montant de l'horloge unique. Comme tous les registres utilisent ce signal d'horloge, toutes les sorties de registres basculent en même temps. Ces sorties de registres sont les entrées de la logique combinatoire qu'il y a entre les registres. À partir de cet instant, les informations font la course en se propageant dans ces parties combinatoires. Le chronogramme de la figure 5.3 décrit le déroulement de cette course. Pour la comprendre, il faut regarder la figure 5.2, copiée de la page p.32, qui montre comment est construit l'additionneur.

Toutes les entrées de cet additionneur basculent au même instant (par exemple au premier front montant de clk sur la figure 5.3). Un peu plus tard, le bit s_0 se stabilise, puis le bit s_1 et ainsi de suite. En effet, chaque FA doit attendre que son entrée de retenue (c_i) soit stabilisée, donc que le FA à droite soit stabilisé. On a donc une propagation de la retenue dans l'additionneur, avec des transitoires sur les sorties (on a parlé des transitoires en 3.2.6 p. 29). Ces transitoires sont

difficile à prévoir, et comme elles ne nous intéressent pas, on les dessine juste comme du gris sur la figure 5.3).

Mais au bout d'un certain temps, les sorties de l'additionneur sont toutes stabilisées. On est alors dans la situation décrite sur la figure 5.1 : il y a une valeur x en sortie des registres, et une valeur $x + 1$ en entrée des registres. Au front suivant de l'horloge, tous les registres basculeront en même temps. La valeur $x + 1$ sera alors stockée dans les registres, donc présentée à leur sortie, et un nouveau cycle pourra commencer.

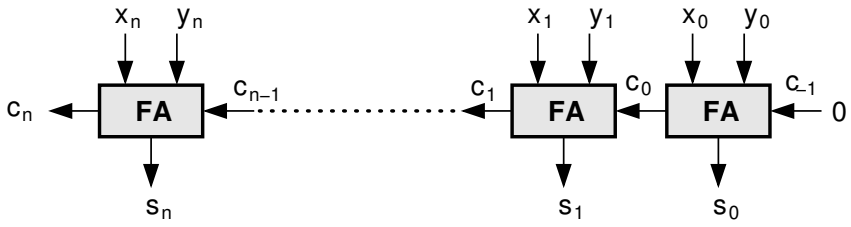


FIGURE 5.2 – Rappel : l'intérieur de la boite "Additionneur" de la figure 5.1.

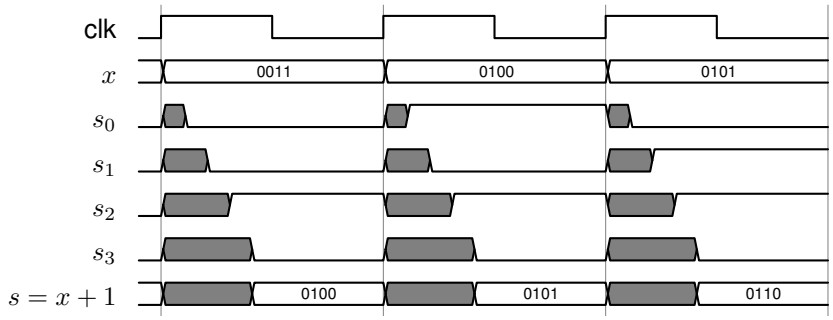


FIGURE 5.3 – Chronogramme détaillé de de qui se passe dans le compteur. Le gris signifie "inconnu" ou encore "transitoires possibles".

Il y a deux conditions pour que ce compteur fonctionne comme décrit.

— Il faut que tous les registres basculent en même temps.

— Il faut que le délai entre deux fronts montants de l'horloge soit plus grand que le temps de calcul au pire cas de notre additionneur.

On a déjà parlé p. 29 du chemin critique dans un circuit combinatoire : c'était le chemin (à travers des portes et des fils) le plus long, c'est-à-dire qui va propager l'information le plus lentement. On généralise cela aux circuits synchrones : on appelle **chemin critique** d'un circuit synchrone le plus long chemin combinatoire, autrement dit le plus long chemin (toujours en terme de temps de propagation de l'information) entre une sortie de registre et une entrée de registre. Sur l'exemple du compteur, c'est le temps d'une propagation de retenue au pire cas, illustrée sur le chronogramme.

Le délai entre deux fronts montant d'horloge, c'est l'inverse de la fréquence d'horloge (le fameux 3GHz de votre PC). Au bout du compte, c'est donc le chemin critique qui définit la fréquence maximale à laquelle fonctionnera correctement le circuit.

Oui mais la perfection n'existe pas *Bien sur, en pratique on ne peut pas physiquement garantir que tous les registres basculent en même temps : on aura toujours un δt d'incertitude, due à des différence entre les longueurs des fils, la variabilité du processus de fabrication du circuit, etc. Qu'à cela ne tienne, on va borner cette incertitude, et il suffira d'ajouter ce δt au chemin critique : la condition pour que le circuit fonctionne devient "chemin critique + δt < période d'horloge".*

Maintenant, considérons un circuit séquentiel synchrone compliqué, par exemple la puce de votre pléstation dualscrine. Il a des entrées (les boutons de la console entre autres), des sorties (les pixels de l'écran entre autres), et plein de registres à l'intérieur. Et entre les registres, plein de portes branchées les unes aux autres. Eh bien le fonctionnement est similaire : tous les registres basculent en même temps. L'information fait la course dans les portes entre les registres. Au bout d'un certain temps elle est stabilisée. Ce temps s'appelle le délai du chemin critique. Alors on peut basculer l'horloge pour attaquer un nouveau cycle.

Si on prend un peu de recul, les conditions que nous avons imposées sur les circuits séquentiels synchrones permettent de cacher toutes les transitoires. Si on n'observe que les valeurs stockées dans les registres, ce qu'on voit est le résultat des fonctions booléennes, on a complètement caché leur implémentation. On peut ainsi raisonner sur un circuit, y compris séquentiel, en terme de fonctions booléennes. C'est bien, parce que c'est abstrait et que cela cache tous les sordides détails d'implémentation.

C'est comme cela qu'on arrive à construire des circuits séquentiels comportant des milliards de portes, et qui marchent quand même.

Pourquoi on impose que tous les registres ont la même horloge *Si on commence à se permettre de mettre des portes sur une entrée d'horloge d'un registre, on va devoir tenir compte de leur délai. On peut essayer de faire comme pour notre δt : calculer le chemin critique des portes qui sont sur l'entrée d'horloge, et l'ajouter séparément aux délai des chemins qui partent de ce registre, mais pas aux autres... C'est faisable mais vous voyez que cela devient vraiment compliqué. Il faudra être très convaincant qu'il y a quelquechose à gagner. Dans le doute, je vous invite à n'envisager que des circuits dans lesquels tous les registres ont la même horloge.*

On assure le synchronisme par la construction d'un *arbre d'horloge équilibré* : dès qu'on a de nombreuses entrées d'horloge à alimenter avec le même signal, il faut l'amplifier, par un arbre d'inverseurs. Un inverseur typique est capable d'alimenter 4 entrées rapidement, au dela de 4 sa performance chûte : on dit qu'un tel inverseur a un *fan out* de 4. Pour obtenir un arbre de distribution d'horloge équilibré à l'échelle d'une puce rectangulaire, on le construit en arbre quaternaire, comme sur le dessin de la figure 5.4.

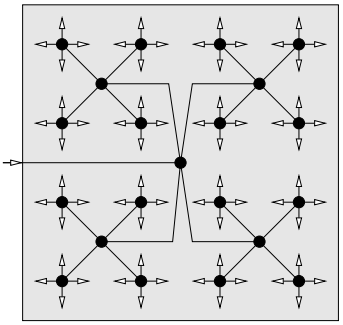


FIGURE 5.4 – Arbre d'horloge équilibré. Les ronds noirs sont des inverseurs.

Chapitre 6

Automates

On avait le formalisme de la logique booléenne pour construire les fonctions combinatoires.

Nous allons présenter dans ce chapitre un formalisme, les automates [1] , qui permet de décrire, vérifier, et même construire automatiquement des circuits séquentiels.

Les automates sont une abstraction facile à dessiner du *comportement attendu* d'un circuit. Le comportement, c'est ce qu'on peut décrire de plus abstrait ! De ce point de vue, l'automate est le pendant d'une fonction booléenne, qui décrit le comportement d'un circuit logique combinatoire.

6.1 Un exemple

Essayons de spécifier le comportement d'un circuit de commande d'un passage à niveau en Russie, entre les villages de Krasnoe Kerpitchnik et Tchaplova (emplacement choisi parce que c'est une voie unique où peuvent passer des trains dans les deux sens, on n'a plus cela chez nous). On place des capteurs assez loin à gauche et à droite du passage à niveau. Ce sont de simples interrupteurs déclenchés par le poids d'un train : ils renvoient un signal logique 0 quand la voie est libre, et 1 lorsqu'il y a un train dessus. On a par ailleurs un feu rouge, également abstrait par un booléen A (comme ampoule) : $A = 0$: éteint ; $A = 1$: allumé.

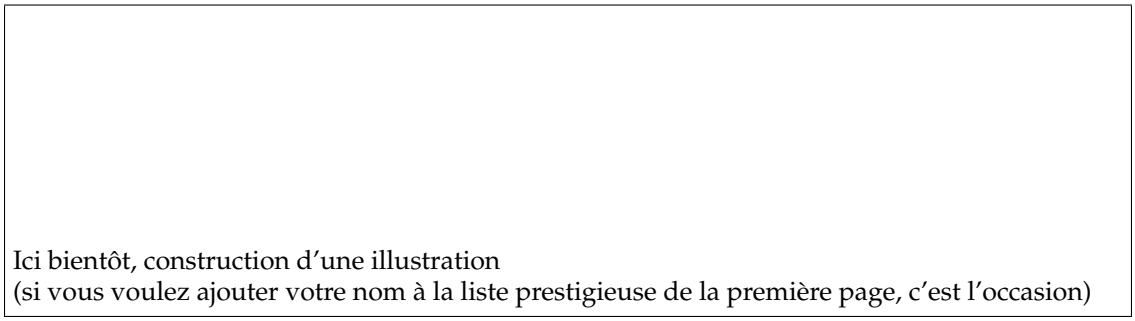


FIGURE 6.1 – Le communisme, c'est les soviets plus l'électrification des passages à niveaux.

On aimerait que le feu s'allume lorsque le train s'approche, et s'éteigne lorsque le train est passé. Convainquons nous qu'on ne s'en tirera pas par une simple fonction combinatoire des capteurs. Par exemple, parfois il passe de simples locomotives : il faut que notre circuit se souvienne de garder le feu allumé tant que la locomotive n'a pas quitté le second capteur.

1. Le vrai nom est *automate à nombre fini d'états*, parfois mal traduit en *automate d'états fini*, de l'anglais *finite state automaton* (c'est l'*automaton* qui est fini, pas le *state*), ou encore *finite state machine*. Tout ceci pour dire que vous verrez parfois les acronymes FSM ou FSA.

La figure 6.2 décrit l'automate correspondant (seule une moitié a été complètement spécifiée, car c'est un poly interactif : vous devez gribouiller l'autre moitié pour vérifier que vous avez compris).

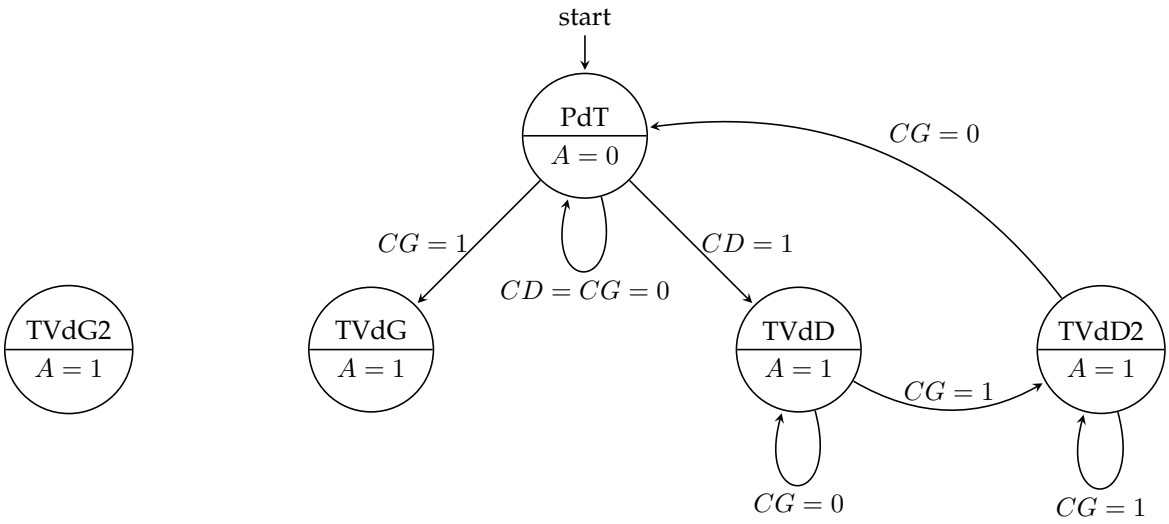


FIGURE 6.2 – Mon premier automate. CG et CD sont les *Capteurs Gauche* et *Droite*. TVdG est l'abréviation de *Train Vient de Gauche*. Completez vous-même l'autre côté.

Les patates sont des états. Un état a un nom, par exemple PdT. Il spécifie également les valeurs des sorties (ici on n'en a qu'une) : $A = 0$ dans l'état *Pas de Train*.

Entre les états, on a des flèches qui indiquent un changement d'état. On appelle ces flèches des transitions. Une transition est étiquetée par l'évènement qui déclenche le changement. Par exemple, on reste dans l'état *Pas de Train* tant que $CG = CD = 0$, et on passe dans l'état *Train Vient de Gauche* lorsque le capteur gauche passe à 1.

Par convention, une flèche sur laquelle une entrée n'apparait pas signifie que la transition est prise quelle que soit la valeur de cette entrée.

Attention, il y a désormais dans ce poly deux sens techniques du mot *transition* : on connaissait une transition entre deux valeurs d'un signal (un front montant ou descendant), on a désormais aussi la notion de transition d'un état à un autre.

Notez au passage que, comme pour les variables dans un programme, cela ne fait pas de mal de donner des noms expressifs aux états, aux entrées et aux sorties.

Au tableau je montre l'automate du digicode, qui reconnait un code.

6.2 Définition formelle d'un automate synchrone

6.2.1 États, transitions

Allons-y donc pour le formalisme. Un automate synchrone tels qu'on les manipule en architecture est un quintuplet (I, O, S, T, F, s_0) où

— I est un ensemble fini d'entrées binaires,
— O est un ensemble fini de sorties binaires
— S est un ensemble fini d'états.
— T est une *fonction de transition* de $S \times I \to S$
— F est une *fonction de sortie* de $S \to O$
— $s_0 \in S$ est un état spécial appelé *état initial*, car il en faut bien un.

I et O définissent la boîte noire du circuit, S, T et F décrivent le comportement.

On fait des dessins avec

— des ronds pour les états, qu'on étiquette avec le nom de l'état, et la valeur de F correspondante
— des flèches d'un état s à un état s', qu'on étiquette avec une valeur x des entrées, chaque fois que $T(s, x) = s'$.

6.2.2 Définition en extension des fonctions de transition et de sortie

Les deux fonctions sont des fonctions discrètes, c'est-à-dire avec un nombre fini d'entrées et de sorties possibles. On pourra donc les définir en extension, en listant leurs valeurs dans une table. C'est pourquoi on parle toujours d'automate fini. Par exemple :

s	x=(CG, CD)	s'=T(s,x)
PdT	00	PdT
PdT	01	TVdD
PdT	10	TVdG
PdT	11	XXX
TVdD	0X	TVdD
TVdD	1X	TVdD2
TVdD2	1X	TVdD2
TVdD2	0X	PdT
...	...	...

s	y=F(s)
PdT	0
TVdD	1
TVdD2	1
...	...

Que signifie le X dans ces tables ? Il signifie *don't care*, ce qui peut se traduire par "on s'en fout".

Mais attention, cela ne veut pas tout-à fait dire la même chose quand le X est en sortie de la table, et quand il est en entrée.

En entrée, on met un "X" sur une valeur d'entrée pour dire "quelle que soit la valeur de cette entrée". Dans notre exemple, pour chaque état, on devrait considérer les 4 combinaisons de valeurs possibles de nos deux capteurs. Mais dans l'état TVdD, le dessin à patates n'a que deux flèches. Il est donc incomplet. Dans la table de transition, on a précisé, par exemple qu'on prend la transition qui boucle de TVdD à lui-même tant que CG vaut 0, quelque soit la valeur de CD. En effet c'est bien ce qu'on veut : à partir du moment ou un train vient de la droite, le feu est allumé, et ce qui nous intéresse est de savoir quand on l'éteindra. Et ce ne sera pas lorsque CD passe à 0, mais lorsque CG passera à 0. Mais avant CG doit passer à 1 : c'est cet évènement qui déclenchera la seule transition sortant de TVdD. Peut-être qu'à ce moment CD sera déjà repassé à 0 (train court) ou pas encore (train long), mais on s'en fout.

En pratique, un *don't care* en entrée peut être remplacé par une énumération des transitions qu'il élide. Par exemple,

TVdD	0X	TVdD

signifie exactement

TVdD	00	TVdD
TVdD	01	TVdD

En sortie (des fonctions T ou F), le "don't care" est plus fort... et plus dangereux. Il signifie "cette situation n'arrivera jamais, donc n'importe quelle valeur fera l'affaire". Par exemple, considérons la quatrième ligne de la table de T ci-dessus. On est dans l'état "pas de train" avec les deux capteurs à 0, et voila que tout à coup, les deux capteurs passent à 1 en même temps. Cela signifie sans doute que deux trains sont en train de se précipiter l'un sur l'autre : il va bientôt avoir une épouvantable catastrophe ferroviaire. Et donc, on s'en fout. En effet, ce cas ne doit jamais arriver. Autre point de vue : l'automate peut bien se retrouver dans l'état qu'il veut, on aura d'autres chats à fouetter si deux trains se rentrent dedans.

En sortie, le *don't care* se lit donc plutôt "ce n'est pas mon problème", et vous voyez que pour l'ingénieur que vous êtes, cela peut être dangereux. Dans l'exemple du feu rouge, il y a peut-être plus intelligent à faire. Par exemple, aller dans un état où le feu est allumé, pour ne pas ajouter un petit accident au gros. Ou bien aller dans un état "erreur" qui va faire venir un technicien. Parce que l'allumage des deux capteurs vient peut-être aussi d'un faux contact quelquepart. Etc.

6.2.3 Correction et complétude d'un automate synchrone

Revenons aux patates. On ne peut pas vérifier que votre automate répond bien aux spécifications fonctionnelles de votre circuit. Par contre, on peut vérifier un certain nombre de propriétés sans lesquelles il est forcément incorrect :

déterminisme : partant du même état, il n'y a pas deux transitions qui arrivent dans des états différents pour une même valeur des entrées.

complétude : dans chaque état,
— la valeur des sorties est définie.
— il y a une transition prévue pour chaque valeur des entrées (fût-ce à l'aide de *don't care*).

Ces deux propriétés sont vérifiables mécaniquement : il suffit d'essayer de construire complètement la table de transition et la table de sorties.

6.3 Synthèse d'un automate synchrone

Il ne reste plus qu'à coder les états par des bits. C'est toujours possible puisqu'ils sont en nombre fini. Par exemple, on peut numéroter les états, puis coder chaque état par la représentation binaire de son numéro. Alors, on peut remplacer s et s' par des vecteurs de bits dans les tables ci-dessus. Ainsi, T et F deviennent des fonctions booléennes. On sait donc les implémenter grâce au chapitre 3.

Et alors, on sait implémenter un automate par le circuit générique décrit par la figure 6.3.

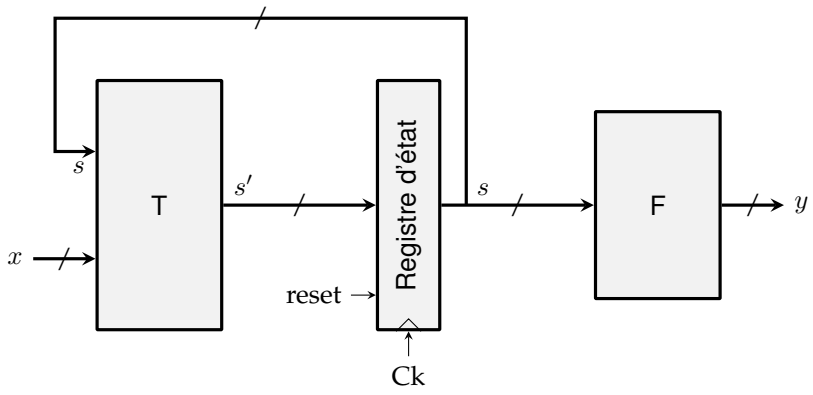


FIGURE 6.3 – Implantation d'un automate synchrone. Notations : s est l'état courant, s' est le prochain état, x est l'ensemble des entrées, y est l'ensemble des sorties. *Notez que le reset n'est pas considéré comme une entrée de l'automate : il n'apparait pas sur les transitions, il est la traduction de l'état initial.*

Explication de ce circuit : au front montant de l'horloge, l'ancien état s est remplacé par le nouvel état s'. Lequel s' est calculé par la fonction de transition T à partir de l'ancien état s et des entrées x. Par ailleurs, les sorties sont calculées à partir de l'état s par la fonction F. C'est bien ce qu'on a dessiné avec des patates.

Et on se ramène à des choses qu'on connaît bien : des flip-flops pour construire le registre d'état, et des fonctions combinatoires [2].

Concrètement et bêtement, à partir d'un dessin de patates correct et complet,
— on choisit un codage des états sur k bits.

2. Il nous faut des flip-flop avec reset pour pouvoir définir l'état initial de l'automate. C'est facile à construire, il suffit de mettre les portes qui vont bien devant l'entrée du flip-flop (reset synchrone).

— on fait le dessin correspondant à la figure 6.3, en nommant les bits d'entrée, les bits de sortie, et les bits d'état.
— on trace le squelette de la table de vérité de T à partir de cela, en énumérant ses entrées possibles.
— on remplit les valeurs de T en suivant les patates du dessin de l'automate.

En fait il y a des tas d'outils automatiques qui feront cela pour vous.

6.3.1 L'approximation temporelle réalisée par l'automate synchrone

Oui mais on a décrit l'automate comme si il changeait d'état dès qu'un capteur passe de 1 à 0. Maintenant qu'on l'a construit, on constate que ce n'est pas tout-à-fait le cas. Il change d'état au front montant d'horloge suivant une transition des entrées. Ce n'est en général pas grave, il suffit d'avoir une fréquence d'horloge assez grande pour que ce petit délai soit négligeable. Par exemple, les trains russes roulant à 50km/h, notre circuit pourrait tourner à 100 Hz et ce serait bien assez. Or on sait le faire tourner à 1GHz.

Par contre, on comprend à présent l'intérêt de bien spécifier les transitions d'un état vers lui-même : dans notre exemple, ce seront les plus utilisées...

En pratique, il est souvent même utile d'ajouter des registres sur les entrées et sur les sorties. Cela permet de bien inclure tout F et tout T dans le calcul du chemin critique.

Pour avoir un circuit qui réagit vraiment instantanément à une transition sur une entrée, il faudrait envoyer cette entrée sur l'entrée d'horloge d'un registre d'état. Mais si on permet cela, ce n'est plus un circuit synchrone, et tout devient tout de suite beaucoup plus compliqué, comme vu au chapitre précédent. Donc je persiste à vous l'interdire.

Les automates en UML *Vous avez peut-être déjà vu des diagrammes état-transition en UML – sinon cela vous pend au nez. Dans ces diagrammes, les transitions sont déclenchées par des évènements de manière asynchrone. Elles peuvent être étiquetées par des actions, réalisées lors de la transition. Bref, c'est bien les même concepts, mais les automates UML sont plus généraux et ne sont pas tous traduisibles mécaniquement en la figure 6.3. Donc ne pas mélanger.*

Et pour finir, il subsiste un tout petit risque de *metastabilité* : si on a une transition d'une entrée trop près d'une transition d'horloge, certains registres peuvent se retrouver coincés entre le 0 et le 1. Cela passe tout seul, mais parfois en plusieurs cycles d'horloge [3]... Ce risque est bien isolé, et peut être géré par des solutions technologiques dans le détail desquelles je ne me risquerai pas.

6.3.2 Optimisation d'un automate synchrone

La question de la minimisation de la fonction combinatoire, qui était déjà un problème difficile, l'est encore plus ici, puisqu'on a un degré de liberté de plus : on peut choisir arbitrairement le codage des états de l'automate par des vecteurs de bits. Toutefois, ici encore, la connaissance du problème permet souvent d'imposer un codage des états qui va minimiser la complexité de la fonction de transition. Typiquement, il s'agit de remplacer le gros automate par plein de petits sous-automates relativement indépendants (c'est à dire avec relativement peu de transitions entre eux).

Exemple : une montre à quartz, d'un point de vue formel, est un automate à autant d'états qu'elle peut afficher d'heures différentes (dans les 24×60^2, sans compter les jours et les mois). En pratique, il est construit algorithmiquement par composition de petits automates à moins d'états, de taille totale en gros $24 + 60 + 60$.

Chacun des sous-automates ayant moins d'états et moins d'entrées, la fonction de transition a moins d'entrées, et est donc plus facile à minimiser.

Pour finir, on peut envisager deux extrêmes pour le codage des états :

— les codage minimaux en termes de bits : on code n états par un vecteur de $\log_2 n$ bits.

3. Amis gamerz, désormais, quand votre petite sœur vous met une pâtée à KickFighter XVII, vous pourrez accuser la metastabilité dans votre manette de jeu chaque fois que vous serez en retard sur une action...

— le codage à jeton *(one-hot encoding)* : on code n états par un vecteur de n bits.

Le second peut aboutir à un circuit plus petit si la fonction de transition s'en trouve très simplifiée.

6.4 Comprendre les circuits séquentiels comme des automates

Nous sommes partis de l'automate-patates pour le construire sous forme de circuit synchrone. Mais on peut aussi faire le contraire. L'exemple de la montre à quartz vu comme un automate à 24×60^2 états peut être généralisé : tout circuit séquentiel synchrone peut être vu comme un gros automate. Remarquez que notre compteur du chapitre précédent était un cas particulier de la figure 6.3.

Par contre le nombre d'états possible devient vite astronomique (et indessinable) : si vous avez n registres binaires dans votre circuit, alors vous avez 2^n états possibles de ce circuit. Par exemple, votre Pentium contient des dizaines de milliers de registres binaires. Tournez les pages vers l'introduction de ce poly : tous les atomes de l'univers ne suffiraient pas à faire une feuille de papier assez grande pour dessiner 2^{10000} patates.

Cette vue est cependant parfois utile, par exemple pour tester les circuits.

6.4.1 La norme JTAG

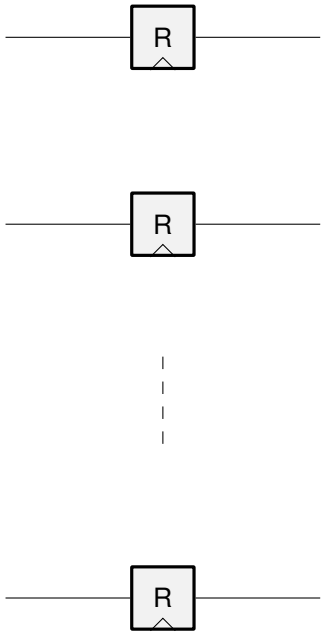
Comment teste-t-on un Pentium à $2^{10\ 000}$ états avant de le mettre en boîte ?

Bien sûr, on pourrait lui faire booter Linux puis Windows et jouer un peu à Quake dessus, et on se dirait qu'on a tout testé. Mais cela prendrait de longues minutes par puce, et le temps c'est de l'argent. Voici une technique qui permet de tester toute la puce en quelque centaines de milliers de cycles seulement (exercice si vous trouvez que c'est beaucoup : comptez combien de cycles à 4GHz il faut pour booter Linux en 20s).

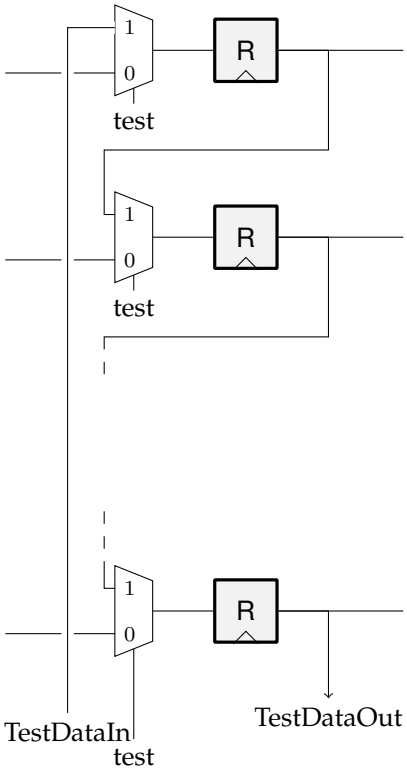
On considère l'ensemble du Pentium comme un gros automate selon la figure 6.3. Dans cette figure, le registre d'état contient tous les registres de la boîte à registres, tous les registres de pipeline, et même tous les registres de l'automate qui contrôle le Pentium. L'état du processeur, quoi.

On ajoute un tout petit peu de circuiterie pour faire de l'ensemble des 10 000 registres binaires un unique immense registre à décalage. C'est une transformation automatique qui est décrite par la figure ci-dessous :

Le registre d'état de la figure 6.3, avant...



... et après sa transformation en JTAG



Avec tout cela, on peut, en 10 000 cycles, mettre le circuit dans un état quelconque. On peut également, toujours en 10 000 cycles, lire l'état complet du processeur.

Une fois ceci en place, on teste le circuit comme ceci :

— On met *test* à 1, puis on pousse un état connu, pas forcément utile, dans le processeur.
— Puis on met *test* à 0, et on fait tourner le processeur pendant quelques centaines de cycles.
— Il fait sans doute n'importe quoi, mais ce n'est pas grave.
— On remet *test* à 1, et on sort l'état complet du processeur (tout en poussant un nouvel état à sa place).
— On compare l'état obtenu avec l'état dans lequel doit être le processeur si chacune de ses portes fonctionne correctement (obtenu par simulation).
— Et on recommence plusieurs fois, avec des états construits pour faire fonctionner tous les transistors de l'énorme fonction T – pas forcément des états dans lequel le processeur peut se trouver en fonctionnement normal.

Tout ceci est même normalisé par le Joint Test Action Group que vous avez peut-être croisé sous le nom de JTAG.

Mais ce n'est pas tout : une fois le circuit construit et emballé, on peut toujours utiliser ces nouvelles broches. En utilisant ce mécanisme, on peut observer ou changer la valeur de n'importe quel registre du processeur en quelque dizaines de milliers de cycle : on lit l'état, on change les bits qu'on veut, et on réécrit l'état modifié.

C'est comme cela que vous pouvez, en TP, observer dans un debugger ce qui se passe à l'intérieur d'un processeur ou d'un FPGA, à condition d'y avoir branché une "sonde JTAG".

6.4.2 Equivalence de circuits

Un autre problème, plus ou moins théorique, qui bénéficie de la vue d'un circuit comme un automate est l'équivalence de circuits. On dit que deux automates sont équivalents si, vu comme

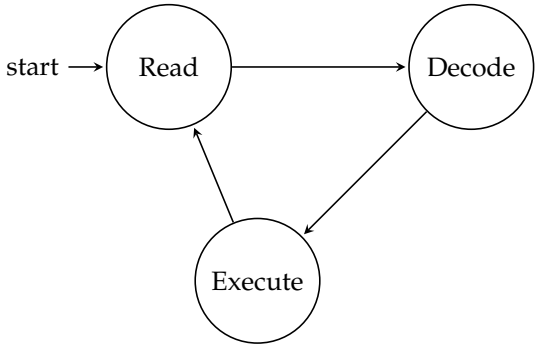


FIGURE 6.4 – Tiens, au fait, voici un automate simplifié pour le cycle de von Neumann. On le complexifiera dans la suite, par exemple parce que l'état "execute" sera décomposé en plusieurs sous-états, et le chemin pris dépendra de l'instruction à exécuter.

des boites noires, ils sont indiscernables. Formellement, pour toute suite de transitions sur les entrées, les deux automates donnent la même suite de transition sur les sorties.

6.5 Conclusion : l'ingéniérie des automates

En résumé, tout automate peut être traduit en un circuit synchrone d'une infinité de manière. Tout circuit synchrone peut aussi être vu comme un automate d'une infinité de manière.

Étant donnée la spécification d'un fonctionnement, il y a de bonnes et de mauvaises manières de dessiner un automate qui l'implémente. Un bon automate aura le moins d'états possibles, ou bien une fonction de transition simple. Il faut commencer par décomposer un fonctionnement compliqué en plein de petits automates simples, les entrées des uns étant les sorties des autres. Voir la montre à quartz.

L'encodage des états est important, et il faut y penser dès avant les patates. Par exemple on ne dessinera pas les 2^n patates pour le compteur, on dessinera directement la boîte noire parce qu'on sait l'implémenter intelligemment.. Pour un automate plus aléatoire, on pourra chercher les encodages des états qui rendent la fonction de sortie F triviale.

Bien qu'on ait des outils presse-bouton qui transforment les patates en circuit, il faut donc quand même de bons ingénieurs pour bien les utiliser.

Parenthèse culturelle : il n'y a pas des automates qu'en circuit et en UML *Les grands théoriciens des automates n'ont pas été les architectes mais les théoriciens du langage. En effet, dans vos compilateurs, c'est un automate qui reconnait un mot clé (il ressemble vraiment au digicode). C'est aussi un automate qui reconnait un nombre entier, un nombre en virgule flottante, un nom de variable. C'est même un automate (d'un type un peu plus compliqué) qui valide la syntaxe de tout votre programme.*
Vous verrez tout cela dans le cours "Grammaires et langages" en 4IF.
En théorie des langages, les automates n'ont pas vraiment d'action à réaliser, mais ils ont des états finaux (ou acceptants) et leur seule entrée est le prochain caractère du langage à reconnaître.
Nos automates à nous sont différents surtout en ce qu'ils auront des actions à faire (des sorties), et qu'on ne distingue pas certains états comme étant finaux/acceptants : ils tournent à l'infini.

Chapitre 7

Transmettre

7.1 Medium

Fil electrique, fréquence radio, fréquence lumineuse, rail avec des billes.

Notion de canal : vision logique du medium.

Notion de multiplexage : on envoie plusieurs canaux logiques sur un canal physique. Multiplexage temporel, multiplexage en fréquence...

Notion de débit (quantité d'information par unité de temps) et de bande passante d'un canal (quantité d'information pouvant passer en même temps dans le canal).

7.2 Liaison point a point

7.2.1 Série ou parallèle

Utilisation du temps, ou utilisation de l'espace. En général on combine intelligemment les deux : sur une liaison parallèle on envoie tout de même les données en série !

Exemples : RS232, I2C, USB pour le pur série.

7.2.2 Protocoles

Si on n'a qu'un seul canal binaire pour transmettre des données entre un émetteur et un récepteur, tout ce qu'on sait faire c'est faire passer ce canal de 1 à 0 puis de 0 à 1. Si on enlève l'information temporelle, on n'observe que 1010101010101010101...

Une communication suppose donc une *synchronisation* (partage du temps en grec dans le texte).

Une solution est qu'émetteur et récepteur aient chacun une horloge suffisamment précise, et se soient mis d'accord à l'avance sur les instants auxquels telle et telle donnée sont transmises. Ceci s'appelle un protocole de communication.

Une solution plus simple est d'avoir deux canaux binaires (au moins) entre émetteur et récepteur. L'un pourra faire passer le tic-tac d'une horloge, par exemple. Avec un seul canal mais au moins ternaire, c'est aussi possible. Mais dans tous les cas il faudra un protocole qui définit comment les données sont emballées, comment commence un paquet de donnée, etc.

Maître et esclave

Notion de maître (celui qui donne les ordres) et d'esclave (celui qui obéit). Le maître peut tourner, mais à un instant donné il vaut mieux qu'il n'y ait qu'un seul maître.

Protocoles synchrones

Utilisation du temps.

Exemple 1 : envoi d'une donnée de maitre à esclave, au moyen de deux canaux. Le maître positionne la donnée sur l'un, puis positionne l'autre canal, que nous appelerons "donnée prête"

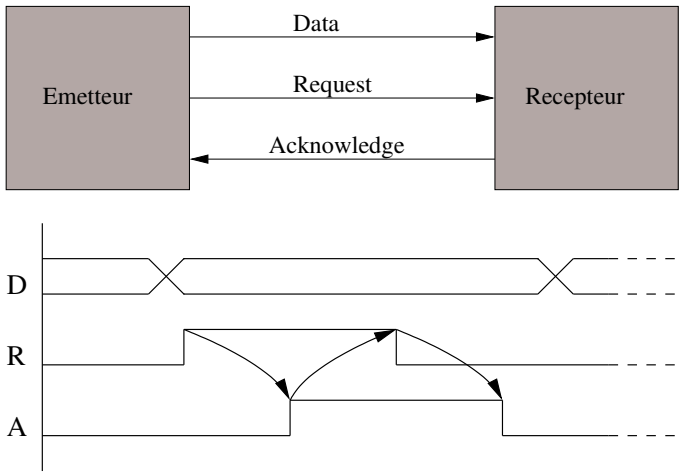


FIGURE 7.1 – Deux entités papotent en utilisant un protocole *handshake*. La donnée est une information d'état, alors que R et A portent des informations temporelles (changement d'état). En termes d'implémentations, Récepteur et Émetteur sont des automates : amusez vous à les dessiner.

et qui valait 0, à 1 pendant 1 pataseconde, et puis de nouveau à zéro. Il sait que l'esclave lui obéira et aura lu la donnée au bout d'une pataseconde : c'est son esclave.

Exemple 2 : réception d'une donné par le maître. Le maître positionne un signal "demande de donnée". Au bout de 1 pataseconde, il lit la donnée sur l'autre canal. Il sait que l'esclave aura obéi dans ce laps de temps.

Exemple 3 : deux fils dont un est un tic tac, envoi d'octets commençant par 10 (pour marquer le début d'un octet) puis 8 bits, puis parité pour la détection d'erreur.

Exemple 4 : Manchester encoding (ethernet) : 0 est codé par 0 pendant une pataseconde puis 1 pendant une pataseconde, 1 codé par 1 puis 0. Ainsi le recalibrage de l'horloge est inclus.

Ce type de protocole ne marche pas avec les circuits Insaliens. D'une part ils sont toujours en retard quand ils sont esclaves, d'autre part ils veulent tous être maître en même temps. C'est pourquoi on a mis au point des protocoles asynchrones.

Protocoles asynchrones

Protocoles qui ne font pas d'hypothèse sur les durées de transmission des signaux.

Archétype : protocole *handshake* pour "secouons nous les mains" en Breton. Il faut trois canaux : un de données (D), un de demande (R comme *request*), un d'acquittement (A comme *acknowledge*) (voir figure 7.1).

Au repos, tout le monde est à 0. Voici comment se passe une émission pilotée par l'émetteur. L'émetteur écrit sur R et D et lit sur A, le récepteur c'est le contraire.

— émetteur : "Voici une donnée" (positionne D, puis lève R, et le laisse à 1 jusqu'à nouvel ordre).
— récepteur : voit R levé, range la donnée ou il faut, puis dit "Bien reçu Chef" (lève A, et le laisse levé jusqu'à nouvel ordre).
— émetteur : voit A se lever, dit "Brave petit" (baisse R, puis attend la baisse de A avant de recommencer).
— récepteur : voit R se baisser, dit "ce fut un plaisir Chef" (baisse A et attend un nouvel ordre).

Attention, ici l'émetteur a l'initiative de la transmission, mais n'est pas maître de tout. Par exemple, il ne peut pas baisser R tant que le récepteur ne lui autorise pas en levant A.

On peut faire un handshake à l'initiative du récepteur : R signifie à présent "envoie une donnée SVP". C'est le récepteur qui écrit sur R et lit sur A (et toujours sur D), et l'émetteur écrit

toujours sur D mais lit sur R et écrit sur A.

Au repos tout le monde est à zéro.

— Récepteur lève R.
— Émetteur positionne la donnée, puis lève A.
— Récepteur traite ou range la donnée, puis baisse R.
— Émetteur baisse A.
— Récepteur voit A descendre, et sait qu'il peut demander une nouvelle donnée.

Il y a quand même une hypothèse temporelle derrière ce protocole, c'est que quand on écrit "émetteur positionne la donnée puis lève A", cet ordre temporel sera respecté à l'arrivée au récepteur. Si le système physique respecte cette hypothèse, on peut d'ailleurs aussi bien envoyer les données par paquets en parallèle, ce qui réduit le coût de l'asynchronisme. C'est comme cela que le processeur parle avec la mémoire dans votre PC (la mémoire peut avoir des tas de bonnes raisons de ne pas répondre tout de suite quand le processeur lui demande une donnée, on verra lesquelles).

Voyons maintenant un protocole vraiment insensible au délai. L'idée est d'utiliser un code dit double-rail pour la donnée : un bit est transmis sur deux canaux, avec la signification suivante : 1 est codé par 01, 0 par 10, la situation de repos est codé par 00. Ainsi mettre une donnée sur le double-rail change obligatoirement au moins un bit, ce qui est détecté par le récepteur, et tient donc lieu soit de R (dans le protocole piloté par l'émetteur) soit d'A (dans le protocole piloté par le récepteur).

Loi de conservation des emmerdements : il faut dans ce cas vraiment deux canaux pour transmettre un seul bit.

Tout se complique si on suppose des erreurs possibles sur la ligne. Solution : encore plus de protocole. Vous avez déjà vu ce qu'il faut mettre dedans.

7.3 Bus trois états

On a déja vu la porte 3 états pour construire des registres. Elle est en général utilisée pour partager un canal entre plusieurs émetteurs. Il faut alors un *arbitrage* qui garantit qu'à un instant donné un seul émetteur écrit sur le canal (sinon : situation logique indéterminée, qui se traduit souvent par une forte odeur de silicium fondu).

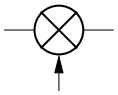


FIGURE 7.2 – Porte de transmission, ou porte "trois états"

Exemples :

— L'intérieur des circuits mémoires, pour lier la donnée sélectionnée par le décodeur d'adresse à la sortie : on sait qu'une seule donnée passera sur ce fil. Cette garantie est fournie par la construction du décodeur d'adresse qui sélectionne une seule des lignes d'adresse.
— Par extension, la construction d'une grosse mémoire à partir de petites.
— Le canal de données qui relie le processeur à la mémoire dans la machine de von Neumann. Il faut que les données puissent aller dans un sens ou dans l'autre, mais on n'a pas besoin qu'elles aillent dans les deux sens en même temps.

En général, un bus est un moyen de transport partagé... ici c'est un fil.

7.4 Réseaux en graphes (*)

En supposant réglée la question de transmettre des données sur un canal, on veut à présent relier entre eux n circuits (ou ordinateurs) par des canaux. Pour simplifier, disons que tous ces

canaux sont bidirectionnels et identiques (même débit etc). Le réseau a alors une structure de graphe, les canaux étant le arêtes et les circuits étant les nœuds.

7.4.1 Les topologies et leurs métriques

Considérons deux topologies possibles de réseaux pour relier n ordinateurs : l'anneau, représenté figure 7.3, et le tore bidimensionnel, représenté figure 7.5 qui est une extension de la grille bidimensionnelle (représentée figure 7.4) dans laquelle tous les nœuds ont le même nombre de canaux.

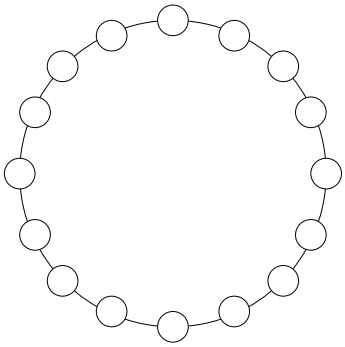


FIGURE 7.3 – Graphe anneau

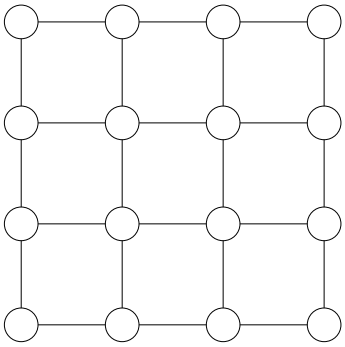


FIGURE 7.4 – Graphe grille 2D

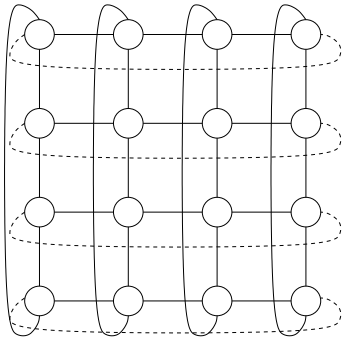


FIGURE 7.5 – Graphe tore 2D

On peut définir sur ces deux exemples les notions suivantes :

Degré Le degré d'un nœud est le nombre d'arêtes reliées à ce nœud. Par extension, le degré d'un graphe est le degré maximum des nœuds du graphe. Par exemple le degré est de 2 pour l'anneau, et de 4 pour la grille et le tore. A priori, plus le degré sera élevé, plus le nœud sera cher. Plus le degré est petit mieux c'est.

Diamètre Le diamètre d'un graphe est le maximum de la longueur du plus court chemin entre deux nœuds. Dans nos réseaux à $n = 16$ nœuds, le diamètre de l'anneau est 8 , le diamètre de la grille est 6, le diamètre du tore est 4. Le diamètre mesure la distance maximum à parcourir pour une information dans le réseau. Plus il est petit mieux c'est.

Bissection La bissection est une mesure de la quantité d'information qui peut passer à travers le réseau à un instant donné : Plus elle est élevée, mieux c'est. Techniquement, on appelle bissection-arêtes (resp. bissection-sommets) le nombre minimum de liens (resp. de sommets) dont la destruction entraîne la déconnexion en deux moitiés de même ordre (à un près).

Tore et anneau ont tous deux des degrés constants, mais le tore a une bissection et un diamètre en $\sqrt{n}$, ce qui semble mieux que l'anneau (bissection de 2 et diamètre en $n/2$. On peut définir un tore tridimensionnel et plus (et au fait l'anneau est un tore unidimensionnel). Mais d'autres topologies offrent des compromis encore plus intéressants :

L'hypercube de degré d a 2^d nœuds, un diamètre de d, un bissection de 2^{d-1}. Son seul défaut est un degré qui peut devenir relativement élevé.

C'est pourquoi on a inventé le Cube Connectant des Cercles ou CCC, représenté figure 7.7 en dimension 3. On place des anneaux de taille d à chaque sommet d'un hypercube de degré d, et chaque nœud d'un anneau possède en plus un canal selon une des dimensions de l'hypercube. Le degré est 3 quelle que soit la dimension, et le diamètre est passé de d à d^2 : c'est toujours logarithmique en le nombre de nœuds.

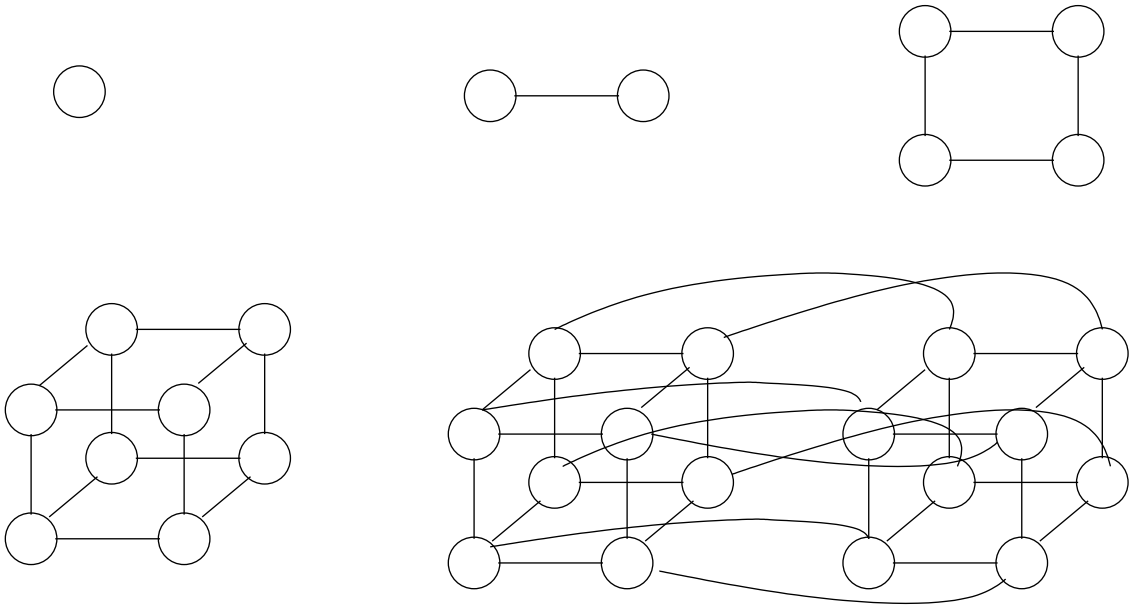


FIGURE 7.6 – Graphes hypercubes

7.4.2 Routage

Le routage c'est le choix du chemin que va suivre l'information dans le graphe.

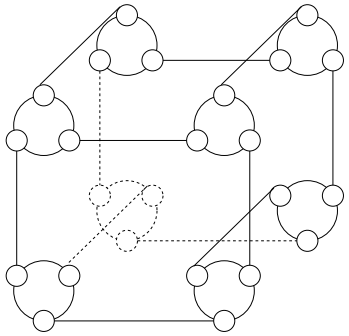


FIGURE 7.7 – Graphe *cube-connected cycles*

Dans un circuit intégré le routage est surtout statique : on a tiré des fils d'un bloc logique A à un autre bloc logique B, et une transmission d'info de A à B passera par ces fils [1].

Dans un grand réseau (téléphonique ou internet), par mesure d'économie, on n'a pas un fil par communication possible. On a tout de même plusieurs chemins possibles. Le routage d'une transmission d'information est calculé lorsqu'on en a besoin. On parle de routage dynamique. Bref, un routage dynamique, c'est trouver un chemin dans un graphe. Alors que les algos qui calculent le routage d'un circuit intégré peuvent se permettre de tourner des heures, pour un routage dynamique il faut un algo assez rapide.

C'est assez facile dès qu'on a un seul émetteur et un seul destinataire, on trouve alors toujours un algorithme glouton (exo : donnez le sur la grille, sur l'hypercube, sur le CCC).

Quand on a plein de paires (émetteur, destinataire) en même temps, le problème du routage devient difficile. Pourquoi ? Pourquoi c'est dur ?

— Il faut éviter que deux données se battent pour le même fil en même temps.
— Idéalement, chaque info va suivre le plus court chemin. Mais déjà, trouver le plus court chemin de A à B dans un graphe est coûteux.
— Quand le plus court chemin est impraticable (un lien est déjà occupé), il faut faire un détour. A force de détours on n'a plus un plus court chemin. Il faut alors montrer que l'on reste dans un facteur raisonnable du plus court chemin (ou du diamètre du graphe), sinon non seulement la communication est plus lente, mais en plus elle consomme encore plus de fils, ce qui aggrave le problème pour les suivants.

Dans ces domaines il est facile de bricoler des heuristiques qui ont l'air de marcher bien, mais il est très dur de montrer qu'elles offrent certaines garanties.

Les métriques du graphe deviennent importantes.

Il y a deux grandes familles de techniques de routage : la *commutation de ligne* (pensez au téléphone de l'URSS) et la *commutation de paquet* qui est ce qui se fait sur internet.

— Pour la commutation de ligne, il faut a priori un central qui supervise le réseau et réserve la ligne.
— Dans la commutation de paquet, un message est coupé en paquets de taille plus ou moins fixe, et ces derniers sont lancés dans le graphes et doivent y trouver de proche en proche leur chemin. Le gros avantage est que les algorithmes de routage deviennent décentralisés.

Remarque : on peut faire de la commutation de ligne virtuelle par dessus de la communication de paquets. Par exemple, on veut pouvoir réserver un canal virtuel de A à B, avec un certain débit garanti pour y faire passer de la video en temps réel.

Deux techniques de commutation de paquets : store and forward, et wormhole qui est un genre de réservation de ligne. Les anneaux du corps du vers s'appellent des flit. Vous avez intérêt

1. C'est vrai pour les petits circuits : un multiplieur, un processeur. De nos jours on est capable de concevoir des circuits intégrants des dizaines de processeurs, de mémoires et d'unités de calcul. Si l'on relie tous ces blocs par du routage statique, on constate que la plupart de ces fils sont inactifs la plupart du temps, c'est du gâchis de surface. Il y a donc beaucoup de recherche actuellement sur les *Networks on Chip*, ou NoC, dans lesquels les fils seront partagés et le routage dynamique.

à avoir noté mes explications au tableau.

Il faut éviter les situations d'interblocage : la communication de A vers B attend pour se terminer que la communication de C vers D libère un certain fil. Mais la communication de C vers D attend elle-même pour libérer ce fil que la communication de A vers B libère un autre fil. Montrer qu'un algorithme de routage ne mènera jamais à un interblocage est en général très difficile.

Dans IP (internet protocol), technique sommaire qui garantit l'absence d'interblocage, et aussi évite qu'un paquet tourne à l'infini dans le réseau sans jamais arriver : chaque paquet part avec une durée de vie, qui est décrémentée à chaque passage dans un noeud. Quand la durée de vie arrive à zéro, le paquet est détruit sans autre forme de procès. Eh, oh, et mes données ? Ben, tu les renverras... si le paquet qui te prévient de l'exécution de ton paquet t'arrive... Oui, les protocoles deviennent compliqués. On parlera des protocoles de l'internet quand on aura construit notre ordinateur.

En attendant, exemples de preuve que pas d'interblocage et pas de famine :

— anneau (protocole Token Ring). Par exemple, au fond de la PléStation3, il y a deux anneaux tournant en sens inverse.
— routage 2D Manhattan,
— routage hypercube.

7.4.3 Types de communication : point à point, diffusion, multicast

7.5 Exemples de topologies de réseau (*)

7.5.1 Le téléphone à Papa

Commutation de ligne, protocole très centralisé ("central téléphonique").

7.5.2 L'internet

Couche physique : ethernet.

Architecture en graphe sans vraiment de structure. Hiérarchie à deux niveaux : *local area network* ou LAN, *wide area network* ou WAN.

Couches logiques : commutation de paquets, routage par store and forward, protocole décentralisé.

7.5.3 FPGAs

Grille 2D, commutation de ligne par *crossbar*, routage statique.

7.5.4 Le bus hypertransport

Allez lire la page Wikipedia, c'est votre collègue Nicolas Brunie qui l'a écrite, je me demande s'il me le pardonnera un jour.

7.5.5 Machines parallèles

On ne dit plus machines parallèles, on dit Ze Grid. Pardon, je me reprends. On ne dit plus Ze Grid, on dit cloud computing. Bref. On assemble actuellement les coeurs sur la puce par un bus, les puces sur une carte par un autre bus, les cartes entre elles par des réseaux simples (la dernière fois que j'ai regardé c'était des anneaux), et les armoires ainsi obtenues par le l'Internet anarchique.

Pour les détails j'ai demandé à Loris Marchal qui a demandé à Jean-Yves l'Excellent : *La réponse plus complète, c'est que sur des machines parallèles du genre silicon graphics, il y a encore des topologies marrantes (hypercubes ? fat-tree,...). Sur des clusters, en général il y plutôt des crossbar quand c'est pas*

trop gros (par exemple : earth simulator : cross bar entre 640 noeuds) et quelque chose de hierarchique pour les trucs plus récents et plus gros. D'autre part, il y a parfois plusieurs types de réseaux disponibles. Pour Blue Gene, il y a 3 réseaux : un pour les communications collectives synchrones (barrières et reductions), un pour les communication point à point et asynchrone, et un pour le monitoring et le système. Chacun utilise une topologie (et une technologie) différente.

Si vous n'avez pas tout compris dans cette section c'est normal et c'est pas grave.

Deuxième partie

Machines universelles

Chapitre 8

Jeux d'instruction

8.1 Rappels

Une machine de von Neumann c'est un *processeur* relié à une *mémoire*.

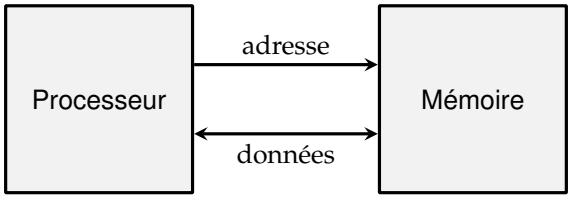


FIGURE 8.1 – Mon premier ordinateur

La mémoire est adressable par mot (de 8, 16, 32 ou 64 bits de nos jours), et grande (de nos jours, de l'ordre de 10^9, ou 2^{30} mots). La mémoire est non typée : chaque mot peut être interprété de multiples manières.

Le processeur exécute le **cycle de von Neumann**. Pour cela, il garde dans un coin une adresse qu'il appelle PC *(program counter)*. Le cycle, réalisé de nos jours dans les 10^9 fois par seconde, est le suivant :

— lire le contenu de la cellule à l'adresse PC
— l'interpréter comme une instruction, et l'exécuter
— Ajouter 1 au PC
— Recommencer

Une **instruction** est donc l'unité de travail que peut réaliser un processeur.

Définir un **jeu d'instruction** c'est définir la manière dont le processeur interprète n bits comme une instruction. C'est le travail du constructeur du processeur, et la courte histoire de l'informatique est pleine d'idées rigolotes dans ce domaine, pour n allant de 4 à 256.

Un bon jeu d'instruction est un jeu d'instruction

1. universel (ou Turing-complet, ou qui permet d'exprimer tout programme exprimable),
2. pour lequel il sera facile de construire le processeur qui l'exécute, et
3. avec lequel il sera facile d'écrire des programmes.

Le point 1 est facile à assurer, il suffit d'exhiber un programme qui simule la machine de Turing qui vous arrange. Par contre, il y a une Loi de Conservation des Emmerdements qui dit que les points 2 et 3 sont antagonistes.

8.2 Vocabulaire

Dans la suite on appelle les instructions du processeur **instruction machines**. Une instruction sera codée par un champ de bits. Dans les processeurs RISC *(reduced instruction set computers)*, toutes les instructions sont de la même taille, correspondant à un mot mémoire, ce qui simplifie leur décodage. Nous allons construire un tel processeur, inspiré de l'ARM.

Dans certains processeurs dits CISC *(complex instruction set computers)*, il peut y avoir des instructions de tailles (en bits) différentes. Le cycle de von Neumann est identique sur le fond. Nous allons observer un tel processeur, le Pentium de mon portable. *Je changerai de police pour évoquer les processeurs CISC, pour ne pas que leur complexité ne pollue trop mon propos.*

Pour les besoins de l'interface avec les humains, chaque instruction (jusque là le contenu d'une case mémoire, donc un vecteur de bits) aura également une représentation textuelle non ambiguë, que l'on appelle **mnémonique**. L'ensemble des mnémoniques forme le **langage assembleur**. Le programme qui convertit le langage assembleur en séquence d'instructions binaires s'appelle aussi un **assembleur**.

8.3 Travaux pratiques

Les TP MSP430 sont là pour vous montrer tout cela pour un processeur RISC moderne (bien que petit). Voici deux autres exemples.

8.3.1 Le jeu d'instruction de votre PC

Tout ce qui suit est à essayer sur un PC sous Linux[1]. Prenez un programme C quelconque `toto.c`, et compilez-le avec `gcc -S`. Voici un exemple de `toto.c` :

```
main()  {
  int i=17;   /* une constante facile à retrouver dans l'assembleur */
  i=i+42;     /* une autre */
  printf("%d\n", i);
}
```

Vous obtiendrez un fichier `toto.s` en langage assembleur (il ne contient que des mnémoniques). Ici il s'agit de l'assembleur du Pentium. Voici les extraits de ce fichier qui sont lisibles à ce stade du cours.

```
       (... de la paperasse administrative ...)
movl $17, -4(%rbp)
addl $42, -4(%rbp)
       (... des instructions pour passer les paramètres à printf )
call printf
       (... encore de la paperasse )
```

La première ligne met *(move)* la constante 17, considérée comme un entier long (suffixe `l`), quelquepart en mémoire. La seconde ligne ajoute 42 à la même case mémoire.

Il existe en fait un assembleur dans la suite `gcc` : vous pouvez donc vous amuser à modifier le fichier `toto.s` puis le transformer en un exécutable (un fichier dans lequel chaque instruction est codée en binaire) par `gcc toto.s`.

Pour obtenir le binaire, lancez la commande `objdump -d a.out` (le `-d` signifie *disassemble*, donc faire le travail inverse de l'assembleur). Vous obtenez des mnémoniques, et également le code binaire (enfin, hexadécimal) de chaque instruction.

Voici les lignes correspondant au code assembleur précédent :

1. Par la magie du Net on peut aussi désormais utiliser `http://gcc.godbolt.org/` qui est plus simple mais va moins loin.

```
 Adresses  Instructions binaires    Assembleur
 (...)
  40052c: 55                        push   %rbp
  40052d: 48 89 e5                  mov    %rsp,%rbp
  400530: 48 83 ec 10               sub    $0x10,%rsp
  400534: c7 45 fc 11 00 00 00      movl   $0x11,-0x4(%rbp)
  40053b: 83 45 fc 2a               addl   $0x2a,-0x4(%rbp)
  40053f: 8b 45 fc                  mov    -0x4(%rbp),%eax
  400542: 89 c6                     mov    %eax,%esi
 % (...)
```

On a des instructions de toutes les tailles. C'est vraiment n'importe quoi ce processeur, ils ne vont pas en vendre beaucoup.

8.3.2 Le jeu d'instruction de votre téléphone portable

Pour le même programme, en utilisant des outils de *cross-compilation* vers ARM (par exemple le paquet `gcc-arm-linux-gnueabihf` sous Debian/Ubuntu). On tape à présent `arm-linux-gnueabihf-gcc -marm -S toto.c`. Vous obtiendrez un fichier `toto.s` contenant entre autres les mnémoniques suivantes :

```
       (... de la paperasse administrative ...)
mov r3, #17
add r3, r3, #42
       (... des instructions pour passer les paramètres à printf )
bl printf
       (... encore de la paperasse )
```

Et voici les résultat de
`arm-linux-gnueabihf-gcc -marm toto.c;`
`arm-linux-gnueabihf-objdump -d a.out`

```
000083fc <main>:
    83fc: e92d4800  push {fp, lr}
    8400: e28db004  add fp, sp, #4
    8404: e24dd008  sub sp, sp, #8
    8408: e3a03011  mov r3, #17
    840c: e50b3008  str r3, [fp, #-8]
    8410: e51b3008  ldr r3, [fp, #-8]
    8414: e283302a  add r3, r3, #42 ; 0x2a
    8418: e50b3008  str r3, [fp, #-8]
    841c: e3080488  movw r0, #33928 ; 0x8488
    8420: e3400000  movt r0, #0
    8424: e51b1008  ldr r1, [fp, #-8]
    8428: ebffffae  bl 82e8 <_init+0x20>
    842c: e1a00003  mov r0, r3
    8430: e24bd004  sub sp, fp, #4
    8434: e8bd8800  pop {fp, pc}
```

Vous observez que chaque instruction est encodée sur 32 bits tout pile. C'est quand même plus régulier : ce sera plus simple à construire et cela consommera moins.

Bonus : ARM contient en fait deux jeux d'instructions, celui-ci, et un autre dans lequel les instructions les plus simples sont encodées sur 16 bits seulement. Avancez dans le poly, et lorsque vous rencontrerez le mot "thumb", revenir à cette section et recommencer en remplaçant `-marm` par `-mthumb`. Vous obtiendrez quelquechose qui ressemble plus à du code MSP430.

8.4 Instruction set architecture

La définition du jeu d'instructions est décisive : ce sera l'interface entre le hard et le soft.

— Le programmeur (ou le compilateur) ne connaît que ces instructions machines et leur sémantique bien documentée dans de gros manuels de référence. Il ne sait pas comment elles sont implémentées, et s'en fiche pas mal. D'ailleurs cela change tout le temps.
— Le concepteur du pentium 12 doit contruire un processeur qui implémente ce jeu d'instruction et sa sémantique. À part cela, il a toute liberté.

On parle souvent d'ISA pour *instruction-set architecture*. Des exemples d'ISA sont IA32 (implémenté par les processeurs Pentium, Athlon, etc), IA64 (implémenté par les processeurs Itanium), Power (implémenté par les processeurs Power très chers, et les processeurs PowerPC meilleur marché), ARM, MSP430, ...

L'exemple d'IA32 montre combien l'ISA est déconnecté du processeur l'implémentant : il n'y a pas un transistor de commun entre les processeurs d'Intel et leurs concurrents d'AMD, et pourtant ils peuvent exécuter les mêmes programmes. On discutera plus bas la relation entre ISA et architecture.

En pratique, les ISA évoluent, et les grandes familles évoquées ci-dessus se déclinent en versions, ajoutant des instructions au jeu d'instructions de base. Exemples (plus ou moins connus grâce à un marketting plus ou moins efficace) : les extension multimedia à IA32, SSE/SSE2/SSE3/... chez Intel [2], 3DNow chez AMD, Thumb, Jazelle, NEON et VFP chez ARM, ...

Il est important de savoir bien distinguer les noms marketing des ISA de ceux des processeurs. Par exemple, ARMv7 est un ISA, alors que ARM7 est un processeur (qui n'implémente pas du tout ARMv7). Allez bouquiner sur Wikipedia les pages sur ARM et IA32, et faites cet exercice.

8.5 Que définit l'ISA

Un programme typique c'est une boucle qui va chercher des opérandes en mémoire, calcule dessus, puis les range en mémoire. On va définir des instructions pour tout cela.

8.5.1 Types de données natifs

Commençons par les opérations de calcul. L'ISA définit avant tout un certain nombre de **types de données** sur lesquels elle aura des opérateurs de calcul matériels. Les types typiques sont

— des adresses mémoire.
— des champs de bits, sur lequel on peut faire des ET, des OU, des XOR, des décalages...
— des entiers de différentes tailles (8, 16, 32, 64 bits pour l'IA32 ; 8 et 32 pour l'ARM ; 32 pour l'Alpha, etc). Il y a deux sous-types, les entiers signés et non signés. Grâce au complément à 2, ils se distinguent uniquement par la gestion des débordements et des décalages.
— des nombres en virgule flottante : simple précision (32 bits) et double précision (64 bits). Ces formats sont imposés par un standard, appelé poétiquement IEEE 754, dont la version courante date de 2008. Ce standard définit également les précisions *quadruple* (128 bits) et *half* (16 bits). *IA32 et IA64 ajoutent des formats à 80 bits, IA64 ajoute un format à 82 bits...*

Les extensions multimedia, introduites dans les années 90, définissent des vecteurs de petits entiers (MMX) et des vecteurs de petits flottants (3DNow, SSEx, AVXy). Pour l'ISA ARM, l'équivalent s'appelle NEON.

2. En 2007, à 15 jours d'intervalle, Intel a annoncé l'extension SSE4 qui introduit par exemple une instruction calculant a*b+c*d en flottant avec 3 arrondis, puis AMD a annoncé l'extension SSE5 ajoutant une instruction FMA : a*b+c avec un seul arrondi. Je ne sais pas comment ils se sont partagé ces effets de manche, mais j'ai parié trois polys que les processeurs qui sortiraient 5 ans plus tard auraient tous les deux extensions. Allez voir si j'ai gagné sur `http://en.wikipedia.org/wiki/FMA_instruction_set`, parce que l'histoire est rigolote.

8.5.2 Instructions de calcul

L'ISA définit ensuite des **instructions de calcul** pour ces différents types de données : addition, multiplication, décalage, OU logique... Nous allons filer l'exemple de l'addition, pour les autres c'est très similaire.

Il faut, dans une instruction d'addition, pouvoir spécifier ce qu'on additionne (les opérandes) et où on le range (la destination). Idéalement, on aimerait donner trois adresses dans la mémoire. Une instruction serait alors "ajouter le contenu de la case mémoire numéro X au contenu de la case mémoire numéro Y et ranger le résultat dans la case numéro Z, les trois contenus étant interprétés comme des entiers 32 bits". Pour un processeur moderne qui peut adresser une mémoire de 2^{32} mots, l'information totale des trois opérandes nécessiterait déjà $3 \times 32 = 96$ bits. Si je veux coder chaque instruction dans un mot de 32 bits c'est mal parti.

C'est pourquoi les ISA définissent tous une mémoire de travail plus petite, interne au processeur, que l'on appelle les **registres**. Chez les RISC, ils sont tous identiques et forment une petite mémoire adressable interne. Si l'on a 16 registres et que les opérations de calcul ne peuvent travailler que sur le contenu de ces registres (cas de l'ARM), les trois opérandes d'une instruction peuvent se coder sur $3 \times 4 = 12$ bits seulement. Un avantage supplémentaire est que ces registres seront lus et écrits plus vite que la mémoire principale, puisqu'ils sont intégrés sur la puce du processeur.

Chez les processeurs CISC, les registres ont de petits noms et des fonctions différentes (accumulateur, registre d'index, ... nous ne détaillerons pas).

Une autre solution est de ne travailler que sur une seule case mémoire à la fois, comme dans l'addition de `toto.c` *ci-dessus. Nous l'expliquons plus en détail plus bas. Mentionnons toutefois qu'elle n'est utilisée que par des ISA datant de la préhistoire de la paléoinformatique, avant que les vitesses de la mémoire centrale et du processeur ne fassent le grand écart,*

Nous pouvons à présent définir une instruction d'addition dans un processeur RISC. Son mnémonique sera par exemple `add R0, R1 -> R3`. La forme générale pour une opération binaire sera `op Rx, Ry -> Rd`, où `op` peut être `add`, `sub`, `mul`, `xor`, ... et `d`, `x` et `y` sont les numéros des registres destination et sources. Le codage de cette instruction sera très simple : un certain nombre de bits pour coder `op`, (mettons les 8 premiers, ce qui nous permet 256 instructions différentes), un certain nombre pour coder `d`, un certain nombre pour coder `x` et `y`. Il sera d'autant plus simple de construire l'architecture que ce code est simple : on mettra le code du registre destination toujours au même endroit, et idem pour les deux opérandes.

Remarquez que les "vrais" mnémoniques sont le plus souvent écrits `op Rd, Rx, Ry` – et parfois la convention est de mettre la destination en dernier (voir `toto.s`), et on trouve même les deux conventions qui cohabitent pour le même processeur suivant les systèmes d'exploitation ! La raison en est surtout historique : les premiers mnémoniques reprenaient les champs de bits dans l'ordre dans lequel ils étaient dans l'instruction, ce qui facilitait l'assemblage – je parle d'une époque où le programme assembleur était lui-même écrit en assembleur sur des cartes perforées. Notre convention est plus lisible, et comme le mnémonique n'est qu'une représentation textuelle on la garde : il faut se convaincre que, si le choix de l'ISA lui-même fait la différence entre un bon processeur et un mauvais, le choix de ses mnémoniques, par contre, n'est qu'une convention et n'a pas grande importance.

Nous avons défini une addition dite **à trois opérandes** puisque les trois opérandes sont exprimés. Il y a d'autres solutions :

- Instructions **à 2 opérandes** : `op Rd, Rx -> Rd`. On écrase l'un des opérandes.
- Intructions à **1 opérande** : idem, mais `Rd` est fixé et implicite. On l'appelle en général l'accumulateur, en mémoire aux temps héroïques où les processeurs n'avaient même pas de multiplieurs, et où l'on accumulait des séquences d'additions.
- Instructions **à pile** (0 opérande) : le processeur dispose d'une pile (comme les calculettes HP), et une instruction `add` prend ses deux opérandes au sommet de la pile et y range le résultat.

Il y a là un compromis :

— l'instruction à trois opérandes est la plus puissante, et donc la plus compacte en termes de nombres d'instructions. Elle est préférée pour les processeurs RISC récents : leur unité d'information fait 32 bits, et on se donne donc 32 bits pour chaque instruction, ce qui permet d'y coder trois opérandes voire plus (voir ci-dessous)
— Avec des instructions à 0, une ou deux opérandes, il faut souvent faire des copies des registres écrasés (c'est une autre instruction), ou bien des `swap` et des `dup` sur la pile : un programme donné a besoin de plus d'instructions. Par contre, en terme de nombre total de bits, le programme est plus compact.
Le plus compact est le jeu d'instruction à pile, puisqu'il n'y a que l'instruction à coder, pas ses opérandes. C'est la raison pour laquelle ce type d'instruction est souvent choisi pour le *bytecode* des machines virtuelles pour les langages comme Caml ou Java.

Il faut noter que dans tous les ISA, l'un des registres peut souvent être remplacé par une constante. Le mnémonique est alors par exemple `add R12, 4 -> R1`. Techniquement, c'est une instruction différente, à laquelle on peut donner un mnémonique différent, par exemple `addi` pour *add immédiate constant*. On peut aussi préferer laisser le mnémonique `add`, puisqu'il n'y a pas d'ambiguïté.

Il est raisonnable de coder la constante en place du numéro de registre qu'elle remplace, ce qui signifie que si on n'a que 16 registres on ne peut coder que les constantes de 0 à 15. C'est déjà pas mal. Il y a plein d'astuces pour récuperer des bits de plus pour ces constantes immédiates.

Un cas particulier est l'instruction `move Rx -> Rd` qui réalise une copie. On aimerait que sa version constante, que nous appelerons `let`, puisse mettre une constante arbitraire de 32 bits dans un registre, mais ceci nécessiterait une instruction de plus de 32 bits... Définissons donc

```
LetHigh 123 -> R4
LetLow 456 -> R4
```

qui placent la constante (16 bits) respectivement dans les moitiés hautes et basses de `Rd`. Cela permettra de charger un registre avec une constante arbitraire en deux instructions de 32 bits. Une option CISC (voir `toto.s`) est d'avoir une instruction de chargement de constante 32 bits qui fait plus de 32 bits.

Dans `toto.s`, l'addition est une opération à deux opérandes, dont une constante et une en mémoire (sur laquelle nous reviendrons).

8.5.3 Instructions d'accès mémoire

On a aussi besoin d'instructions pour l'accès à la mémoire.

Lecture et écriture basique

Par exemple, une instruction qui demande à la mémoire le contenu de la case d'adresse le nombre contenu dans R2, et le place dans R5, s'écrira

```
Read [R2] -> R5
```

et l'instruction d'écriture s'écrira

```
Write R3 -> [R6]
```

Adressage indirect

Dans une machine 3 opérandes, on n'utilise pas le troisième champ opérande. Comment l'utiliser intelligemment ? En inventant des *modes d'adressages* plus sophistiqués. Par exemple, si `A` est un tableau stocké en mémoire à partir de l'adresse a, l'accès à l'élément `A[i]` de ce tableau demande d'ajouter a et i pour obtenir l'adresse de la case. Il est naturel de proposer les instructions

```
Read [R2+R3] -> R5
Write R3 -> [R6+R7]
```

Remarquez que dans le write, on n'a aucun registre de destination, les trois registres sont lus.

Exemple d'accès à un tableau dans ARM : `LD R1, [R3, R5 LSL#4]`
(si les cases du tableau sont de taille 16) et avec post-incrément : `LD R1, [R3, R5 LSL#4]!`
(c'est pour parcourir un tableau en gardant un pointeur sur la case courante)

Instructions de gestion de pile

Les langages de haut niveau utilisent beaucoup des piles *(last in, first out)*. Par exemple, c'est comme cela que l'on gère les appels de procédures (`call` et `return` que l'on va voir bientôt).

— Une pile est stockée en mémoire.
— Un registre sert de pointeur de pile (on l'appelle alors SP pour *stack pointer*).
— L'instruction `Push R1` va empiler la valeur de R1 sur la pile, c'est-à-dire réaliser la séquence d'actions `Write R3 -> [SP]` puis `SP+1 -> SP` (le tout en une instruction)
— L'instruction `Pop R3` va faire le contraire, c'est-à-dire `SP-1 -> SP` puis `[SP]->R3`

La description précédente nous donne une pile montante (en mémoire) avec un pointeur vers la prochaine case vide. En fait il y a 4 variantes de pile :

— Si vous échangez `SP-1` et `SP+1` ci-dessus, vous obtenez une pile descendante : quand on empile, on descend dans la mémoire.
— Si vous échangez les deux actions de chaque séquence (par exemple `Push` devient `SP+1 -> SP` puis `Write R3 -> [SP]`), vous obtenez une pile dont le pointeur pointe sur le dernier élément empilé.

Certains processeurs imposent l'une de ces 4 variantes (exemple : IA32), d'autres vous laissent choisir. Par exemple sur ARM n'importe quel registre peut servir de pointeur de pile, et le push et le pop sont des variantes des instructions mémoire classiques. Par exemple `LD R1, [R3, #4]!` peut servir de pop sur une pile descendante qui pointe sur pile pleine, et dont le pointeur de pile est R3.

8.5.4 Instructions de contrôle de flot

Pour le moment notre processeur est capable d'exécuter un cycle de von Neumann sur carton perforé, mais pas de sauter des instructions ni de répéter en boucle un programme. Il reste à ajouter des instructions de *branchement*.

Sauts inconditionnels, relatifs et absolus

La version minimaliste offre deux instructions :
`GoForward 123` qui avance de 123 instructions
`GoBackward 123` qui recule de 123 instructions.

Ces deux instructions sont en pratique des additions/soustractions sur le PC. Cela s'appelle un saut ou branchement *relatif* (relatif au PC). La constante ajoutée/soustraite occupe tous les 24 bits restant.

On ne peut donc pas sauter à plus de 2^{24} cases de la position actuelle du PC. Si vous y tenez, le processeur vous offrira en général une instruction permettant de sauter à un emplacement arbitraire (saut ou branchement *absolu*). A nouveau, soit il faut plus de 32 bits pour coder une telle instruction, soit il faut bricoler. On peut simplement mettre des adresses de saut dans des registres, et le saut absolu est alors un `MOV Rx -> PC` (il nécessite donc plus d'une instruction, pour remplir Rx avec l'adresse voulue avant le saut). Il y a aussi des mécanismes plus spécifiques. Par exemple, on a vu des sauts absolus indirects, motivés par le besoin d'avoir des appels systèmes partagés : l'ISA décrète que les 2^{10} premières cases mémoires contiennent des adresses de saut (que vous remplirez avec les adresses des fonctions système de base comme `malloc` ou `printf`), et l'instruction de saut absolu n'a besoin que de 10 bits pour coder une telle adresse. Par contre, un saut absolu nécessite une lecture mémoire supplémentaire. Il y a toujours une LCDE sur les sauts absolus.

Sauts conditionnels – drapeaux

Pour revenir à une instruction de saut minimale, il faut tout de même pouvoir implémenter des boucles `for` et `while`. Les instructions de saut existent en version *conditionnelle*, par exemple : `GoForward 123 IfPositive`
qui avance de 123 instructions si le résultat de l'instruction précédent cette instruction était positif. On aura un certain nombre de telles conditions (si le résultat était nul, non nul, s'il y a eu un débordement de capacité). Pour ces branchements conditionnels, la distance de saut occupe moins de bits, car il faut des bits pour coder la condition.

Le processeur doit donc garder en mémoire l'information correspondante (résultat positif, résultat nul, ...) d'une instruction à l'autre. Cette information est compacte (peu de bits) et est stockée dans un registre de *drapeaux*.

C'était le minimum syndical : des drapeaux, et une instruction de saut conditionnel. Mais plus la technologie évolue plus un saut conditionnel devient coûteux, essentiellement parce qu'il introduit de l'incertitude qui empêche le pipeline – on verra plus tard. Voyons donc quelques variations.

Une idée récente, rendue possible par le passage aux instructions 32 bits, est de permettre que *toutes* les instructions soient conditionnelles : les conditions sont codées dans un champ supplémentaire du mot d'instruction, partagé par toutes les instructions. C'est le cas dans l'ARM et IA64.

Dans IA64, de plus, il n'y a pas un seul jeu de drapeaux, mais des *registre de prédicat* : chaque instruction de test peut choisir dans quel registre de prédicat elle stocke le résultat du test, et chaque instruction peut être *prédiquée* (conditionnée) par un registre de prédicat.

Dans la variante récente dite Thumb2 du jeu d'instruction ARM, il y a une instruction IT pour If-Then qui rend conditionnelles les instructions suivantes (jusqu'à 4 : l'instruction suivante est conditionnelle, et on encode dans le IT si les 3 suivantes, et si elles font partie du *then* ou du *else*). Au final cela permet de rendre toutes les instructions conditionnelles pour des instructions encodées sur 16 bits.

Appel de sous-routine

Enfin, il y a un certain nombre d'instructions qui servent à supporter les langages de haut niveau, en particulier un `call` et un `return` pour les appels de procédure. Sur le fond, `call printf` est un saut absolu à l'adresse de `printf`. Toutefois, avant de sauter, il doit sauvegarder quelquepart l'adresse de l'instruction qui suit le `call`. Ainsi, l'instruction `return` (à la fin du code de `printf`) peut copier cette adresse sauvegardée dans le PC, ce qui reprend l'exécution juste après le `call`.

Le plus souvent, l'adresse de retour (l'adresse de l'instruction qui suit le `call`) est empilée sur une pile, et `return` est essentiellement un `pop PC`. On va voir des variantes un peu plus loin.

On parle d'appel de sous-routine et pas encore d'appel de procédure, parce qu'une procédure sous-entend un certain nombre de paramètres avec leur types, etc. Notre `call` fait juste un saut, sans passer de paramètre. Et `return` correspond bien à "`return;`" en C, mais on ne sait pas encore comment faire "`return value;`". La manière dont le passage de paramètre est implémenté sera vue en détail au chapitre 11.

8.5.5 Les interruptions et l'atomicité

Une interruption c'est quand le Reste Du Monde interrompt le processeur. Exemples :
— Il est arrivé un paquet sur la carte réseau, et il faut s'en occuper.
— La carte son signale qu'elle n'a bientôt plus d'échantillons à jouer.
— Un fâcheux a appuyé sur une touche du clavier. Pas moyen d'être tranquille.
— Etc.

Une interruption est un événement qui peut être provoqué soit par une cause externe (il y a des broches pour cela sur la puce du processeur), soit par une cause interne (division par zéro, accès mémoire interdit, on en verra plein d'autres).

Lorsque survient un tel événement, le processeur interrompt (temporairement) ce qu'il était en train de faire (son programme) pour sauter à la routine de traitement des interruptions. Typiquement, c'est un mécanisme de call/return.

Ce call, si vous avez suivi, fait deux choses en une instructions : 1/ sauvegarder l'adresse de retour et 2/ faire le saut. Il est important que ces deux actions soient fait en une seule instruction *(atomique)* pour ne pas permettre qu'une interruption intervienne entre les deux. Plus précisément, ce qui doit être atomique est l'empilement de l'adresse de retour *(push)* : si on le fait en deux instructions (par exemple écrire en mémoire puis incrémenter le pointeur de pile), une interruption entre les deux va provoquer un second *push*, donc deux écritures à la même case mémoire, la première étant perdue.

Voici en détail deux mécanismes qui existent sur de vrais processeurs :

— `call` atomique : fait un empilement puis un saut de manière atomique. C'est la manière traditionnelle de faire. Une interruption déclenche également un `call` vers une adresse prédéterminée. Vous pouvez vérifier que tout se passe bien.

— les designers de l'ARM ont trouvé que le mécanisme de call était trop complexe. Plus exactement il a trois actions à réaliser (une sur SP, une sur la mémoire, et une sur PC) : c'est trop. L'instruction ARM qui remplace le `call` est `BL` pour *branch and link* qui copie le PC dans R14 puis exécute le saut. C'est le destinataire du saut qui est chargée de sauvegarder R14 sur la pile (et parfois on peut l'économiser).
Mais alors, une interruption entre le BL et la sauvegarde de R14 ferait qu'on perdrait R14. Pour cette raison, une interruption sur ARM, utilise un R14 et un SP qui sont spécifiques aux interruptions. On parle de changement de mode du processeur.
Pour des raisons similaires, ARM a des instructions de sauvegarde atomique de tous les registres en mémoire.

8.5.6 Autres instructions

Instructions de changement de mode On a vu un mode interruptions, il y a aussi au moins *utilisateur* et *superviseur* (système). Les instructions craignos, comme celles qui servent à initialiser la protection de la mémoire, sont accessibles uniquement en mode superviseur. Certains processeurs ont toute une hiérarchie de modes intermédiaires.

Par contre, les interruptions ont une hiérarchie de priorité parallèles : on peut choisir qu'un signal extérieur interrompera le processeur s'il est en mode utilisateur mais pas s'il est en mode superviseur (elle sera alors mises dans une file d'attente). Ainsi, on peut assurer par exemple que le traitement d'une interruption n'est pas interrompu par l'interruption suivante, ou encore que le processeur, lorsqu'il doit exécuter un morceau de code dont le minutage est critique, ne sera pas interrompu.

Instructions de synchronisation Voici une autre classe d'instructions qui doivent être atomiques, et donc non interruptibles, et donc justifient des instructions spécifiques. Pour pouvoir programmer un processeur multicœur, il faut pouvoir implémenter des verrous qui permettent de synchroniser des cœurs. Les primitives de verrouillage doivent faire, de manière atomique, une lecture mémoire, un test sur la valeur lue, et éventuellement une écriture.

Instructions d'entrées/sorties Cela a existé, mais on le voit de moins en moins. Il est plus simple de considérer les entrées/sorties comme des cases mémoires spéciales *(memory-mapped IO)*.

8.5.7 Quelques ISA

Voici pour finir quelques exemples représentatifs d'ISA, par ordre croissant de performance... et de consommation électrique.

— **MSP430** est un jeu d'instruction RISC 16 bits à 16 registres qu'on retrouve dans les applications très faible consommation (cartes à puces, tags RFID). Il y a des instructions à un et deux opérandes. Si ces opérandes sont des registres, l'instruction tient en un mot de 16 bits. Les opérandes peuvent aussi être une case mémoire, dont l'adresse (sur 16 bits) est codée dans un des mots suivant l'instruction. Ou des constantes immédiates sur 16 bits, idem. Dans ce cas les instructions font 32 ou 48 bits. Certains registres sont spéciaux : le PC est R0, les drapeaux sont dans le registre R1, et le registre R2 peut prendre différentes valeurs constantes utiles (0, 1, -1...). *Un mot de contexte : c'est un jeu d'instruction pour microcontrôleur très basse consommation. Un microcontrôleur embarque processeur, mémoire, quelques périphériques (timers, etc) et interfaces d'entrée/sortie dans la même puce. L'objectif de faible consommation limite la fréquence, tandis que l'objectif d'une puce tout-en-un limite la capacité mémoire. D'où le choix d'une architecture 16 bits seulement, et le choix d'avoir des instructions qui calculent directement sur la mémoire. Le choix de fréquence basse permet en effet de faire une lecture mémoire et un calcul par cycle. D'ailleurs, le nombre de cycles que prend chaque instruction est exactement égal au nombres d'accès mémoire qu'elle fait.*

— **ARM** est un jeu d'instruction RISC 32 bits à 16 registres qu'on trouve entre autre dans tous les téléphones portables. Toutes les instructions sont codées dans exactement un mot mémoire (32 bits). Les instructions sont à 4 opérandes : la quatrième est un décalage qui peut s'appliquer au second opérande. Le second opérande, et le décalage peuvent être des constantes. Par exemple, `add R1, R0, R0, LSL #4` calcule dans `R1` la multiplication de `R0` par 17, sans utiliser le multiplieur (lent, et parfois d'ailleurs absent). Il y a d'autres applications plus utiles, comme la manipulation d'octets, et la construction de grandes constantes.
Remarque : Pour le marché de l'embarqué, ARM a ensuite introduit un mode du processeur qui utilise un second ISA, dit *thumb* (comme Tom Pouce) d'économie de mémoire. Toutes les instructions y ont 2 opérandes seulement et tiennent sur 16 bits. Le processeur expanse en interne chaque instruction en l'instruction 32 bits correspondante. Cette expansion n'est pas coûteuse, par contre le même programme, s'il fait moins d'octets au final, demande plus de cycles. Ce mode est utilisé typiquement pour les parties non critiques de l'OS d'un téléphone portable.
Enfin, tous les processeurs ARM récents permettent un mode Thumb-2 dans lequel des instructions de 16 et 32 bits peuvent être mélangées. L'option `-mthumb` de gcc permet de produire du code Thumb-2.

— **Power** est un autre jeu d'instruction 32 bits qu'on trouve de la Xbox aux serveurs de calculs d'IBM. Les instructions ont également 4 opérandes : l'opération arithmétique de base est le *fused multiply and add* ou FMA, qui fait `Rd := Rx × Ry + Rz`, laquelle permet même de faire des additions et des multiplications.

— **SPARC** est ISA 32 bits à trois opérandes. Son originalité est d'implémenter à partir de SPARC V2 une *fenêtre glissante* de registres sur un ensemble de registres plus grands. Une instruction spéciale fait glisser la fenêtre. Ainsi on a peu de registres visibles à un instant donné (donc un mot d'intruction qui reste petit), mais beaucoup de registres architecturaux. Cela a plein d'avantages, comme lors du passage des paramètres à une fonction. Cela a enfin des avantages pour implémenter un pipeline logiciel. Cette idée est reprise par l'ISA IA64.

— IA32 est une usine à gaz. Il possède plusieurs couches de registres ajoutées par les évolutions de l'ISA, et la taille de ces registre a grandi aussi. Les instructions sont de tailles variables, comme on a vu. Il peut réaliser des opérations arithmétiques dont un opérande est directement en mémoire (voir notre `toto.s`), ce que ne font plus les RISC à cause de l'écart de performance entre accès à un registre et accès à la mémoire.

— J'en présenterai deux autres en 9.6

8.6 Codage des instructions

Même avec de l'imagination, cela nous fait moins de 256 instructions différentes [3] : l'instruction elle-même sera codées sur 8 bits du mot d'instruction. Dans les 24 bits restant, on codera selon le cas trois registres, deux registres et une petite constante, deux registres (accès mémoire), un registre et une petite constante (décalage de bits), un registre et une grande constante.

On s'attachera à ce que le jeu d'instruction soit le plus *orthogonal* possible : les registres seront toujours codés au même endroit, ce qui facilitera la construction du circuit.

On constate que chaque instruction laisse plein de bits inutilisés. C'est du gâchis, et si on voulait construire un processeur compétitif, on coderait dedans des options supplémentaires qui rendent l'instruction plus puissante. Pour ceux que cela intéresse, cherchez "ARM assembly language" pour avoir une idée de telles options.

8.7 Adéquation ISA-architecture physique

En principe, le jeu d'instruction représente une bonne abstraction de la mécanique sous-jacente du processeur. Ainsi le programmeur a accès à toutes les ressources du processeur, et seulement à elles.

Par exemple, lorsque l'ISA offre une opération de division, cela veut en général dire que le processeur dispose d'un diviseur. Lorsque les opérandes d'une opération peuvent être des registres numérotés de R0 à R15, cela signifie en principe qu'il y a une boîte à registres (une petite mémoire adressable) à 16 entrées dans l'architecture.

Toutefois, cette belle harmonie, qui était de règle à l'époque héroïque des processeurs 8 bits, et qui est de règle pour les ISA récents, souffre trois grosses exceptions.

La première est le fait de jeux d'intruction historiques, tels l'IA32. Par exemple, cet ISA fut à l'origine défini avec 8 registres flottants, ce qui était à l'époque un bon compromis. De nos jours, 8 registres flottants ne sont plus suffisants pour la gestion efficace d'une unité *superscalaire* et *pipelinée*, c'est à dire capable de lancer à chaque cycle plusieurs (2 à 4) instructions flottantes dont le résultat n'arrivera que plusieurs cycles (3 à 5) plus tard. L'architecture physique comporte donc beaucoup plus de registres, et le processeur fait tout un travail de *renommage* des registres ISA en registres physiques. Ainsi, l'architecture physique ne correspond plus à l'ISA. Ceci a un surcoût matériel certain, mais aussi quelques avantages. Par exemple, un jeu d'instruction à 8 registres est plus compact qu'un jeu d'instruction à 128 (cas de l'IA64) : les programmes prendront moins de mémoire. Donc il y a un courant de pensée qui préconise des ISA ne correspondant pas au matériel, tant qu'on sait implémenter leur sémantique efficacement en matériel.

Le second cas de non-correspondance entre l'ISA et l'architecture date d'une époque où l'on programmait plus en assembleur qu'aujourd'hui. Il paraissait judicieux de mettre dans l'ISA des opérations rares et complexes, comme par exemple les fonctions élémentaires sinus, exponentielle, ou des opérations de traitement sur des chaînes de caractères, ou des modes d'adressages indirects réalisant en une instructions plusieurs accès mémoire... Ce sont des instructions IA32, qui au final sont exécutées par un petit programme stocké dans le processeur lui-même. On appelle cela la *microprogrammation*. C'est une caractéristique des ISA CISC, et pour le coup, tout le monde pense que c'était une mauvaise idée. En effet, par rapport à la même fonctionnalité implémentée en logiciel utilisant des instructions simples, cela rend le processeur

— plus coûteux (il faut stocker les microprogrammes),
— plus complexe (il faut un mini cycle de von Neumann à l'intérieur du grand),
— plus risqué (un bug dans le microcode nécessite de changer le processeur, alors qu'un bug dans du logiciel se corrige facilement)
— moins flexible : on ne peut pas adapter ces routines à un contexte donné. Par exemple, on aime avoir plusieurs versions d'une fonction élémentaire : l'une très rapide mais peu précise, une autre très précise mais plus lente, une troisième orientée débit, etc.

3. Il y en a plusieurs centaines dans IA32...

Petite histoire : en 1990, quelques années après que les intructions exponentielle et logarithme aient été microcodées pour le processeur 80386/80387 (ancêtre du pentium, en 1985), les PC (comme maintenant) étaient livrés avec des mémoires de plus en plus importantes (loi de Moore). P.T.P Tang proposa des algorithmes qui calculent ces fonctions beaucoup plus rapidement, grâce à l'utilisation de grosses tables de valeurs précalculées (grosse voulant dire : 1Ko). Depuis, tout le monde calcule ces fonctions en logiciel, et les instructions correspondantes sont inutilisées. Pourquoi ne pas implémenter les mêmes algorithmes en microcode, me direz-vous ? Parce que pour le coup, le coût en transistors de la table ne serait pas justifié.

Le troisième cas de non-correspondance entre ISA et architecture est d'ajouter à l'ISA des instructions qui ne sont pas implémentées dans les premières versions du processeur, mais dont tout porte à croire qu'elle le seront à l'avenir. Exemples : la précision quadruple (flottants de 128 bits) dans les SPARC, l'arithmétique flottante décimale dans certains processeurs IBM. Tant que le processeur n'implémente pas ces instructions en matériel, elles déclenchent un saut (techniquement, en utilisant le mécanisme des interruptions) vers une sous-routine logicielle qui traite l'instruction.

8.8 Un peu de poésie

IA32 a des instructions qui font de 1 à 17 octets. L'encodage est sans queue ni tête : il y a des préfixes, des suffixes, etc. Il n'y a que 8 registres entiers, et tous ne sont pas utilisables par toutes les instructions. Les opérations entières travaillent sur un modèle registre-registre ou registre-mémoire, et les opérations flottantes sur un modèle de pile (qui n'a jamais pu être implémenté comme initialement prévu, à savoir une pile virtuelle infinie dont la pile physique fonctionne comme un cache).

La bonne nouvelle est que les compilateurs arrivent très bien à vivre avec un sous ensemble très réduit de ce jeu d'instruction. Par exemple, certaines instructions traitent des chaînes de caractère (copie de chaîne etc), mais l'utilisation des instructions simples donne un code plus rapide.

Cherchez sur Internet le document qui spécifie le jeu d'instruction IA32, et comptez les centaines de pages.

Chapitre 9

Architecture d'un processeur RISC

Pour construire notre processeur, on procède en trois temps.
— D'abord on définit les différents blocs (boîte à registres, boîte à opérations ou ALU pour *arithmetic and logic unit*, les différents registres qui contiennent le PC ou les drapeaux, etc.
— Ensuite on les relie par des fils, éventuellement à travers des multiplexeurs.
— Enfin on construit un automate qui gère les différents signaux de commande qui vont permettre à l'architecture d'implémenter un cycle de von Neumann.

9.1 Un jeu d'instruction RISC facile

Nous allons implémenter un jeu d'instructions inspiré du ARM mais simplifié dont les caractéristiques sont les suivantes :
— Code 3 opérandes, 64 registres, toutes les instructions conditionnelles.
— Les bits 0 à 5 du mot d'instruction codent l'opération à effectuer. On a ainsi 64 instructions possibles.
— On a 4 drapeaux (archiclassiques)
  — Z (zero) qui vaut 1 si le résultat d'un calcul est nul
  — C (carry) qui contient la retenue sortante d'une addition
  — N (negative) qui contient une copie du bit de signe du résultat
  — V (oVerflow) qui indique si une addition/soustraction, **considérée en complément à 2**, a débordé. Techniquement, pour l'addition, si les deux opérandes ont des signes différents, le résultat ne déborde jamais. Si les deux opérandes sont du même signe, alors l'addition a débordé si le bit de signe du résultat est différent de celui des opérandes. Pour la soustraction, réfléchissez vous-même.
— Les bit 6 à 9 codent la condition sous laquelle s'exécute l'instruction. On a 16 conditions possibles (dont bien sûr "inconditionnellement").
— Le bit 16 code le ! : s'il vaut 1 on met à jour les drapeaux, s'il vaut 0 on ne les met pas à jour.
— Les bits suivants codent le numéro du registre destination (11 à 16), le numéro du premier registre opérande (17 à 22), et le champ correspondant au second opérande (23 à 28).
— Le bit 5 (dans le champ instruction) code en général si le second opérande est un registre (auquel cas le numéro du registre est codé dans les bits 23 à 28) ou bien si c'est une constante (auquel cas la constante est un entier positif codé par les bits 23 à 31).
— On a deux instructions `LetHigh` et `LetLow` qui mettent une constante de 16 bits dans le registre destination. La constante est codée dans les bits 17 à 31. Zut, il en manque un. Ce sera le bit 16 (le !) : qui a besoin de mettre à jour les drapeaux dans ce cas ?
— On a des sauts relatifs en avant et en arrière qui prennent aussi une constante positive, qui récupère aussi le bit 16, et en plus les bits 11 à 16, soit 22 bits en tout. On peut sauter de 4 millions d'instructions, c'est parfait.

— On a des sauts absolus qui prennent la destination dans un registre. Il y a plein de bits qu'ils n'utilisent pas.
— On a pour les accès mémoire : `Rj->[Ri]`, `Rd<-[Ri]`, `Rj->[PC+k]`, `Rd<-[PC+k]`. Pas d'accès indexé du genre `Rd->[Ri+Rd]` pour rester simple (il faudrait ajouter une sortie de Rd de la boite à registres juste pour cette instruction).

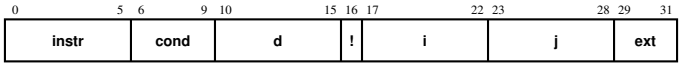


FIGURE 9.1 – Codage des instructions dans un mot d'instruction 32 bits

9.2 Plan de masse

J'ai déjà dit tout le bien que j'en pensais de la notion d'orthogonalité.

L'architecture de notre processeur putatif est donnée par la figure 9.2. Par rapport à ce que j'ai pu dessiner en cours, il est possible que ce soit plus propre. Sur les registres, l'entrée `we` pour *write enable* c'est ce qu'on a pu appeller aussi parfois *clock enable*.

L'architecture abstraite d'un processeur capable d'implémenter un jeu d'instructions 16 bits à 2 opérandes, construit en TD par vos camarades de l'ENS de 2001 (je crois), est donnée figure 9.3.

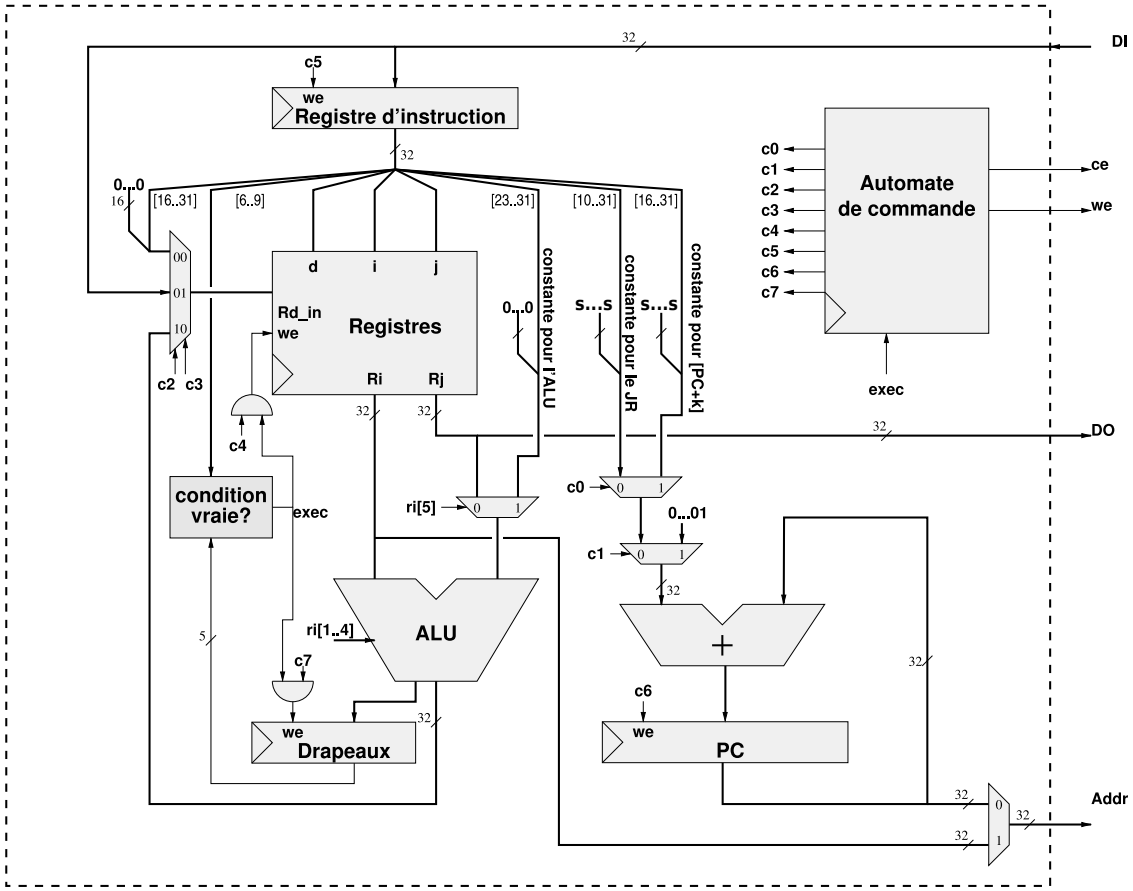


FIGURE 9.2 – En gros le processeur dessiné au tableau, pour ceux qui prennent des notes

Il faut vous convaincre que vous savez déjà tout construire là dedans.

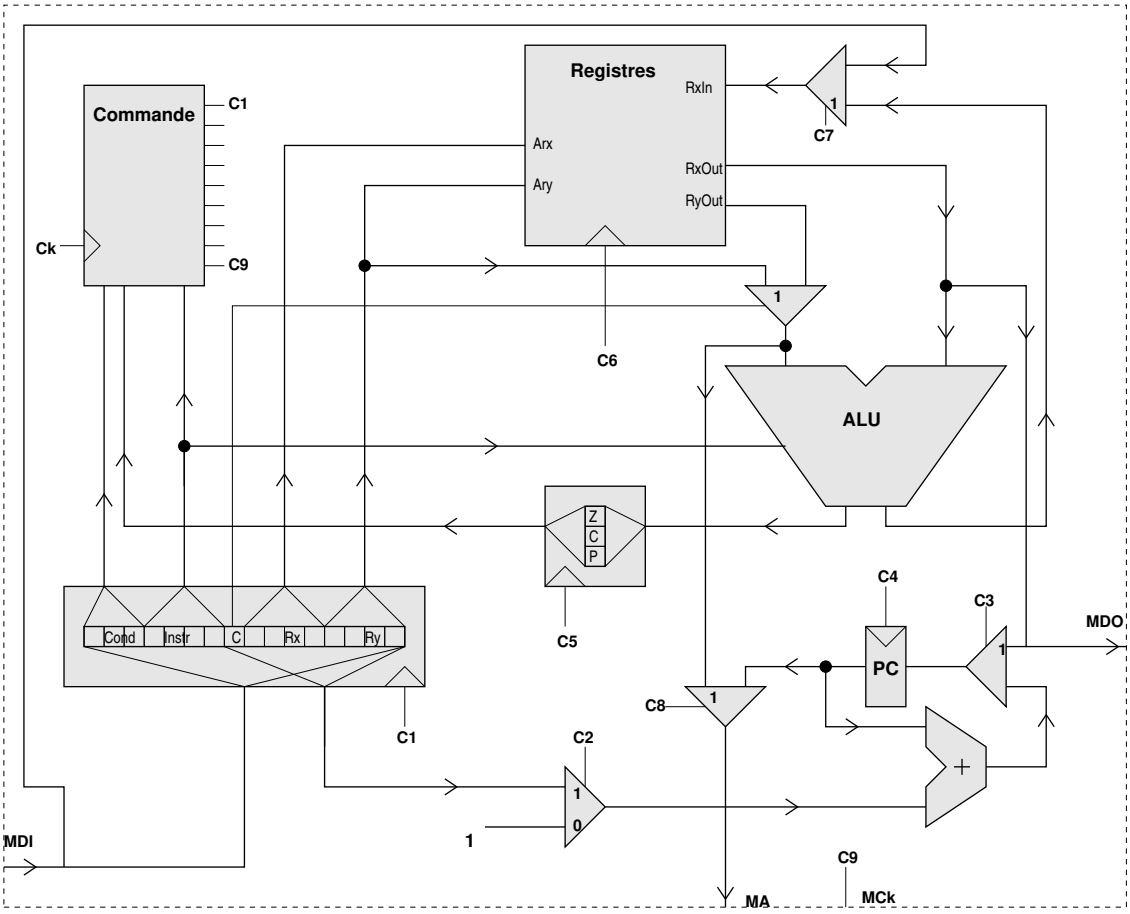


FIGURE 9.3 – On ne va pas en vendre beaucoup non plus, mais c'est presque le MSP430.

9.3 Construction de l'automate de commande

Une fois ce dessin fait, il ne reste plus qu'à construire l'automate intitulé "commande", et qui positionne tous les petits signaux de commande en fonction de l'instruction.

Il faut définir un ensemble d'état : ce sera en gros les étapes du cycle de von Neumann, avec plusieurs états pour l'étape d'exécution suivant la classe d'instruction qui s'exécute (opération de calcul, lecture mémoire, écriture mémoire ou branchement). On dessine au passage des chronogrammes qui spécifient ce qu'on veut, par exemple ceux de la figure 9.4 (actuellement tout faux). Sur ces chronogrammes, chaque instruction s'exécute en plusieurs cycles d'horloge.

Pour l'encodage des états, on regarde aussi le chronogramme, et on s'arrange pour que les bits codant les états soient directement des signaux de commande.

Enfin il ne reste qu'à dessiner l'automate correspondant puis l'implémenter comme vu au chapitre 6.

Et voilà un processeur universel qui fonctionne. C'est un objet de ce genre là qu'on vous demande de construire en TD. Dans le reste de ce chapitre nous explorons différentes manières de le rendre plus efficace.

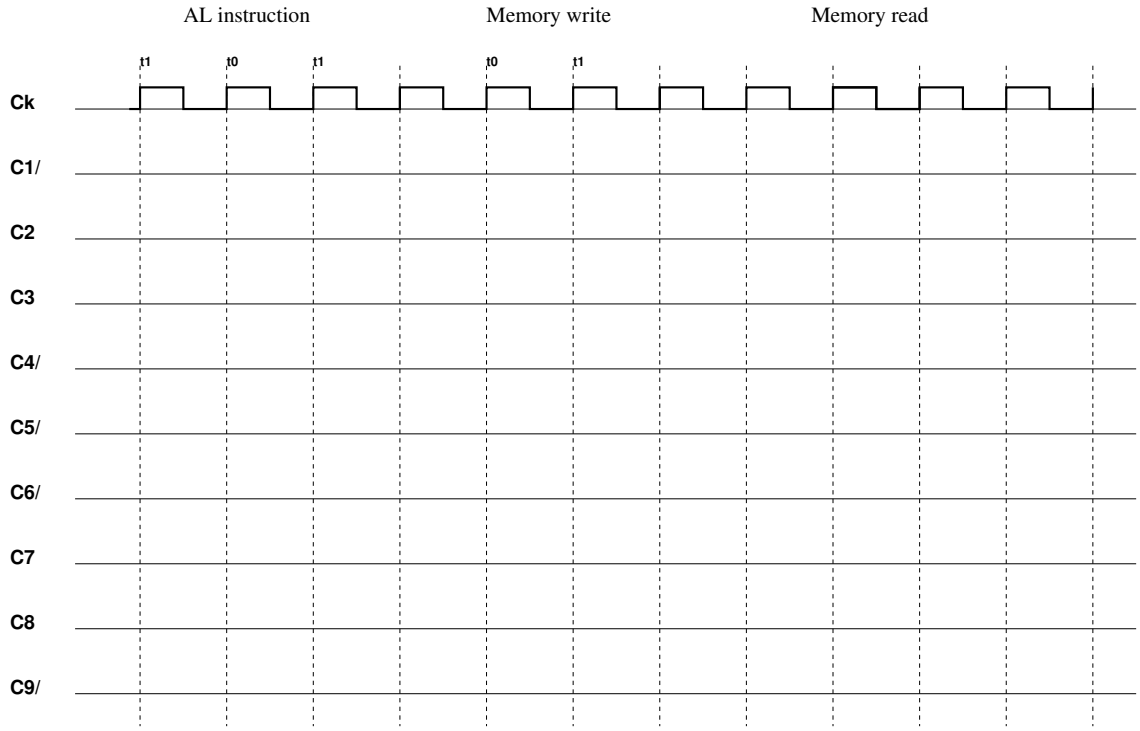


FIGURE 9.4 – Les chronogrammes lorsque le processeur de la figure 9.2 fonctionne. Chaque instruction s'exécute en deux cycles.

9.4 Pipeline d'exécution

Les bases

L'idée est simple : c'est celle du travail à la chaîne.

Faisons pour commencer l'hypothèse simplificatrice que notre processeur ne sait faire que des additions et des soustractions. On suppose donc qu'on n'a qu'une unité d'exécution, et on oublie momentanément les accès mémoire.

Alors, pendant que le processeur charge l'instruction n du programme (*instruction fetch* ou *IF*), il peut décoder l'instruction $n-1$ (*decode* ou *D*), lire les registres pour l'instruction $n-2$ (*operand load* ou *OL*), exécuter l'instruction $n-3$ (*execute* ou *EX*), et écrire le résultat de l'instruction $n-4$ dans le registre destination (*writeback* ou *WB*).

Exemple : On a du matériel qui fait l'IF en 1ns, une lecture et une écriture des registres en 1ns, et toutes les opérations en 1ns. Dessiner le diagramme d'exécution (en x le temps, en y les étages du pipeline) pour le programme suivant :

```
1: R1 <- add R2,R3
2: R4 <- sub R5,1
3: R6 <- add R7,R8
```

t	1	2	3	4	5	6	7	8	9											
IF																				
ID																				
OL																				
EX																				
MEM																				
WB																				

On définit ainsi les étages du pipeline. Techniquement jusque là c'est très simple, il suffit d'avoir un paquet de registres pour chaque étage, qui transmet l'instruction en cours d'un étage à l'autre. Remarquez qu'il faut transmettre le long du pipeline toutes les infos utiles à la vie future de l'instruction, par exemple son registre de destination.

Les accès mémoire

Maintenant parlons des accès mémoire. On va déjà supposer qu'on a un accès mémoire séparé programme/donnée, pour pouvoir EX une instruction mémoire en même temps qu'on IF une instruction subséquente. En pratique, on a effectivement deux accès mémoire distincts lorsqu'on a des mémoires caches séparées pour les instructions et pour les données (ICache et DCache). On verra en 12.1 que cela a plein d'autres avantages. Admettons pour le moment.

Ensuite on peut considérer qu'une instruction mémoire est une instruction comme les autres, qui s'exécute dans l'étage EX. Cela pose deux petits problèmes : le premier est que c'est vraiment une mécanique différente des autres qui s'occupe des accès mémoire. Le second est : adieu les modes d'adressages compliqués.

Donc on va plutot dire que toutes les instructions passent par un étage supplémentaire, dit *MEM*, après l'étage EX. Ciel, cela ajoute un cycle au temps d'exécution de chaque instruction. Oui mais on est en pipeline : si on considère le temps d'exécuter tout le programme de 100000 cycles, cela ne lui ajoute qu'un cycle aussi. Dessinez des chronogrammes de pipelines jusqu'à avoir bien compris la phrase précédente.

Les aléas de la vie

Considérons le programme suivant :

```
1               R0 <- xxxx    // adresse du vecteur X
2               R1 <- yyyy    // adresse du vecteur Y
3               R2 <- nnnn    // taille des vecteurs
4               R3 <- 0

5  .boucle      R10 <- [R0]
6               R11 <- [R1]
7               R12 <- R10 * R11
8               R3  <- R3 + R12
9               R0 <- R0 + 4
10              R1 <- R1 + 4
11              R2  <- R2 - 1
12         ifNZ B boucle
```

On a plein de *dépendances* dans ce programme. On parle de dépendance lorsqu'une instruction dépend pour s'exécuter du résultat d'une instruction précédente. Une autre manière de voir est que s'il n'y a pas de dépendance entre deux instructions, on peut les exécuter dans n'importe quel ordre. Ou en même temps (en parallèle).

On distingue

dépendance de donnée qui sont de trois types :

- *lecture après écriture* ou *RAW*. Par exemple l'instruction 7 doit attendre pour lire la valeur de R11 que l'instruction 6 ait écrit cette valeur dans R11.
- *écriture après lecture* ou *WAR*. Par exemple l'intruction 10 ne peut pas modifier R0 tant que l'instruction 5 n'a pas fini de la lire.
- *écriture après écriture* ou *WAW*. On n'en a pas ici, mais si on avait deux lignes commencant par `R1 <- ...`, on ne pourrait pas les échanger au risque de changer la sémantique du programme.

Seule la première (RAW) est une vraie dépendance, c'est à dire une dépendance d'un producteur vers un un consommateur. Les deux autres sont dites "fausses dépendances", car elles ne proviennent que du fait qu'on a un nombre de registres limités. On pourrait toujours les supprimer en utilisant un autre registre la seconde fois, si on avait un nombre infini de registres. Par exemple, la dépendance WAR de 5 à 10 se résolverait en utilisant R100 à la ligne 10 (et en renommant R100 toutes les occurences suivantes de R0 dans le programme). Cette notion est importante à comprendre, car on va avoir plein de techniques matérielles qui reviennent à renommer les registres pendant l'exécution du programme pour ne pas être gêné par les fausses dépendances.

Attention, il peut y avoir des dépendances de données entre deux instructions à travers d'une case mémoire. C'est plus difficile à détecter puisque cela ne se voit pas forcément à la compilation du programme.

dépendance de contrôle typiquement pour savoir quelle branche on prend en cas de saut conditionnel. Par exemple on ne peut pas décider ligne 12 du saut tant que la ligne 11 n'a pas produit la valeur du drapeau Z.

dépendance de ressources (encore une fausse dépendance qui pourrait être supprimée par plus de matériel) : si par exemple une lecture mémoire fait deux cycles et n'est pas pipelinée, l'exécution de l'instruction 6 dépend de ce que l'instruction 5 ait libéré la ressource.

Et pourquoi je vous embête avec cette notion dans le chapitre sur le pipeline ? D'une part parce que c'est une notion centrale en informatique (elle parle de la sémantique d'un programme). D'autre part parceque dans le pipeline, le W arrive longtemps après le R. Dans le cas de deux instructions consécutives avec une dépendance RAW, si on ne fait pas attention, le W de la première pourrait se dérouler après le R de la seconde, qui lirait alors n'importe quoi.

On gère les dépendances

1. d'abord en les détectant. Convainquez-vous que c'est relativement facile, bien que coûteux en matériel. En fait, c'est à cela que sert l'étage ID *(Instruction Decode)* entre IF et EX. Sinon, dans notre processeur RISC orthogonal, le décodage d'instruction était quasi trivial. Le plus coûteux serait de gérer les dépendances de données à travers la mémoire, mais pour le moment le problème ne se pose pas car lecture et écriture mémoire se font dans le même étage, donc dans l'ordre du programme).
2. ensuite en bloquant le pipeline (plus d'IF) en cas de dépendance jusqu'à ce que l'instruction bloquante ait avancé suffisamment pour que la dépendance soit résolue. On parle aussi d'insérer une bulle dans le pipeline (est-ce une métaphore ou une allégorie ?).
3. enfin en ajoutant des *court-circuits* à notre pipeline. Les deux principaux sont
   — un court-circuit des données : on avait un signal qui remontait le pipeline pour aller stocker dans la boite à registre le résultat des opérations durant WB. Ce signal est désormais espionné par l'étage EX, qui y repérera les couples (numéro de registre, valeur) qui l'intéressent. Ainsi, une dépendance de données ne bloque plus une instruction dans l'étage OL mais dans l'étage EX. En fait, on va même court-circuiter l'étage MEM pour les instructions qui ne font pas d'accès mémoire, mais attention alors aux conflits possibles entre un couple qui viendrait de l'étage EX et un couple qui viendrait de l'étage MEM au même cycle. Les court-circuits se voient bien sur le diagramme temporel.
   — Un court-circuit des drapeaux vers l'étage IF.
4. Une dernière technique pour gérer les dépendances de contrôle est l'*exécution spéculative*. On prédit le résultat d'un branchement, on charge le pipeline en fonction de la prédiction, et quand on s'est trompé on doit jeter des instructions en cours d'exécution (fastoche, c'est le fil *reset* des registres du pipeline, et aucune instruction n'est définitive tant qu'elle n'a pas écrit dans les registres). En terme de cycles perdus, c'est comme si on avait bullé sur la dépendance de contrôle, mais on ne paye ce prix que lorsqu'on s'est trompé dans la prédiction.

   Et comment qu'on prédit ?

— Heuristique simple : on prédit qu'un branchement est toujours pris. C'est (presque) vrai pour les branchements en arrière à cause des boucles, et vrai une fois sur deux pour les branchements en avant. Si on a toutes les instructions conditionnelles (ARM, IA64), les branchements en avant pour cause de `if` sont rares, et la prédiction est bonne a plus de 90% ($1/nnnn$ dans notre programme ci-dessus).
— On utilise un prédicteur de branchement : petit automate à 4 états (2 bits) associé à chaque adresse de branchement par une fonction de hachage simple (typiquement les 8-10 derniers bits de l'adresse de l'instruction de branchement). Il y a deux états pour chaque prédiction, et il faut se tromper deux fois pour changer la prédiction, si bien qu'on ne se trompe qu'une fois par boucle.

On voit que la gestion physique des dépendances est variable : par exemple, certaines dépendances vont bloquer les instructions à l'étage OL, d'autres à l'étage EX. Quand on va dans les détails, ces techniques sont très complexes. Il faut se convaincre qu'on peut partir du plus simple (buller) et aller vers le plus complexe (court-circuits et exécution spéculative).

En résumé, la notion de dépendance est une notion abstraite importante. Elle est gérée aussi en amont par le compilateur qui va construire le graphe de dépendance d'un programme pour minimiser leur impact. Reprenons notre programme : un compilateur optimiseur saura intercaler les instructions de gestion de boucle entre les instructions d'accès mémoire et de calcul qui présentent de vraies dépendance.

Fenêtres de registres *Les systèmes de fenêtres de registres permettent un renommage de chaque registre à chaque itération en une seule instruction (et dans IA64 c'est même dans l'instruction de branchement) : c'est la technique du* pipeline logiciel *qui permet (dans notre exemple) de charger un registre à l'itération n, de faire la multiplication et l'addition à l'itération $n+1$, et de ranger le résultat à l'itération $n+2$. Ainsi on a éloigné les instructions dépendantes suffisamment pour qu'elles aient chacune le temps de traverser tranquillement le pipeline.*
Il y a un surcoût : l'initialisation et la fin de la boucle sont plus compliquées, car il faut amorcer puis vider le pipeline logiciel. Donc le code est plus gros.

Aléas *Enfin, il y a des* aleas *qui se traitent comme des dépendances. Par exemple, une division par zéro doit provoquer une interruption du programme (ou* exception*). Concrètement, en découvrant que le dividende vaut zéro (dans l'étage EX), le processeur doit sauter à une routine d'erreur. Du coup, toutes les instructions qui suivaient la division, et qui avaient été lancées dans le pipeline, deviennent caduques : on doit vider le pipeline avant de traiter une exception. Il se passe la même chose en cas de* traitement d'interruption *en général.*
Un autre cas important est le cache miss *(on verra ce que c'est en 12.1, vous comprendrez donc cette phrase seulement à la seconde lecture du poly). Un* cache miss *va bloquer à l'étage MEM (on parle d'aléa mémoire).*
Comme le mécanisme de traitement des dépendances et celui de traitement des aléas sont similaires, les dépendances sont parfois aussi appelées aléas *(en Cockney,* hazard*). C'est le cas dans le Patterson et Hennessy, mais je n'aime pas du tout. Un aléa, pour moi, c'est aléatoire.*

La techno et les pipeline

Construire un pipeline simple c'est juste ajouter des registres. Pour qu'il soit efficace il faut
— qu'on ait bien équilibré la tranche de travail de chaque étage. Ce n'est pas si difficile, grâce à la notion de *retiming*.
— que le surcoût des registres (en terme de temps de traversée) soit négligeable. En 2002 il y a eu deux papiers dans la conf ISCA convergeant vers l'idée que la profondeur optimale était de 8 à 10 portes de base dans chaque étage. Le pentium 4 était en plein dedans. Malheureusement ces papiers ont négligé les nouveaux petits soucis des techno submicroniques, et le P4 n'a pas pu être poussé jusqu'aux fréquences prévues.

9.5 Exploitation du parallélisme d'instruction : superscalaire

Si on creuse, on constate que notre pipeline peut encore être amélioré.

— On a fait comme si l'étage EX était monolithique, mais en fait on a dedans un additionneur/subtracteur (1ns), un multiplieur (5ns, pipeliné), les mêmes en flottant (8ns), une unité logique (1ns), etc. Tous ces opérateurs pourraient fonctionner en parallèle, avec des profondeurs de pipelines différentes.

— Mais alors, les instructions vont se terminer dans le désordre. Remarque : le multiplieur en 5 cycles peut s'insérer dans notre pipeline de la section d'avant, il insérera 4 bulles, et voila tout.

— Tant qu'à faire, vu que les instructions se terminent dans le désordre, on peut les lancer dans le désordre.

— Tant qu'à vivre dans le désordre, on peut aussi lancer plusieurs instructions à la fois.

Un processeur qui fait tout cela, *tout en conservant une sémantique séquentielle au jeu d'instructions* est appelé superscalaire. Nous allons détailler les mécanismes qui rendent cela possible ci-dessous.

Sémantique d'un programme assembleur *La sémantique de l'assembleur est (normalement) définie par l'ISA.*

— *La sémantique d'une instruction est une fonction qui prend l'ordinateur dans un certain état, et le rend dans un autre état. L'état est constitué des valeurs des registres visibles de l'ISA (le PC, ceux de la boite à registre, les drapeaux) ainsi que des valeurs de la mémoire. Les autres registres (ceux ajoutés pour le pipeline, etc) ne doivent pas intervenir dans la sémantique.*

— *La sémantique d'un programme est construite par composition des sémantiques de ses instructions, avec un opérateur de point fixe pour les boucles. C'est cette sémantique que l'exécution dans le désordre doit préserver.*

Par opposition il existe des processeurs dont le jeu d'instruction a une sémantique exposant explicitement le parallélisme. L'extrême dans cette direction est un processeur *VLIW (very large instruction word)* dans lequel chaque instruction est un mot de (par exemple) 128 bits, et est composée de 4 sous-instructions de 32 bits destinées à être lancées sur autant d'unités d'exécution. On aura typiquement deux unité entière, un FMA flottant, et une unité mémoire. Remarquez qu'il n'y a pas lieu d'imposer l'orthogonalité du codage des instructions de chaque sous-classe. Les différents unités d'exécution sont toutes pipelinées, mais avec des profondeurs différentes. Dans la version extrême de cette idée, c'est le compilateur qui se charge de gérer toutes les dépendances de données : si le FMA fait 5 cycles, alors il ne produira pas de code dans lequel deux FMA dépendants sont séparés par moins de 5 instructions. Les bulles dans le pipeline sont ici des Nop (No Operation) insérés par le compilateur.

L'inconvénient du VLIW est d'avoir du code plus gros, car plein de Nops. Cela se soigne en ajoutant une petite couche de compression/décompression de code. Il faut se convaincre que cela sera moins coûteux que l'architecture superscalaire qui suit...

9.5.1 Architecture superscalaire (*)

Comment construit-on une architecture superscalaire ? Pour détecter les instructions qu'il peut lancer en parallèle, le processeur doit analyser les *dépendances* sur une fenêtre d'instruction en entrées. Il doit ensuite renommer les registres, d'une manière ou d'une autre (voir plus bas) pour faire disparaitre les fausses dépendances. Il peut alors sélectionner dans sa fenêtre des instructions indépendantes et les lancer. Une instruction se termine au bout d'un nombre de cycles variables, qui dépend non seulement de la latence de l'opération à effectuer, mais aussi des dépendances de données et même des aléas mémoire, etc. Lorsque le calcul est terminé, le processeur envoie d'abord le résultat à toutes les instructions qui l'attendent (à travers éventuellement les registres de renommage), puis il doit recopier le résultat (le registre de renommage)

dans les "vrais" registres (ceux du jeu d'instruction), en assurant la sémantique séquentielle. Ce processus est encore compliqué par le fait qu'on doit en plus maintenir la cohérence séquentielle en cas d'exception.

On va voir un exemple d'implémentation, mais encore une fois dans ce domaine l'imagination est reine. L'idée centrale est celle du renommage des registres, en gardant la correspondance avec les registres du code dans des tables. On va avoir plusieurs tables partiellement redondantes : c'est pour que leur accès soit rapide.

Analyse des dépendances et renommage

Les instructions qui arrivent sont lancées dans l'ordre ou le désordre (plusieurs par cycle). La seule condition à ce niveau pour lancer une instruction est qu'il y ait une unité libre pour la recevoir.

En fait l'instruction lancée atterrit dans une petite table de *stations de réservation* située en amont de l'unité d'exécution. Elle va y attendre la valeur de ses opérandes.

Chaque station de réservation (SR) contient une instruction en attente d'exécution, avec ses registres sources et destination, mais aussi des champs destinés à recevoir les opérandes. Ces champs peuvent contenir :

— soit la valeur de l'opérande, si elle était déjà disponible dans la boite à registres au lancement de l'instruction,

— soit un pointeur vers la station de réservation qui contient l'instruction qui va produire cet opérande, en cas de dépendance de donnée.

On a donc au niveau du lancement une table qui dit "tel registre sera produit par telle SR". C'est cette table, consultée avant le lancement des instructions suivantes, qui assure le renommage des registres opérandes et supprime ainsi les fausses dépendances. Elle peut aussi tout simplement indiquer "tel registre est disponible dans la boite à registres", c'est d'ailleurs son état au démarrage.

En résumé, les registres destination sont éventuellement renommés en SR au lancement. Ainsi, lorsqu'on a deux instructions proches qui écrivent dans R1, les deux R1 ne sont pas renommés dans la même SR. Bien sûr, au lancement d'une instruction dont la destination est R1, la table précédente est mise à jour pour dire "désormais R1 sera produit par telle SR".

Exécution dans le désordre

Il y a un bus commun des résultats (il est très large, typiquement plusieurs centaines de bits, car il peut y passer plusieurs résultats par cycles). L'info qui passe est de type (registre destination, SR de l'instruction, donnée). Chaque station surveille ce bus pour y attraper les opérandes qui manquent à ses SR (en comparant les SR de ses opérandes en attente aux SR qui passe sur le bus).

Enfin, à chaque cycle, chaque unité de calcul choisit, parmi ses SR, une entrée pour laquelle tous les opérandes sont disponibles, et la fournit au pipeline de calcul. La SR correspondante est alors libérée.

Au fait, le nombre de SR par unité de calcul est un paramètre dont la détermination est compliquée. Ce sera typiquement entre 1 et 4.

Terminaison dans l'ordre

C'est pas tout, mais il faut écrire dans la mémoire et dans les registres ISA dans l'ordre. Pour cela, il y a une unité de "terminaison dans l'ordre" en aval des unités d'exécutions, et qui surveille également le bus commun. Cette unité contient un tampon circulaire des instructions lancées et pas terminées *(reordering buffer, ROB)*. Chaque entrée est créee au lancement, et contient une instruction avec sa SR. Lorsqu'une instruction a fini de s'exécuter, son résultat passe sur le bus résultat suscité. Le ROB l'attrape et la marque comme terminée. À chaque cycle, le ROB prend

un paquet d'instructions terminées consécutives, et les termine effectivement, en écrivant leur résultats dans les registres ou en mémoire.

Si vous avez suivi, une SR peut servir plusieurs fois à des intervalles rapprochés. Pour assurer qu'on pointe bien vers la bonne instruction, les pointeurs vers des SR évoqués plus haut sont en fait des couples (SR, entrée de ROB).

La ROB gère les mauvaises prédictions de branchement (et autres exceptions) : en cas de mauvaise prédiction, on vide la ROB de toutes les instructions qui suivent le branchement mal prédit. Ces instructions vont finir de s'exécuter, mais leurs résultats n'arriveront jamais ni dans un registre ni dans la mémoire. Il faut également purger de la table "tel registre sera produit par telle SR" les informations mises par les instructions effacées de la ROB (cherchez comment, je ne suis pas sûr de moi). Puis on repart sur le code de la branche correcte.

Reste à assurer la cohérence séquentielle de la mémoire. C'est pour cela qu'on a besoin d'unités mémoire parmi nos unités de calcul. En lecture, elles doivent s'assurer avant de retourner la valeur lue qu'aucune des instructions précédemment lancée ne va modifier l'adresse à lire. Toute l'info est là, entre la ROB et les SR des unités d'écriture mémoire. En écriture, il y a un tampon qui calcule les adresse mais n'écrit qu'au feu vert de la ROB.

Au final, un vrai processeur va pouvoir lancer jusqu'à 4 à 6 instructions par cycles, en retirer autant, et en avoir bien plus en vol à un instant donné. Mais en moyenne, on aura un nombre d'instruction par cycles (IPC - l'inverse du CPI qui mesure la qualité du pipeline) qui dépassera rarement 2.

9.5.2 VLIW ou superscalaire (*)

Avant tout, faisons un peu de statistiques sur du vrai code. On constatera avec amertume que le parallélisme d'instruction moyen que l'on arrive à extraire (en tenant compte uniquement des vraies dépendances, pas des fausses) est de l'ordre de 3-4. Autrement dit, avec un processeur superscalaire idéal, on arriverait en moyenne à lancer 3-4 instructions en parallèle par cycle. C'est une question compliquée, d'une part parce qu'il existe des code intrinsèquement parallèles (mon éternel produit de matrices l'est presque), d'autre part parce que cela dépend du jeu d'instruction : avec des instructions plus complexes, donc plus puissantes, on réduit le parallélisme d'instruction... Disons qu'il y a un consensus actuel disant qu'il ne sera pas rentable de construire des processeurs ayant plus d'une dizaine d'unités d'exécutions : on ne saura pas les remplir avec du code séquentiel (et pour le code parallèle, voir plus bas, 9.7.1).

La différence fondamentale entre VLIW et superscalaire est que la gestion du parallélisme d'instruction est faite dans un cas par le compilateur, dans l'autre par du matériel. Voyns les avantages et les inconvénients.

— Le gros avantage du VLIW est la simplicité de son matériel : on a vu que la détection des dépendances et l'exécution dans le désordre sont coûteuses. De plus, elles consomment de l'énergie pour une tâche purement administrative, pas pour le calcul. Les VLIW consomment beaucoup moins.
— Le VLIW a typiquement du code plus gros (plein de Nop), ce qui finit par avoir un impact sur la conso et la perf (besoin de caches plus grands par exemple). Le superscalaire a du code plus compact.
— Le code VLIW n'est pas portable d'une archi à la suivante : il faut tout recompiler si la profondeur du pipeline FMA passe de 3 à 5 par exemple. À l'opposé, votre Pentium 4 extrait les dépendances de vieux code DOS aussi bien que du code plus récent.
— Le compilateur voit une fenêtre de code beaucoup plus grande, et est capable d'en extraire presque parfaitement tout le parallélisme d'instruction pour le donner au VLIW. Le matériel ne pourra considérer qu'une fenêtre plus petite du code en cours d'exécution.
— Par contre, le compilateur fait un travail statique, alors que le matériel a l'avantage de ne considérer que les dépendances sur la branche de code effectivement prise. Sur du code plein de `if`s, la détection des dépendances peut être beaucoup plus fine si faite en matériel.

Considérant tout ceci, les processeurs VLIW ont du succès dans le domaine embarqué, où

— la basse consommation est importante ;

— les traitements qui ont besoin de beaucoup de puissance de calcul sont très parallèles (multimedia, compressions de données) ;
— le code est compilé une fois pour toutes (donc la non-portabilité n'est pas vraiment un problème) ;

9.6 Deux jeux d'instructions récents

9.6.1 IA64

Le jeu d'instruction IA64, conçu pour le calcul haute performance, est un peu bâtard. Il considère les avantages et inconvénient ci-dessus, et essaye de prendre le meilleur des deux mondes en terme de performance. En principe, tout le travail que le compilateur peut faire pour un VLIW, il peut aussi le faire pour un superscalaire. Mais il faut pouvoir l'exprimer ensuite pour le jeu d'instruction.

Donc le jeu d'instruction IA64 est un VLIW bizarre : paquets de 128 bits pour 3 instructions, mais le parallélisme est exprimé surtout par des bits qui délimitent les ensembles d'instructions qu'on peut lancer en parallèle (et ces ensembles peuvent faire de une à beaucoup beaucoup de sous-instructions). Les sous-instructions ne sont pas directement en face du matériel correspondant, il y a une certaine liberté. Et il y a tout de même du matériel de détection des dépendances, pour que le code soit portable.

Au final, IA64 récupère aussi le pire des deux mondes (LCDE) : des compilateurs compliqués et inefficaces, et beaucoup de matériel consacré à l'administration. Mais la performance est au rendez-vous.

9.6.2 Kalray K1

Le processeur Kalray est un VLIW pur, car sa conception a été orientée vers la basse consommation. Une instruction VLIW (lancée chaque cycle) se compose de
— deux instructions entières 32 bits (ou une instruction 64 bits)
— une instruction de multiplication et/ou de calcul flottant, ou une troisième instruction entière.
— une instruction mémoire
— une instruction de branchement/contrôle.

Ce processeur n'est pas aussi orthogonal que mon ARM. En effet, la complexité matérielle du manque d'orthogonalité, tant qu'elle se limite à une petite couche de logique combinatoire pour le décodage des instructions, a un coût minime, donc on se l'autorise. La complexité en terme de conception est prise en charge par des outils avancés générant le processeur+compilateur.

Les registres sont complètement unifiés. Un registre 64 bits est obtenu en appariant deux registre 32 bits consécutifs.

L'exécution est complètement dans l'ordre, bien que l'exécution des instructions aient des latences variables : 1 cycle pour les instructions entières, 2 cycles pour les multiplications entières, 3 cycles pour les instructions flottantes.

Une originalité est l'instruction complexe (mais pas CISC) qui calcule $a \times b + c \times d$ en flottant.

Mais le point fort du processeur Kalray est d'intégrer 256 coeurs K1 sur une puce. Ce qui nous amène à la section suivante.

9.7 Exploitation du parallélisme de processus

Le fonctionnement en parallèle de deux processus partageant potentiellement de la mémoire amène un indéterminisme intrinsèque qu'on n'avait pas jusque là. En effet, que ce passe-t-il lorsque les deux veulent écrire deux valeurs différentes à la même adresse ? On saura construire du matériel qui tranche dans ce cas, mais il est difficile de donner une sémantique propre à ce

genre de situations. Il faut donc les éviter par des mécanismes de verrous, mais... c'est à la charge du programmeur : on ne sait plus donner une sémantique à tous les programmes.

Entre parenthèses, comme modèle de programmation, les threads sont une horreur. Le moindre programme de 4 lignes peut cacher des bugs horriblement subtils. Cela ne nous regarde pas vraiment dans ce cours, mais quand même, si vous pensez que les threads sont une bonne idée, googlez les deux petits articles suivants :

— *The problem with threads* par Edward Lee.
— *The trouble with multicore* par David Patterson.

9.7.1 Multifilature *(multithreading)*

L'idée est que le processeur est capable de gérer plusieurs contextes, qui exécuteront plusieurs processus légers (threads). On duplique essentiellement la boite à registres et le PC, et au augmente un peu la taille de certaines autres structures de données comme les stations de réservation.

Quand un thread doit attendre (accès mémoire lent, ou même dépendance de donnée ou branchement conditionnel), le processeur bascule sur un autre thread qui lui n'est pas bloqué. Bref, un thread remplit les bulles de l'autre et réciproquement. C'est une manière relativement simple et bon marché d'optimiser l'utilisation des ressources de calcul du processeur.

Il a été proposé des calculateurs sans cache, mais avec 128 threads pour recouvrir la latence de la mémoire. Il faut les trouver, les 128 threads, dans votre Tetris. Oracle vend actuellement des processeurs très multithreadés pour les serveurs web et autres, qui peuvent traiter des centaines de transactions à un instant donné.

La différence entre un thread et un processus, c'est que deux thread vont typiquement partager de la mémoire, alors que les processus vivent chacun dans leur mémoire, étanche aux autres processus. On verra comment cette étanchéité est assurée par le système.

HyperThreading, c'est le jargon marketing Intel pour dire "multithreading à 2 threads seulement"...

9.7.2 Processeurs multicoeurs

C'est facile, on duplique le processeur (voir les photos de puces multicoeurs). C'est pour cela que tout le monde en fait actuellement.

La difficulté est de faire travailler ces cœurs de manière

— cohérente (sans indéterminisme)
— et efficace (sans que la plupart des cœurs ne passent la plupart du temps à se tourner les pouces).

Il y a principalement deux manières de faire coopérer plusieurs cœurs :

— en leur faisant partager la même mémoire,
— en leur donnant chacun une mémoire privée, et en leur permettant d'échanger des messages.

En terme de modèle de programmation, le partage de mémoire est plus confortable. En terme d'architecture, l'échange de messages est plus simple à construire. En pratique, on peut émuler l'un par l'autre et réciproquement. On trouvera typiquement du partage de mémoire au niveau d'une puce (processeur multicœur), et de l'échange de message entre deux PC/blades/cartes mères.

On verra en 12.1.8 comment construire un processeur multicœur à mémoire partagée efficace.

Chapitre 10

Interfaces d'entrée/sorties

Et le clavier ? Et l'écran ? Et la carte son ? Et la prise réseau ? Tous ces gadgets sont appelés *périphériques* (en canadien de l'ouest *devices*).

Pour gérer un périphérique on met en place des *protocoles* partagés par l'ordinateur et le périphérique. Deux exemples (très simplifiés)

— Ce que vous tapez au clavier s'accumule dans une zone de mémoire appelée le *tampon* (ou *buffer*) clavier. L'ordinateur, lorsque cela lui chante, lit ces informations, puis informe le clavier (par une écriture dans une autre zone de mémoire) qu'il les a lues, et donc que la zone correspondante du tampon est à nouveau disponible.

— De même, l'ordinateur écrit, quand cela lui chante, dans une autre zone mémoire, des valeurs de couleurs. La carte vidéo, à la fréquence dictée par le moniteur, accède à ces mêmes cases mémoires et envoie l'information correspondante au moniteur qui les jette sur l'écran avec son canon à électrons, formant une image. Enfin du temps des moniteurs cathodiques c'était cela.

Plus généralement, les entrées/sorties s'appuyent sur deux mécanismes :

— la projection en mémoire des entrées/sorties *(memory-mapped IO)*. Le périphérique a le droit d'écrire ou de lire des informations dans certaines zones bien précises de la mémoire, et ce dans le dos du processeur. Le matériel qui implémente cette interface s'appelle *device controler* dans le texte, parfois *coupleur* en québecois.

— un mécanisme d'*interruptions* permet à un signal externe d'interrompre temporairement le cycle de von Neumann : le PC courant est mis de côté, et sa valeur est remplacée par l'adresse d'une *routine de traitement d'interruption (interrupt handler)* qui fait en général partie du système d'exploitation. Le processeur se consacre à gérer ce qui a provoqué l'interruption, puis une instruction spéciale (RETI pour *return from interruption*) restaure le PC qui avait été mis de côté, et l'exécution reprend là où elle avait été interrompue. Dans les détails c'est plus compliqué que cela : il y a plusieurs niveaux de priorité d'interruption, et plusieurs modes de processeur (plus ou moins interruptibles) pour les gérer.

On voit que le protocole qui régit la communication avec le périphérique est aussi en partie implémenté par du logiciel, qu'on appelle *device driver* c'est-à dire *pilote de périphérique*.

Naturellement, pour des raisons de sécurité ou de performance, tout ceci a été raffiné. Par exemple, l'accès à la mémoire video fonctionne à plein régime tout le temps. Pour un écran de 1024×800 pixels codés sur 4 octets, rafraîchi à 60Hz (à moins, il fatigue les yeux), la carte video doit envoyer $60 \times 4 \times 1024 \times 800 = 196608000$ octets/s au moniteur. Cela justifie que ces accès ne passent pas par le bus général de l'ordinateur, comme jadis. Il y a désormais un bus dédié à la vidéo.

Le problème reste que le processeur doit calculer et envoyer à la carte video une quantité similaire de données par seconde pour définir l'image, dans les contextes où elle bouge beaucoup (Lara Croft poursuivie par les aliens, par exemple). L'idée suivante est donc de faire réaliser les calculs graphiques (rotations, perspective et éclairage) par la carte video. L'avantage n'est pas uniquement que cela fait moins de boulot au processeur : cela fait aussi beaucoup moins de

communications entre processeur et carte graphique, puisqu'une image est alors décrite par un ensemble de triangles et de textures, qui est beaucoup plus compact.

La notion importante est que les entrées/sorties sont le plus souvent *asynchrones*, c'est-à-dire selon un minutage indépendant de celui du processeur. C'est plus simple ainsi.

Chapitre 11

Du langage au processeur (*)

Ce chapitre a eu sa raison d'être à une époque, il ne sera sans doute pas traité en cours mais on le laisse quand même.

11.1 Introduction : langages interprétés, langages compilés, processeurs virtuels

... à l'oral.

C, C++, Fortran, qui sont compilés directement en instructions machines.

Python ou Javascript sont interprétés.

Java ou C# sont compilés, mais pour un processeur virtuel qui va tourner dans une machine virtuelle munie d'un processeur virtuel (purement simulé) et d'un espace d'adressage virtuel. Avantages : portabilité, *sandboxing*. Inconvénient : performance. Solution à l'inconvénient : compilation *Just In Time*, qui traduit les parties les plus utilisées du programme assembleur virtuel directement en "vraies" instructions machines. Exemple : Dalvik pour Android.

Ne pas confondre... *Au passage, ne pas confondre machine virtuelle comme dans JVM* (Java Virtual Machine) *et machine virtuelle comme dans VirtualBox.*
La première émule un processeur différent du processeur matériel, qui peut même être irréaliste en matériel.
La seconde émule la machine au sens boîtier avec ses périphériques etc, mais avec le même processeur, en faisant au final tourner le code de la machine émulée aussi directement que possible sur le matériel.
Il y a aussi des émulateurs de machines. Par exemple je peux faire tourner dans une machine virtuelle VirtualBox un navigateur qui utilise une machine virtuelle Java pour faire tourner dedans un émulateur de Gameboy qui émule entre autre le processeur Z80 de la Gameboy. Merci à la loi de Moore, avec ces 4 couches d'émulation/virtualisation, les jeux tournent quand même plus vite que sur la console d'origine...
Il y a aussi la notion de mémoire virtuelle, qu'on verra dans le chapitre suivant, et qui est encore autre chose.

Dans toute la suite on parle de langages compilés nativement, et en fait on va surtout compiler le C [1].

11.2 L'arrière-cuisine du compilateur

11.2.1 Variables et expressions

Il faut comprendre une variable comme le contenu d'une case mémoire (ou de plusieurs cases consécutives si nécessaires). Par exemple si la variable A est stockée à l'adresse @A, la ligne de C

1. C'est une contrepèterie.

`A=A+1;` se décompose en une séquence d'instructions machines qui vont 1/rapatrier le contenu de la case d'adresse @A dans un registre du processeur ; 2/ ajouter 1 à ce registre, et 3/ écrire en mémoire le résultat à l'adresse @A.

Si vous ne demandez pas d'optimisation particulière à gcc, le code assembleur produit ressemble exactement à cela, et c'est pourquoi vous trouverez des séquences de code visiblement stupides, par exemple dans mon programme jouet compilé vers ARM :

```
str r3, [fp, #-8]
ldr r3, [fp, #-8]
```

Oui, il s'agit bien d'une écriture d'un registre à une adresse, immédiatement suivie d'une lecture de la même adresse pour remettre son contenu dans le registre où il est déjà. C'est idiot. Explication : le programme est compilé instruction C par instruction C ; l'écriture termine une instruction C (`variable=expression;`. La lecture commence l'instruction C suivante.

Au passage, voila pourquoi on utilise le même mot dans "compilation d'un programme" et "compilation des meilleurs tubes des années 70". Dans les deux cas c'est la juxtaposition de morceaux indépendants.

Si vous branchez les optimisations (gcc -O1, -O2, etc) le compilateur fera le ménage dans ces instructions inutiles.

Une expression arithmétique est un arbre, et en général il faut parcourir l'arbre pour réaliser les calculs intermédiaires dans le bon ordre, éventuellement en les stockant dans des cases mémoires temporaires.

11.2.2 Désucrage des opérations de contrôle de flot

Conditionnelle `if-then-else` On peut toujours implémenter le `if-then-else` par un test et un saut conditionnel (et possiblement deux autres sauts non conditionnels en cas de `else`.

Si le jeu d'instruction supporte les instructions conditionnelles (le *conditional move* est assez courant) ou les instructions prédiquées, on peut aussi implémentater un `if-then` ou un `if-then-else` par prédication. La prédication consiste à conditionner au résultat du test l'exécution d'instructions qui ne sont pas des sauts. Par exemple, pour un `if-then-else`, on exécutera les deux branches pour au final ne retenir que celle qui correspond au résultat du test.

Cela peut valoir le coup parce que les branchements sont coûteux dans un processeur pipeliné.

Boucles L'implémentation d'une boucle (For, Do, While) se fait selon une variation du schéma suivant (se reporter au document donnant la sémantique du langage) :

1. évaluer l'expression contrôlant la boucle
2. si elle est fausse, brancher après la fin de la boucle, sinon commencer le corps de boucle
3. à la fin du corps de boucle, réévaluer l'expression de contrôle
4. si elle est vraie, brancher au début de la boucle sinon continuer sur l'instruction suivante

Il y a donc un branchement par exécution du corps de boucle. En général un branchement produit une certaine latence (quelques cycles).

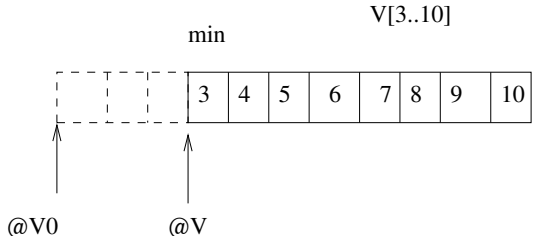
Case aka switch La difficulté d'implémentation des instructions `case` (e.g. `switch` en C) est de sélectionner la bonne branche efficacement. Le moyen le plus simple est de faire une passe linéaire sur les conditions comme si c'était une suite imbriquée de `if-then-else`. Dans certains cas, les différents tests de branchement peuvent être ordonnés (test d'une valeur par exemple), dans ce cas on peut faire une recherche dichotomique du branchement recherché ($O(log(n))$ au lieu de $O(n)$).

On peut aussi faire des sauts à une adresse calculée à partir de la variable du switch, ou utiliser une table d'adresses. Dans ce cas, la décision est en temps constant.

11.2.3 Tableaux

Le stockage et l'accès aux tableau sont extrêmement fréquents et nécessitent donc une attention particulière. Commençons par la référence à un simple vecteur (tableau à une dimension). Considérons que V a été déclaré par $V[min \dots max]$. Pour accéder à $V[i]$, le compilateur devra calculer l'*offset* de cet élément du tableau par rapport à l'adresse de base à partir de laquelle il est stocké. L'offset est $(i - min) \times w$ ou w est la taille des éléments de w. Si $min = 0$ et w est une puissance de deux, le code généré s'en trouvera simplifié (les multiplications par des puissance de 2 se font par décalage de bits, ce qui est en général plus rapide qu'une multiplication).

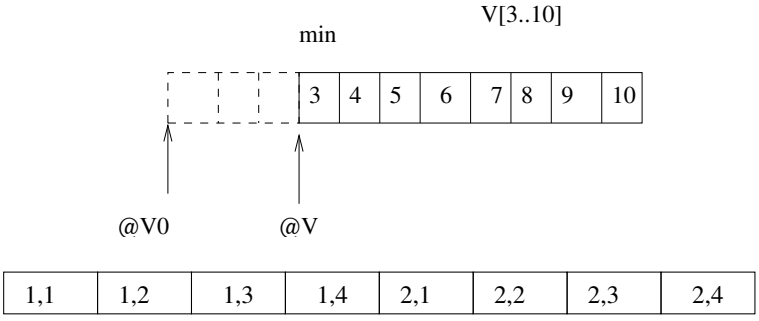
Si la valeur de min est connue à la compilation, le compilateur peut calculer l'adresse (virtuelle) $@V_0$ qu'aurait le tableau s'il commençait à 0 (on appelle quelquefois cela le faux zéro), cela évite une soustraction à chaque accès aux tableaux :



Si la valeur de min n'est pas connue à la compilation, le faux zéro peut être calculé lors de l'initialisation du tableau.

Pour les tableaux multidimensionnels, il faut choisir la façon d'envoyer les indices dans la mémoire. Il y a essentiellement trois méthodes, les schémas par lignes et par colonnes et les tableaux de vecteurs.

Par exemple, le tableau $A[1..2, 1..4]$ comporte deux lignes et quatre colonnes. S'il est rangé par lignes (*row major order*), on aura la disposition suivante en mémoire :



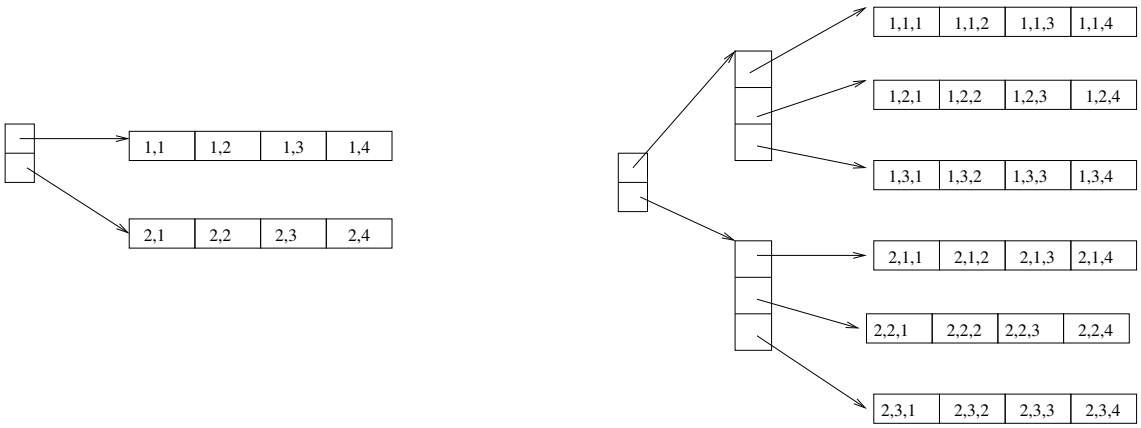
S'il est rangé par colonne on aura :

1,1	2,1	1,2	2,2	1,3	2,3	1,4	2,4

Lors d'un parcours du tableau, cet ordre aura de l'importance. En particulier si l'ensemble du tableau ne tient pas dans le cache. Par exemple avec la boucle suivante, on parcourt les cases par lignes ; si l'on intervertit les boucles i et j, on parcourera par colonnes :

```
for i ← 1 to 2
    for j ← 1 to 4
        A[i, j] ← A[i, j] + 1
```

Certains langages (java) utilisent un tableau de pointeurs sur des vecteurs. Voici le schéma pour $A[1..2, 1..4]$ et $B[1..2, 1..3, 1..4]$



Ce schéma est plus gourmand en place (la taille des vecteurs de pointeur grossit quadratiquement avec la dimension) et l'accès n'est pas très rapide, de plus le programmeur doit écrire une boucle pour allouer le tableau ligne par ligne. Concernant la place, ce schéma permet toutefois d'implanter efficacemement des tableaux non rectangulaires (matrices triangulaires) mais c'est sans doute anecdotique.

Considérons un tableau $A[min_1..max_1, min_2..max_2]$ rangé par ligne pour lequel on voudrait accéder à l'élément $A[i, j]$. Le calcul de l'adresse est : $@A[i, j] = @A + (i - min_1) \times (max_2 - min_2 + 1) \times w + (j - min_2) \times w$, ou w est la taille des données du tableau. Si on nomme $long_2 = (max_2 - min_2 + 1)$ et que l'on développe on obtient : $@A[i, j] = @A + i \times long_2 \times w + j \times w - (min_1 \times long_2 + min_2 \times w)$. On peut donc aussi précalculer un faux zéro $@A_0 = @A - (min_1 \times long_2 \times w + min_2 \times w)$ et accéder l'élément $A[i, j]$ par $@A_0 + (i \times long_2 + j) \times w$. Si les bornes et la taille ne sont pas connues à la compilation, on peut aussi effectuer ces calculs à l'initialisation. Les mêmes optimisations sont faites pour les tableaux rangés par colonnes.

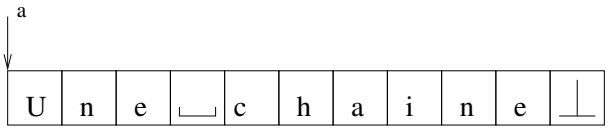
L'accès aux tableaux utilisant des pointeurs de vecteurs nécessite simplement deux instructions par dimension (chargement de la base de la dimension i, offset jusqu'à l'élément dans la dimension i). Pour les machines où l'accès à la mémoire était rapide comparé aux opérations arithmétiques, cela valait le coup (ordinateur avant 1985).

Lorsqu'on passe un tableau en paramètre, on le passe généralement par référence, même si dans le programme source, il est passé par valeurs. Lorsqu'un tableau est passé en paramètre à une procédure, la procédure ne connaît pas forcément ses caractéristiques (elles peuvent changer suivant les appels comme en C). Pour cela le système a besoin d'un mécanisme permettant de récupérer ces caractéristiques (dimensions, taille). Cela est fait grâce au descripteur de tableau (*dope vector*). Un descripteur de tableau contient en général un pointeur sur le début du tableau et les tailles des différentes dimensions. Le descripteur a une taille connue à la compilation (dépendant uniquement de la dimension) et peut donc être alloué dans l'AR de la procédure appelée.

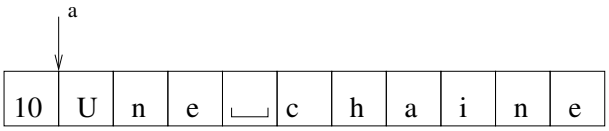
Beaucoup de compilateurs essaient de détecter les accès en dehors des bornes des tableaux. La méthode la plus simple est de faire un test sur les indices avant chaque appel à l'exécution. On peut faire un test moins précis en comparant l'offset calculé à la taille totale du tableau.

11.2.4 Chaînes de caractères

La manipulation des chaînes de caractères est assez différente suivant les langages. Elle utilise en général des instructions opérant au niveau des octets. Le choix de la représentation des chaînes de caractères à un impact important sur les performance de leur manipulation. On peut stocker une chaîne en utilisant un marqueur de fin de chaîne comme en C traditionnel :



ou en stockant la longueur de la chaîne au début (ou la taille effectivement occupée, si ce n'est pas la taille allouée) :



La plupart des assembleurs possèdent des opérations pour manipuler les caractères (sur les ARM, `LDRB` au lieu de `LDR` pour charger un octet. Pour les MSP430, exercice : allez chercher dans la doc donnée en TP). Pour cette raison, la mémoire est le plus souvent également adressée par octet, même pour une ISA pure 32bit, comme ARM.

Toutefois, sur ARM on impose que les chargements de mots de 32 bits (par LDR, mais aussi le chargement des instructions) soit uniquement sur des adresses multiples de 4 (octets). Sur les pentium modernes aussi, il est très conseillé d'*aligner* les accès mémoire pour les données 32 et 64 bits sur des adresses multiples respectivement de 4 et 8 octets. Sinon, je crois que cela marche mais je suis certain que cela prend plus du double du temps.

Cela dit il y a eu des architectures 32 bits où la mémoire était adressée par mots de 32 bits (le DEC Alpha arrivait à tourner à des fréquences bien supérieures à ses concurrents, grâce en partie à ce genre de sacrifice). Dans ce cas on doit faire des décalages et des masques pour extraire un octet particulier d'un mot 32bits.

Pour les manipulations de chaînes (comme pour les tableaux d'ailleurs), beaucoup d'assembleurs possèdent des instructions de chargement et de stockage avec autoincrément (pré ou post), c'est à dire que l'instruction incrémente l'adresse servant à l'accès en même temps qu'elle fait l'accès. Ceci permet de faire directement l'accès suivant au mot (resp. caractère suivant) juste après.

11.2.5 Structures

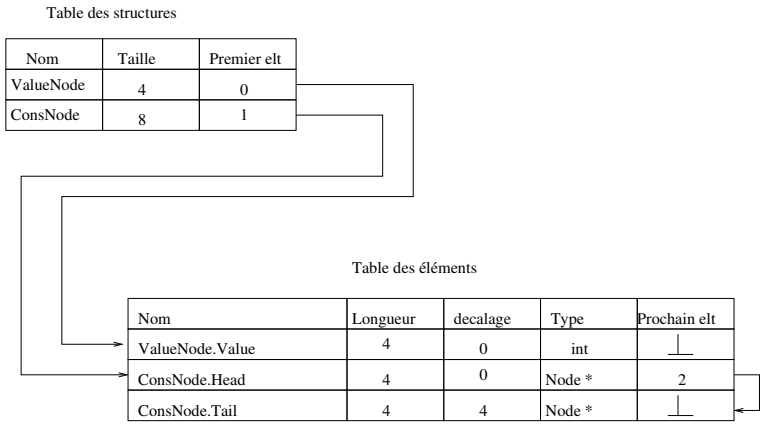
Les structures sont utilisées dans de nombreux langages. Elles sont très souvent manipulées à l'aide de pointeurs et créent donc de nombreuses valeurs ambiguës. Par exemple, pour faire une liste en C, on peut déclarer les types suivants :

```
struct ValueNode {
    int Value
};

structu ConsNode {
    Node *Head;
    Node *Tail;
};
```

```
Union Node{
    struct ValueNode;
    struct ConsNode;
};
```

Afin d'émettre du code pour une référence à une structure, le compilateur doit connaître l'adresse de base de l'objet ainsi que le décalage et la longueur de chaque élément. Le compilateur construit en général une table séparée comprenant ces informations (en général une table pour les structures et une table pour les éléments).



Avec cette table, le compilateur génère du code. Par exemple pour accéder à p1 $\rightarrow$ Head on peut générer l'instruction :

$$loadA0 \quad r_{p1}, 0 \quad \Rightarrow \quad r_2 \qquad \text{// 0 est le décalage de 'Head'}$$

Si un programme déclare un tableau de structure dans un langage dans lequel le programmeur ne peut pas avoir accès à l'adresse à laquelle est stockée la structure, le compilateur peut choisir de stocker cela sous la forme d'un tableau dont les éléments sont des structures ou sous la forme d'une structure dont les éléments sont des tableaux. Les performances peuvent être très différentes, là encore à cause du cache.

11.3 Application binary interface

l'ABI *(application binary interface)* est un contrat entre le compilateur, le système d'exploitation et la machine cible. Le but de ce contrat est de fixer les règles qui permettront à ces trois acteurs de coopérer harmonieusement :

Ce contrat répartit les responsabilités de chacun, et fixe des règles du jeu pour

— l'allocation des ressources (quel registre est réservé pour quoi ? qui est chargé de réserver l'espace mémoire des différentes variables ? quel registre sera utilisé comme le pointeur de pile lorsqu'il y a le choix comme sur ARM ?),

— le comportement (quels registres une procédure a le droit d'écraser, quels registres une procédure a le devoir de rendre dans l'état où elle l'a trouvé),

— le placement en mémoire (où est le point d'entrée d'une procédure, où sont ses variables locales et globales ?).

Nous allons décrire une ABI typique pour le C et Fortran, qui doit permettre

— la compilation séparée des procédures et la construction de bibliothèques.

— les procédures récursives

L'ABI est contrainte par les possibilités de l'ISA, mais est indépendante de l'ISA : c'est essentiellement un ensemble de conventions. Par exemple Windows et Linux, sur le même processeur, n'ont pas la même ABI.

11.3.1 Procédures et compilation séparée

Pour la plupart des langages impératifs, la procédure est l'unité de base. Le compilateur permet par exemple la *compilation séparée* des procédures. C'était indispensables aux temps héroïques de l'informatique. De nos jours, le moindre PC peut en théorie gérer dans sa mémoire centrale des programmes énormes, mais on continue à faire de la compilation séparée pour ses nombreux avantages :

— gain de temps lors du développement : on ne recompile que ce que est nécessaire,

— dualement, possibilité de passer plus de temps à faire des optimisation poussées sur des morceaux de codes assez petits,
— possibilité de distribuer des bibliothèques pré-compilées, sans distribuer le code source
— Possibilité d'édition de lien dynamique.

Tous ces avantages viennent au prix d'une étape d'*édition de lien*, qui assemble le code compilé des différentes procédures pour en faire un programme complet. L'édition de lien autorise à compiler du code pour une procédure, indépendamment du programme dans lequel elle va s'insérer.

La compilation séparée est encore plus intéressante si l'on peut appeler depuis un programme en C une bibliothèque écrite en Fortran, et réciproquement. Pour que ce soit possible, il faut que l'interfaces des procédures au niveau du code objet soit compatible avec tous les langages.

Cette interface doit donc offrir l'union de tous les langages cibles envisagés, et pourra donc paraître inutilement compliquée lorsqu'on regarde le code compilé pour un langage simple comme C. Deux exemples :

— Pascal a des procédures imbriquées, pas C [2]. Une procédure imbriquée, c'est une déclaration de procédure locale à une autre procédure. Il y a donc plusieurs espaces de nommage imbriqués.
— Les langages objets nécessitent des mécanismes spécifiques pour permettre héritage et surcharge, encore une fois pour savoir à quelle sur-classe appartient une variable d'un objet. Ces mécanismes seront inutilisés en C.

La compilation d'une procédure doit faire le lien entre la vision haut niveau (espace des noms hiérarchique, code modulaire) et l'exécution matérielle qui voit la mémoire comme un simple tableau linéaire et l'exécution comme une simple incrémentation du PC.

11.3.2 Récursivité et pile

Considérons l'exécution d'une procédure récursive (pensez à la factorielle ou à Fibonacci). Lorsque j'appelle `factorielle(10)`, l'ordinateur doit

1. réserver de la place pour les variables locales de la procédure `factorielle()`
2. brancher au début du code de `factorielle()`
3. revenir d'où il vient à la fin de ce code.

Les processeurs offrent tous une instruction machine qui permet ce type de "branchement en se souvenant d'où on vient".

— Sur la plupart des processeurs (IA32 et MSP430 inclus), on a deux instructions, `call` et `ret`. L'instruction `call adresse` réalise deux opérations :
  1. empiler l'adresse de l'instruction suivant le call (PC+1, pour une certaine valeur de 1)
  2. faire `PC<-adresse`

  L'instruction `ret` n'a pas d'argument, mais c'est aussi un branchement : elle dépile une adresse, puis branche à cette adresse.
— Sur les ARM et quelques autres, `call` est remplacé par `BL adresse` qui fait également deux choses :
  1. une copie de R15+4 (le PC de l'instruction suivant le BL) dans R14 (le *link register*
  2. puis un saut vers `adresse`

  L'instruction `ret` est remplacée par un `MOV R14 -> R15`. L'intérêt est de ne pas faire d'accès mémoire quand on n'en n'a pas besoin (appel à une procédure terminale, c'est-à-dire qui n'appelle personne d'autre). En cas d'appel récursif, la procédure appelée devra quand même empiler R14 au début, et le dépiler à la fin.

Je rappelle que toutes ces instructions sont atomiques, c'est-à-dire ininterruptibles. On pourrait en principe les émuler chacune par plusieurs instructions (des push, des pop et des sauts) mais ce serait le bordel en cas d'interruption au milieu.

Donc on a une *pile d'appel*. La récursivité nécessite fondamentalement un mécanisme de pile.

2. A vrai dire, `gcc` accepte les procédures imbriquées, c'est une extension GNU à C. Mais je vous défends formellement de vous en servir

11.3.3 Variables locales et passage de valeurs sur la pile

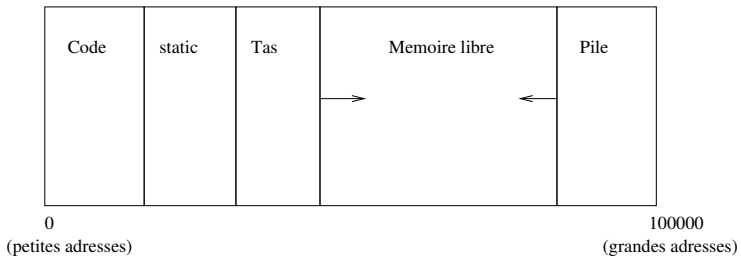
Cette pile est également l'espace de stockage idéal pour les variables locales d'une procédure (voir la figure 11.1) :

— L'allocation et la déallocation se font par des additions et soustractions sur le pointeur de pile, au début et à la fin de la procédure. Pas besoin de `malloc` compliqué.
— Pour ma factorielle, les différentes *instances* de la fonction (`factorielle(10)` a appelé `factorielle(9)` qui a appelé `factorielle(8)`, etc) ont chacune leurs variables locales sur la pile. En haut de la pile, j'ai toujours les variables locales de la procédure en cours d'exécution.
— Du coup, pour le processeur, l'adresse d'une variable locale est `SP+cste`. La constante est appelée aussi *offset* de la variable.
— Le passage des paramètres peut également se faire sur la pile : lorsque `toto` appelle `tata(17,42)`, les valeurs 17 et 42 seront empilées juste avant le `call tata`. La procédure appelée sait qu'elle a deux paramètres et qu'elle les trouvera aux adresses `SP-1` et `SP-2`. On pourrait aussi les passer dans des registres, mais on a parfois plus de paramètres à passer que de registres [3]
— De même, la procédure peut passer sa ou ses valeurs de retour sur la pile, où l'appelant les retrouvera au retour du `call` [4].

11.3.4 Vision globale de la mémoire

Tous les langages permettent aussi d'allouer de la mémoire dynamiquement (par exemple pour les variables dont la taille n'est pas connue à la compilation). Cette mémoire est allouée sur le *tas*. Typiquement, les variables dont la taille est petite et connue à la compilation (entiers, flottants, pointeurs sur les objets) sont allouées dans la pile. Par contre les tableaux, les objets, etc sont alloués sur le tas. En C, C++ ou Java cette allocation est explicite (il faut des `malloc` ou des `new`).

La mémoire linéaire est généralement organisée de la façon suivante : la pile sert à 'allocation des appels de procédures (elle grandit par exemple vers les petites adresses), le tas sert à allouer la mémoire dynamiquement (il grandit alors vers les grandes adresses). Ceci aussi fait partie de l'ABI, en particulier pour l'ARM pour lequel n'importe quel registre peut servir de pointeur de pile, et les piles peuvent monter ou descendre.



Ce n'est pas encore tout. Une donnée peut, suivant sa portée et son type, être stockée en différents endroits de la mémoire (on parle de classe de stockage, *storage class*). Les principales sont

— stockage local à une procédure (sur la pile, déjà vu),
— stockage dynamique (sur le tas, déjà vu),
— stockage statique d'une procédure (à côté du code),
— stockage global.

3. Les vraies ABI définissent des politiques du genre "si la procédure a moins de 4 paramètres, ils sont passés dans les registres R0 à R4. Sinon, les 4 premiers sont passés dans les registres et tous les suivants sur la pile. Une conséquence est qu'on peut observer une grosse dégradation de performance en ajoutant juste un paramètre à une procédure, cela m'est arrivé...

4. ... ou dans des registres, voir la note précédente.

On va s'arrêter là dans les détails. Et on pourrait reprendre pour un langage orienté objet.

11.3.5 Sauvegarde des registres

Si on ne prend pas de précaution, `call` est une instruction très particulière : elle peut potentiellement changer les valeurs de tous les registres ! En effet, l'appelé va utiliser les registres comme bon lui semble.

De plus, dans le trip de la compilation séparée,
— l'appelant ne sait pas comment est compilé l'appelé, donc quels registres il va utiliser.
— l'appelé ne sais pas qui l'appelle, donc quel registre l'appelant est en train d'utiliser.
L'ABI doit définir également un code de bonne conduite qui peut être une des variantes suivantes :

1. C'est la responsabilité de l'appelant de sauvegarder (en les empilant) les registres dont il veut conserver la valeur après le `call`. L'appelé n'a aucun souci à se faire, il peut utiliser tous les registres comme bon lui semble.
2. C'est la responsabilité de l'appelé de sauvegarder (en les empilant) les registres qu'il va utiliser. L'appelant n'a aucun souci à se faire, toutes les valeurs des registres seront inchangées après le `call`.
3. Pareil que 2/, mais pas pour tous les registres : les registres Rx à Ry peuvent être écrasés par l'appelé (que l'appelant se le tienne pour dit). Ce sont des *scratch registers*. Par contre, les registres Rz à Rt doivent être préservés par l'appelé *(preserved registers)*. L'appelant n'a pas à se soucier de les sauvegarder lui-même.

Pour les détails : lisez l'ABI de votre système.

11.3.6 En résumé : l'enregistrement d'activation

L'espace mémoire mis en place en haut de la pile lors de l'appel d'une procédure est appelé *enregistrement d'activation* (en rosbif *activation record* : `AR`). Cet espace est désalloué lorsque la procédure se termine.

L'enregistrement d'activation comporte la place pour
— les paramètres effectifs,
— les variables locales à la procédure,
— la sauvegarde de l'environnement d'exécution de la procédure appelante (essentiellement les registres)
— l'adresse de retour de la procédure,
— et possiblement d'autres pointeurs, suivant les ABI :
  — un pointeur vers les variables globales ;
  — un pointeur sur l'`AR` de la procédure appelante (utile pour les procédures imbriquées) ;
  — pour les méthodes de langages objet, un pointeur vers l'objet (`this`), qui est lui-même une structure de donnée compliquée, avec des attributs et des pointeurs vers les surclasses pour accéder aux attributs des surclasses en respectant les règles de visibilité...)
  — ...

Il faut encore insister sur le fait que tout ceci est une convention, un contrat, un choix. Tout est toujours discutable, et les ABI évoluent avec le temps.

Par exemple, voici le code que `gcc` produit typiquement en entrée d'une procédure (`esp` est le pointeur de pile, `ebp` est l'ARP, et la pile descend dans la mémoire comme sur la figure précédente) :

```
.main:
pushl %ebp    (empiler le lien vers l'ARP de l'appelant)
movl %esp,%ebp    (l'ARP de cette procédure sera le pointeur de pile à cet instant)
subl $24,%esp    (réserver sur la pile l'espace pour les variables locales)
```

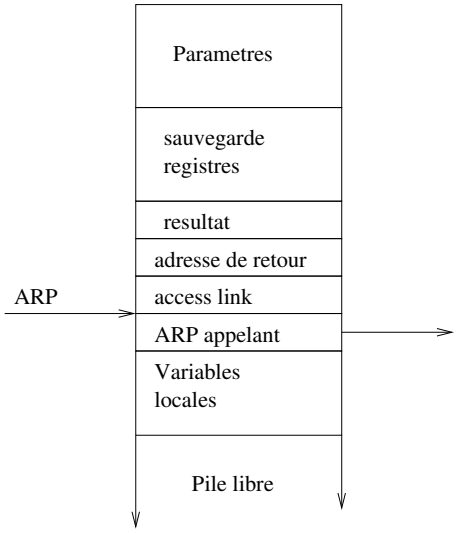


FIGURE 11.1 – Enregistrement d'activation sur la pile

11.3.7 En résumé : code à générer par le compilateur

Le code précédent est le prologue de la procédure. Chaque procédure contient un prologue et un épilogue. De même, chaque appel contient un pré-appel et un post-appel.

- **Pre-appel** Le pre-appel met en place l'AR de la procédure appelée (allocation de l'espace nécessaire et remplissage des emplacements mémoire). Cela inclut l'évaluation des paramètres effectifs, leur stockage sur la pile, l'empilement de l'adresse de retour, l'ARP de la procédure appelante, et éventuellement la réservation de l'espace pour une valeur de retour. La procédure appelante ne peut pas connaître la place nécessaire pour les variables locales de la procédure appelée.
- **Post-appel** Il y a libération des espaces alloués lors du pré-appel.
- **Prologue** Le prologue crée l'espace pour les variables locales (avec initialisation éventuelle). Si la procédure contient des références à des variables statiques, le prologue doit préparer l'adressabilité en chargeant le label approprié dans un registre.
- **Épilogue** L'épilogue libère l'espace alloué pour les variables locales, il restore l'ARP de l'appelant et saute à l'adresse de retour. Si la procédure retourne un résultat, la valeur est transférée là où elle doit être utilisée.

De plus, les registres doivent être sauvegardés, soit lors du pre-appel de l'appelant, soit lors du prologue de l'appelé.

En général, mettre des fonctionnalités dans le prologue ou l'épilogue produit du code plus compact (le pre-appel et post-appel sont écrits pour chaque appel de procédure).

11.4 Compilation des langages objets

Fondamentalement, l'orientation objet est une réorganisation de l'espace des noms du programme d'un schéma centré sur les procédures vers un schéma centré sur les données. Donc il faudrait tout reprendre... J'espère vous avoir donné les clés pour que vous compreniez la doc si vous avez un jour à la lire.

Chapitre 12

Hiérarchie mémoire

12.1 Mémoire cache

Le principe du cache est de reprendre la table 4.1 page 45, et d'y intercaler des mémoires de taille et de performance intermédiaire à deux endroits :
— entre les registres et la mémoire centrale,
— entre la mémoire centrale et le disque.
On obtient la table 12.1.

12.1.1 Principes de localité

Pourquoi un cache est utile ? Parce que

1. Une adresse mémoire actuellement accédée a toutes les chances d'être accédée à nouveau dans un futur proche (localité temporelle)
2. Une adresse proche d'une adresse mémoire actuellement accédée a toutes les chances d'être accédée aussi dans un futur proche (localité spatiale)

Exemples : les boucles (il s'agit alors d'adresses du code). Le tri d'un tableau (adresses de données). Etc. Ce principe de localité est très général. Contre-exemple : le produit de matrices si la matrice est très grande. Solution : organiser le produit de matrice par blocs. Il existe une bibliothèque, ATLAS pour Automatically Tuned Linear Algebra, qui définit la taille des blocs en fonction de la taille du cache.

12.1.2 Scratchpad versus cache

On peut laisser au programmeur/compilateur la charge de rapatrier manuellement dans le cache les données actuellement/localement utilisées. Dans ce cas on parle plutôt de *scratchpad memory*, ou mémoire brouillon. Le surcoût matériel se limite alors à la mémoire cache elle-même.

Type	temps d'accès	capacité typique
Registre	0.1 ns	1 à 128 mots (de 1 à 8 octets)
Mémoire cache	1-4ns	4Koctets – 4Mo
Mémoire vive	10 - 100 ns	4 Goctets
Cache disque	100ns	512Koctets
Disque dur	10ms	100Goctets
Archivage	1mn	(illimité)

TABLE 12.1 – Hiérarchie mémoire d'un ordinateur en 2010

Par contre il y a un surcoût de difficulté de programmation/compilation, ainsi qu'un surcoût en terme de taille de code, puisqu'il faut des instructions qui servent à rapatrier des données dans le cache.

Pour éviter cela, le plus souvent, lorsqu'on parle de cache, on parle d'un mécanisme automatique qui intercepte les requêtes mémoires, et les aiguille soit vers le cache soit vers la mémoire, en gérant au passage le remplissage du cache avec les données actuellement utilisées. C'est à ce mécanisme qu'on s'intéresse dans la suite.

Ce mécanisme sera implémenté en matériel pour les accès mémoire, et plutôt en logiciel pour les accès disques qui sont de toute manière lents. Mais les principes sont les mêmes. Nous allons nous concentrer sur le cache mémoire.

Construire ce mécanisme de gestion du cache a un coût matériel et énergétique qui s'ajoute au coût de la mémoire cache elle-même. Mais cela simplifie la vie du programmeur.

Considérant tout cela, on trouve des scratchpad sur les systèmes embarqués : on veut éviter la consommation électrique de la MMU, et par ailleurs leur code est statique et connu à l'avance. On trouve des vrais caches sur tous les systèmes un peu plus ouverts. Scratchpad versus cache, c'est un peu la même discussion que VLIW versus superscalaire.

12.1.3 Cache hit et cache miss

Et donc on va garder dans le caches les adresses (et les données correspondantes) accédées récemment. Un accès mémoire commencera par comparer l'adresse demandée avec celles qui sont présentes dans le cache. En cas de succès (souvent) la donnée sera lue depuis le cache *(cache hit)*. En cas d'échec (défaut de cache ou *cache miss*, rarement espère-t-on), la donnée sera d'abord chargée de la mémoire vers le cache. Elle y sera présente pour cet accès et aussi pour les accès suivants (localité temporelle). Et au passage, le cache chargera aussi les quelques données suivantes pour exploiter la localité spatiale.

Concrètement, le cache charge depuis la mémoire centrale une *ligne de cache* à la fois. La ligne de cache est l'unité d'échange entre le cache et la mémoire, et fait quelques centaines d'octets. Une ligne de cache de taille 2^l est définie par "toutes les adresses ayant tous leurs bits identiques sauf les l de poids faible".

Une fois définie la ligne de cache, et si on a les moyens, on peut organiser notre mémoire physique pour qu'elle soit adressée par ligne uniquement. La mémoire physique sera alors construite pour réagir à des adresses sans les l derniers bits et envoyer 2^l octets en parallèle sur un bus très large entre le processeur et la mémoire. On verra cela en 12.1.7.

12.1.4 Hiérarchie mémoire

Une mémoire cache se comporte, vu de l'extérieur, comme une RAM normale, mais réagit plus rapidement. Par exemple, le cache disque est à peine plus lent que la mémoire vive – un peu tout de même car les bus qui y accèdent sont moins larges que ceux qui accèdent à la mémoire vive.

Il peut en fait y avoir plusieurs niveaux de mémoire cache :

— le premier sur la puce même du processeur, au plus près de chaque cœur ;
— le second toujours sur la puce, mais partagé entre plusieurs cœurs, donc un peu plus loin de chacun ;
— le troisième sur une puce séparée, mais avec un bus très large (128-256 bits) avec le processeur.

De même, il peut y avoir un cache disque dans le disque dur, et un autre dans la mémoire centrale. Dans la suite, on ne va considérer qu'un niveau car les mécanismes sont les mêmes lorsqu'il y en a plusieurs.

12.1.5 Construction d'un cache

A chaque ligne de cache est associée dans le cache son adresse, ainsi qu'un bit de validité (par exemple, toutes les lignes du cache sont invalides lorsqu'on allume l'ordinateur). On voit que d'organiser le cache en lignes (et pas en mots) ne sert pas qu'à exploiter la localité spatiale : cela sert aussi à minimiser le surcoût de ces données supplémentaires. Pour un cache organisé par mots, il faudrait autant de bits pour les données que pour leur adresse...

Lorsque le processeur envoie une demande de lecture à une adresse donnée, il faut savoir si cette adresse est présente dans le cache. Pour cela, le cache devrait comparer l'adresse recue (sans ses l derniers bits) avec toutes les adresses présentes dans le cache. C'est coûteux, donc on fait des compromis :

— Cache *Direct-mapped :* une ligne ne peut aller qu'à un seul endroit dans le cache, défini par les bits de poids faible de l'adresse de la ligne. Simple à réaliser, mais conflits nombreux.

— Cache *Fully associative :* une ligne peut aller n'importe ou dans le cache. Le cache est alors une mémoire associative, comme un dictionnaire dans lequel l'adresse est le mot cherché, et la donnée est la définition. Cela implique que le cache doit comparer une adresse d'entrée avec toutes ses adresses, et en parallèle pour que ce soit rapide : c'est coûteux en matériel.

— Cache *n-way set associative* : c'est un compromis intermédiaire. Une ligne peut aller dans le cache dans un ensemble de taille $n = 2$, 4 ou 8 lignes. Il faut comparer l'adresse de la ligne avec les adresses juste dans cet ensemble.

On a fait des statistiques, du genre ***2 :1 Cache Rule :*** *The miss rate of a direct-mapped cache of size N is about the same as that of a two-way set-associative cache of size $N/2$.* (Hennessy-Patterson, CAQA 1ère édition, page 0)

Éviction du cache

Un problème se pose lorsque le cache est plein, ce qui arrive assez vite puisque sa capacité est réduite. Pour ajouter une nouvelle ligne, il faut en virer une. Laquelle ?

En correspondance directe on n'a pas le choix.

Dans les autres cas, en principe on aimerait virer la moins utile dans le futur, mais... on ne connaît pas le futur. Alors on le prédit en fonction du passé : en première approximation, la ligne à virer est la moins utilisée dans le passé (*least recently used* ou LRU).

Mais pour cela, il faut accrocher une date à chaque ligne. Encore des frais. Mais pas tant que cela : tout ce qu'on veut savoir, c'est qui est le plus récent. Par exemple, pour un cache à 2 voies il suffit d'un bit par ligne : à chaque accès on met à 1 le bit de la ligne accédé et à 0 le bit de l'autre. Pour les caches à plus de 2 voies il faut plus de bits et il faudra les comparer tous, donc c'est compliqué. On re-approxime, par exemple on fait deux paquets de 2 voies et on a un bit qui dit quel paquet a été LRU, et dans chaque paquet on a un bit LRU comme pour le cache à deux voies.

Une technique qui marche bien aussi est de remplacer une ligne au hasard...

Écriture dans un cache

On n'a parlé que de lectures, pas encore des écritures mémoire. Convainquons-nous d'abord qu'elles sont bien moins fréquentes que les lectures. Déjà, le code est surtout lu. Et pour les données, considérez le produit de matrice de service : on fait n^2 écritures pour n^3 lectures.

Deux stratégies en cas d'écriture :

— écriture directe dans la mémoire centrale *(write through)* : lent à chaque écriture, puisqu'on doit attendre d'avoir fini l'écriture dans la mémoire centrale. Par contre le cache est toujours *cohérent* avec la mémoire centrale, donc on peut virer une ligne sans plus de travail

— écriture différée *(write back)* : on n'écrit que dans le cache, ce qui est rapide. Mais le cache devient incohérent avec la mémoire centrale. Lorsqu'on vire une ligne, il faut d'abord la recopier en mémoire centrale.

Ces questions de *cohérence* deviennent cruciales lorsqu'on a un ordinateur multiprocesseur (ou multicœur, c'est plus la mode) à mémoire partagée. Dans ce cas il faut que tous les caches soient cohérents entre eux. Ce qui veut dire que toute écriture à l'adresse a doit invalider toutes les lignes de caches contenant a dans tous les autres processeurs. Les protocoles qui assurent cette cohérence deviennent vite subtils, et il y a souvent des bugs de ce type dans le révisions initiales des processeurs. Heureusement, contrairement au bug de la division du pentium, ils sont cachés à l'utilisateur par le système.

Cache unifié ou cache séparé

Il peut être utile d'avoir un cache *séparé programmes/données*, ce qui a l'avantage supplémentaire d'éviter le conflit sur le bus mémoire entre lecture d'instruction et lecture de donnée. Toutefois, il risque de ne pas être rempli aussi optimalement qu'un cache *unifié*, par exemple dans le cas d'un petit nid de boucle traitant beaucoup de données – encore mon produit de matrices. On voit souvent des caches séparés pour le premier niveau de cache. Et des caches unifiés pour les niveaux suivants.

Attention toutefois, les deux caches séparés doivent tout de même être cohérents.

En résumé

En résumé, il y a un paquet de paramètres à considérer pour un cache :
— unifié ou séparé
— taille de la ligne de cache
— taille du cache
— associativité
— politique de remplacement
— write-through ou write-back
et ceci, pour chaque niveau de la hiérarchie mémoire (combien en faut-il ? Encore un paramètre).

On gère ces paramètre en simulant des processeurs exécutant des benchmark. Au final, les hiérarchies actuelles assurent un *miss rate* inférieur à 1% en moyenne.

Petit dessin de l'architecture d'un cache à deux voies, avec 2K voies de 2 lignes de 4 mots.

12.1.6 Statistiques et optimisation des caches (*)

Il faut faire des statistiques sur les accès mémoire pour définir les paramètres du cache.

Ces statistiques dépendent des programmes, et diffèrent selon qu'on parle de code ou de données.

Ensuite on combine ces statistiques à coups de loi d'Amdhal (calculs p. 564 du Patterson/-Hennessy). On constate en particulier que la performance d'un cache peut se dégrader rapidement si on augmente juste la fréquence du processeur sans toucher à la hiérarchie mémoire.

12.1.7 Entre le cache et la mémoire physique (*)

Si on résume, le cache doit, en cas de *miss*, rapatrier toute une ligne de la mémoire physique. Les options sont (de gauche à droite sur la figure 12.1)

— d'avoir un bus de la largeur de la ligne de cache (ici 8×32 bits) : on rappatrie une ligne en une latence mémoire, mais c'est coûteux en filasse.
— d'avoir un bus de la largeur d'un mot, et un automate qui génère les adresses consécutives. On rappatrie une ligne en 8 latences mémoires dans cet exemple.
— de pipeliner les lectures mémoires sur ce bus : on envoie toutes les requêtes en lecture, puis on recoit en rafale toutes les données. Ainsi on rappatrie une ligne en 8+latence mémoire cycles. Cette solution présente un bon rapport performance/coût. Le truc est d'utiliser des mémoires *entrelacées*, qui vont répondre chacun à son tour, un mot par cycle.

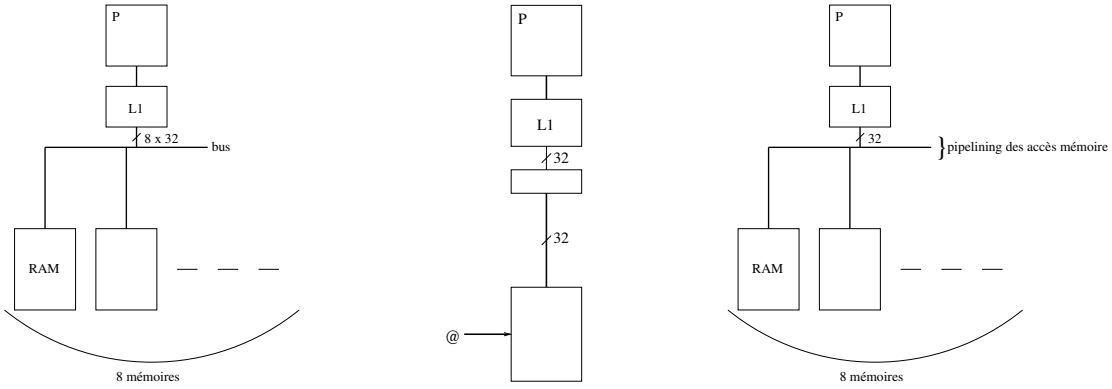


FIGURE 12.1 – Entre cache et mémoire physique

12.1.8 Les caches dans un multicœur à mémoire partagée (*)

On avait déjà un goulot d'étranglement au niveau de l'interface mémoire, cela ne va pas s'arranger si on a plusieurs cœurs qui se battent pour la mémoire. La solution est bien sûr de donner un cache à chaque cœur. La grosse question est alors la *cohérence* des différents caches. Si plusieurs cœurs travaillent sur la même adresse et que l'un l'écrit, il faut propager cette écriture aux autres avant leur prochaine lecture.

Voici un apercu d'une technique qu'on peut mettre en œuvre dans ce cas : le protocole MESI. On associe, à chaque ligne de cache dans chaque cache, un état qui peut être l'un des suivants :

— I pour Invalide : un accès à une adresse de cette ligne provoquera un défaut de cache (cache miss) pour faire remonter la donnée de la mémoire centrale.
— S pour Partagé *(Shared)*. Le cœur a accès en lecture mais pas en écriture, car d'autres cœurs ont aussi cette ligne dans leur cache, également en lecture seulement pour l'instant.
— E pour pour Exclusif : Le cœur a accès en lecture et écriture, parce qu'aucun autre cœur n'a cette ligne dans son cache. Mais il n'a pas encore écrit dedans : la ligne est cohérente avec la mémoire, on peut la virer du cache sans autre forme de procès.
— M pour Modifié : pareil qu'Exclusif, mais il y a eu des écritures dans la ligne, et elle n'est plus cohérente avec la mémoire centrale. Avant de la virer du cache il faudra l'écrire en mémoire *(write back)*.

Il n'y a plus qu'à construire l'automate associé à ces 4 états. Les transitions sont provoquées par les instructions de lecture/écriture mémoire des différents cœurs. Vous le trouverez dans tous les bons bouquins. Le matériel responsable de la cohérence des caches sera chargé d'implémenter cet automate.

Il y a plein de communications cachées dans ce protocole. Par exemple, lorsqu'un cœur veut écrire une donnée à une ligne qui est dans l'état S, il doit d'abord lancer une demande d'invalidation de cette ligne à tous ses copains. Il n'est pas question de réaliser l'écriture avant d'être certain que toutes ces invalidations ont été effectuées, et donc que l'écriture ne va pas créer une incohérence.

12.2 Mémoire virtuelle

Dans cette section, on va voir une des vraies différences entre un microcontrôleur (comme le MSP430) et un ordinateur. Le second est conçu pour pouvoir gérer plusieurs processus, et offre un certains nombre de mécanismes matériels pour cela. La mémoire virtuelle est l'un de ces mécanismes.

Le principe est simple : les adresses manipulées par les programmes sont des adresses *virtuelles*, qui ne correspondent pas du tout aux adresses *physiques*. La traduction de l'une en l'autre est réalisée automatiquement (en matériel), et dépend du processus.

12.2.1 Vue générale

Le processeur possède un registre "numéro du processus en cours" ou PID qui contient le numéro du processus courant, mettons sur 16 bits. Ce registre est mis à jour à chaque changement de processus (on dit aussi *changement de contexte*).

L'adresse virtuelle (sur 32 bits) générée par une instruction est d'abord étendue par le PID. On obtient une adresse virtuelle unique de 48 bits. Cette adresse virtuelle unique est ensuite traduite en une adresse physique (de 28 bits si vous êtes pauvre, de 36 si vous êtes très riche) qui est envoyée sur le bus physique de la mémoire.

On voit que par ce mécanisme, on peut avoir deux processus qui exécutent un même code, produisant les mêmes adresses absolues, sans se marcher sur les pieds. Par exemple, il est typique d'organiser la mémoire ainsi : le code en bas, suivi par le tas qui croît en montant, et la pile descendant de l'adresse maximale $2^{32} - 1$. Tous les processus auront leur pile partant de l'adresse (virtuelle) maximale, mais ce ne sera pas la même pile.

A ce point un petit dessin à deux processus s'impose. A gauche, les espaces d'adressages virtuels. A droite, l'espace d'adressage physique.

Remarque : si vous avez un processeur 64 bits (genre AMD64), cela signifie que ses adresses virtuelles sont de 64 bits, et donc que chaque processus peut adresser jusqu'à 2^{64} octets. Toutefois, par le même mécanisme il pourra vivre dans une mémoire physique plus petite, et tout de même avoir la pile qui descend de l'adresse $2^{64} - 1$.

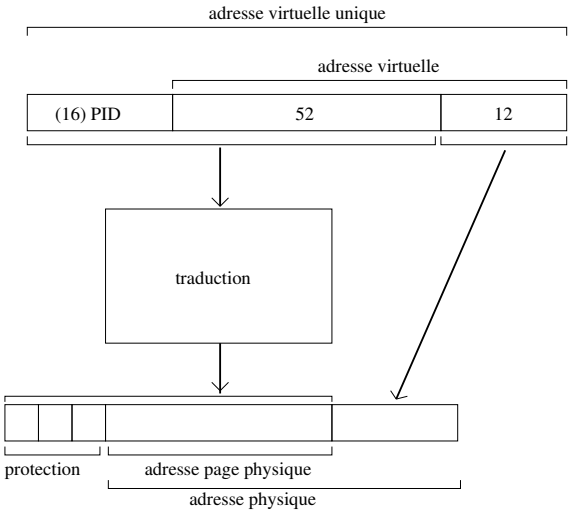


FIGURE 12.2 – Vue d'ensemble de l'adressage à travers la mémoire virtuelle pour des adresses virtuelles de 64 bits

12.2.2 Avantages de la mémoire virtuelle

Puisque nous allons étendre les adresses virtuelles avec le PID, on peut en profiter pour établir un système d'allocation et de protection de la mémoire. Et puisque les adresses manipulées par le programme ne sont plus des adresses physiques, on peut envisager de modifier l'emplacement des adresses physiques pour une adresse virtuelle donnée suivant les besoins.

Voici un résumé de ce que la mémoire virtuelle va permettre :

1. Donner à chaque processus l'impression qu'il est tout seul dans l'ordinateur.
   — impossibilité mécanique pour un processus d'écrire dans la mémoire des autres processus (isolation mémoire)
   — possibilité pour chaque processus d'allouer toute la mémoire physique (pas tous en même temps bien sûr).

2. Proposer des mécanismes de protection de la mémoire plus fins que l'isolation : par exemple, certaines zones de la mémoire, contenant par exemple le code de `printf`, sont *partagées*, accessibles en lecture par tous mais en écriture uniquement par certains processus systèmes.
3. Donner l'illusion qu'il y a plus de mémoire que la mémoire physique réellement disponible
   — les adresses (virtuelles) qui ne sont pas actuellement activement utilisées par l'ordinateur ne seront pas traduite en mémoire physique, mais en mémoire disque. La partie du disque correspondant s'appelle alors *mémoire d'échange* ou *swap* en swahili.
   — L'OS remontera ces adresses en mémoire physique automatiquement quand le besoin s'en fera sentir.
   — rappel de la LCDE : le Mo sur le disque est bien moins cher, mais bien plus lent que le Mo en mémoire vive.

L'idée d'un registre de PID n'est qu'une idée parmi d'autres. Par exemple, le powerPC n'a pas de registre de PID, mais a à la place 16 "registres de segments" qui font l'expansion des 4 bits de poids forts de l'adresse virtuelle en une adresse de page virtuelle sur 24 bits. Ce sont ces 16 registres qui sont changés par l'OS à chaque changement de processus. Lorsque certains de ces registres contiennent des valeurs identiques pour deux processus, la mémoire correspondante est partagée par les deux processus.

12.2.3 Aspects architecturaux

La traduction d'une adresse virtuelle en adressse physique peut se faire en principe par une lecture de table. Deux problèmes pratiques, que nous exposons dans le cas d'adresses étendues par le PID :

1. La table de traduction a 2^{48} entrées, donc est plus grosse que la mémoire physique.
2. Chaque accès mémoire se traduit désormais par deux accès mémoire...

La solution au premier problème va être de faire cette traduction par *pages mémoire*, une page étant une unité de mémoire intermédiaire, typiquement 4 Ko. Ainsi, le coût de la traduction sera amorti sur toutes les adresses d'une page.

La solution au second problème est d'utiliser un mécanisme de cache : les traductions récentes sont conservées dans une petite mémoire associative, appelée TLB pour *translation look-aside buffer*. En pratique, cette TLB réalise la traduction d'adresse très rapidement la plupart du temps. Lorsqu'elle échoue, le système ou le matériel doit réaliser la traduction pour de bon, ce qui peut impliquer plusieurs accès mémoire, mais c'est très rare : moins d'une adresse sur 4000 pour une page de 4Ko, espère-t-on. Rappelons que le principe de localité s'applique toujours : les adresses d'une page ont toutes les chances d'être réutilisées de nombreuses fois, ce qui amortit encore plus le coût.

12.2.4 Dans le détail : table des pages (*)

Tables des pages directe, organisée hiérarchiquement (Fig.2 de l'article *Virtual memory, issues of implementation* du magazine *Computer*).

Accès top-down ou bottom-up.

Pb : adresses virtuelles de 64 bits ?

Solution : table des pages inversées (Fig. 5).

12.2.5 Cache d'adresses virtuelles ou cache d'adresses physiques ? (*)

Au fait, de quelles adresses on parle ?

Le principe de localité fonctionne aussi bien en adresses virtuelles qu'en adresses physiques, dès lors que la traduction se fait par pages plus grosses que la ligne de cache. Si l'on choisit un

cache en adresses virtuelles, il faut naturellement qu'il utilise les adresses virtuelles étendues par le PID, sans quoi l'isolation des processus n'est plus garantie. Quoiqu'une alternative est que l'OS marque tout le cache comme invalide à chaque changement de processus.

Il est plus naturel d'avoir un cache en adresses physiques. L'inconvénient est qu'il faut alors réaliser la traduction virtuelle-physique, puis l'accès au cache en séquence.

Avec un cache en adresses virtuelles, on économise cette traduction en cas de hit, ce qui permet de retourner la donnée plus vite. L'inconvénient est que le code de `printf`, et en général toute donnée partagée, va se retrouver en plusieurs exemplaires dans le cache. On a du gâchis de cette mémoire coûteuse.

Un bon compromis est sans doute de réaliser le cache de niveau 1 (le plus près du processeur, et celui qui doit être le plus rapide) en adresses virtuelles, et les niveaux suivants en adresses physiques.

On va construire complètemement la version d'une hiérarchie mémoire qui a besoin du moins de matériel. Puis on verra ce qui existe dans les "vrais" processeurs grâce à un article paru dans *IEEE Computer*.

12.3 Une mémoire virtuelle + cache minimale

— Un registre de PID
— Un cache de niveau 1 séparé, indexé virtuellement, et tant pis pour les doublons.
— En cas de *hit*, l'accès mémoire ne pose pas de pb.
— Un défaut de cache génère une interruption. Le système consulte ses tables, fait la traduction de la ligne virtuelle en ligne physique, vérifie les droits, choisit une ligne de cache à virer, et copie la ligne physique dans le cache. Le tout idéalement en matériel.
— Les niveaux suivants de la hiérarchie sont indexés physiquement.

12.3.1 Instructions spécifiques à la gestion mémoire

Et voici quelques instructions à ajouter à notre processeur. Sauf la première, ce sont des instructions accessibles uniquement en mode superviseur du processeur.

— prefetch (pour la performance) : cette instruction demande à ce qu'une adresse soit présente dans le cache.
— lecture et écriture directement en mémoire physique, pour y mettre par exemple la page des tables
— invalidation d'une ligne de cache ou d'une page
— éventuellement écriture dans les tables matérielles si elles existent
— etc...

Chapitre 13

Conclusion : vers le système d'exploitation (*)

L'OS a deux rôles fondamentaux (et largement interdépendants) :

— *virtualiser* le matériel : donner à l'utilisateur l'illusion qu'il est tout seul sur un ordinateur ; donner au programme l'illusion qu'il s'exécute sur un matériel "mieux" (plus simple, moins limité) que le vrai.

— *gérer les ressources* (les différents niveaux de mémoire, le temps de calcul, les processeurs, les entrées/sorties,...) pour les partager au mieux entre plusieurs utilisateurs/applications.

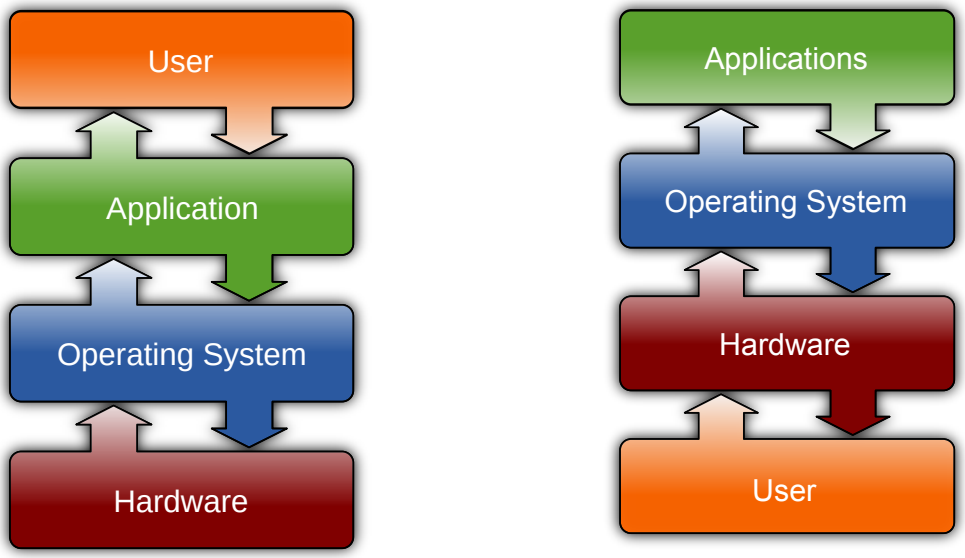


FIGURE 13.1 – La place de l'OS selon Wikipedia (à gauche) et G. Salagnac (à droite). Du point de vue de l'architecte, c'est celle de droite qui correspond le mieux à la réalité.

13.1 Le rôle de l'OS (du point de vue d'un prof d'archi)

Du plus fondamental au plus gadgetesque :

— Définir (ou tout au moins publier en fournissant une interface) des règles du jeu permettant à un programme de fonctionner sur une machine donnée. Par exemple, comment demander (allouer) de la mémoire, comment les procédures se passent des paramètres, où se trouve la pile, comment afficher un caractère à l'écran, comment créer un fichier disque, comment lancer un programme...
— Assurer les tâches de maintenance de routine : compacter la mémoire lorsqu'elle est libérée, envoyer les bonnes couleurs sur l'écran et les bons échantillons sonores dans le tampon de la carte son au bon moment,...
— Démarrer les sous-systèmes dans l'ordre.
— gérer les *processus*, et on va commencer par voir cela.
— Fournir diférentes abstractions du matériel et du logiciel, certaines dégénérant franchement en allégories. Exemples : de l'arborescence du système de fichiers jusqu'au bureau avec des fenêtres et du *drag and drop* entre elles.

13.2 Multiutilisateur ou multitâche, en tout cas multiprocessus

On dit processus, ou tâche.
Les processus doivent se partager
— la mémoire de l'ordinateur (l'espace) comme vu au chapitre 12 ;
— le temps ;
— d'autres ressources, notamment des entrées/sorties, et depuis peu les ressources d'exécution : les cœurs de vos processeurs multicœurs de gamerz.
Le rôle de l'OS est d'assurer ce partage sans disputes et dans la bonne humeur.

13.2.1 Partage du temps

Pour partager le temps, on le découpe en tranches de quelques milliers de cycles, et chaque processus exécute une tranche de calcul a son tour.

Entre deux processus, le processeur doit *changer de contexte*. Un contexte c'est au moins l'état de tous les registres du processeur. Il sera stocké par le système en mémoire en attendant le prochain tour.

La tranche de temps doit être
— assez courte pour que l'utilisateur, qui fonctionne à 100Hz, ait l'impression que les processus s'exécutent en parallèle,
— mais assez longue pour que le temps du changement de contexte reste négligeable devant le temps passé à faire le boulot.

Un processeur de 1GHz qui qui veut changer de processus à 100Hz peut réaliser 10 millions de cycles de chaque processus entre deux changements de contextes... c'est beaucoup.

Le partage du temps peut être coopératif (chaque processus est responsable de rendre la main au système après le temps qu'il juge raisonnable) ou préemptif (le système alloue les tranches de temps et interrompt les autres processus). Les premiers OS multitâches grand public étaient souvent coopératifs :
— c'est plus simple pour l'auteur du système (c'est essentiellemnt aux auteurs des applications de gérer le multitâche)
— à une époque où les ordinateurs étaient moins rapide, cela permettait relativement facilement de rendre le système réactif aux interactions avec l'utilisateur : par exemple, lorsque celui-ci déplace une fenêtre, le processus qui s'occupe de cela va décider de garder la main jusqu'à ce que l'utilisateur lache la souris. Ainsi, 100% du temps CPU est consacré à assurer la fluidité du déplacement de la fenêtre.
— enfin cela n'a besoin d'aucun support matériel particulier.

Le gros inconvénient du multitâche coopératif est qu'un plantage d'un processus peut planter tout le système. De nos jours, on n'a plus que du multitâche préemptif.

Celui-ci a besoin d'un support matériel minimal, sous formes au moins d'une interruption, typiquement déclenchée par une horloge extérieure au processeur tous les 100ème de seconde, qui interrompt le processus courant et saute à la routine système qui va endormir ce processus (sauvegarder son contexte) et réveiller le processus dont c'est le tour.

13.2.2 Partage des entrées/sorties

La plupart des processus ne font rien la plupart du temps : ils attendent que l'utilisateur bouge sa souris, son clavier, ou qu'un paquet arrive sur le réseau...

Pour implémenter une telle attente, on peut écrire une boucle infinie qui teste l'arrivée d'un événement. C'est ce que vous ferez sur votre processeur. Toutefois, cette approche, dite *attente active*, est un gros gaspillage de ressources de calcul : si par exemple le système a un gros calcul à réaliser en tâche de fond (par exemple une impression, qui consiste à convertir un document en une image à la résolution, énorme, de l'imprimante), et que personne n'appuie sur une touche du traitement de texte, on préfère donner plus de temps de calcul à l'impression, et moins à l'attente de la touche.

La bonne approche est d'endormir les processus qui attendent un événement (c'est à dire de ne plus leur donner de tranche de temps du tout). À l'arrivée d'un événement, le système réveille le ou les processus qui attendent cet événement.

Autrement dit, le résultat principal de l'appel à `scanf` ou `getc` est d'endormir votre processus.

Ainsi, l'attente active est remplacée par l'attente passive, et c'est ainsi que votre *CPU load* est à 0 quand votre PC ne fait rien, ou que l'impression d'un document est capable d'accaparer 99% du temps CPU, le 1% restant étant ce dont votre PC a besoin pour gérer un étudiant tapant du texte le plus vite possible de ses petites mains à 3Hz.

En fait, même les processus systèmes responsables des entrées/sorties (ceux qui vont réveiller vos processus à vous) sont eux-même en attente passive. Toutefois, eux sont réveillés par des interruptions matérielles : appui sur une touche, la carte son réclame des échantillons à jouer, *timer* à une fréquence de l'ordre de la miliseconde, etc.

Certains processeurs peuvent même arrêter complètement leur horloge jusqu'à la prochaîne interruption matérielle. Je ne sais pas ce qui se passe dans votre PC.

13.2.3 Partage des ressources d'exécution

On a à résoudre un problème d'équilibrage des charges, ou *load balancing* en australien. Si on vous demande de le faire, vous implémenterez un algorithme glouton qui fera bien l'affaire : quand arrive un nouveau processus, on le refile au processeur qui a l'air le moins chargé.

Si par la suite tous les processus du cœur 0 se terminent ou s'endorment alors que le processeur 1 est toujours à 100%, on peut envisager de redéployer certains des processus du second sur le premier. Un tel rééquilibrage de charge a un coût très supérieur à un changement de contexte, donc il faut le faire avec discernement. La difficulté est de prévoir l'avenir : quelle est la probabilité que tel processus endormi se réveille, et pour combien de temps ? L'OS peut maintenir des statistiques par processus.

Troisième partie

Annexes

Annexe A

Rudiments de complexité

Il est utile, en architecture comme en programmation, de raisonner à la louche sur le coût (en temps, en mémoire, en surface, ...) d'une implémentation. Mais ce coût dépend en général de la taille du problème. Nous donnons ici des éléments qui permettent, toujours à la louche, d'évaluer comment un coût croît avec la taille du problème. Ce n'est pas toujours juste proportionnel !

Dans toute la suite, n décrira la taille du problème.

Les fonctions utiles à ce cours sont les fonctions suivantes. Quand n est grand,

$$1 \ << \ log_2 n \ << \ n \ << \ n^2 \ << \ n^3 \ << \ ... \ << \ 2^n \ << \ n^n \ < \ n! \ .$$

Là dedans, vous connaissez et visualisez bien n et n^2. Il est important de bien se familiariser avec 2^n et $\log_2 n$. Non pas comme des objets abstraits, mais comme des objets qui arrivent naturellement en architecture.

On va voir que 2^n, c'est déjà très cher. Par conséquent, les fonctions plus grande comme n^n et $n!$ seront de peu d'intérêt pratique.

A.1 Les fonctions 2^n et $\log_2 n$

La fonction 2^n apparaît dès le chapitre du codage. Dans ce cas, il suffit de comprendre que le 2 de 2^n est le 2 de binaire.

— Sur n bits on peut coder 2^n informations différentes.

— Autrement dit, un mot de n bits peut prendre 2^n valeurs différentes.

— La table de vérité d'une fonction combinatoire à n entrées binaires a 2^n lignes.

Sa propriété fondamentale est la suivante :

$$2^{a+b} = 2^a \times 2^b.$$

Mais en fait on utilise surtout la variante suivante (pour $b = 1$) :

$$2^{a+1} = 2 \times 2^a.$$

La fonction 2^n apparaît donc naturellement chaque fois qu'on a une construction algorithmique du type : *pour obtenir truc*$(n+1)$*, j'assemble deux truc*(n). En effet, le coût de truc$(n+1)$ sera le double du coût de truc(n). Cela suggère que c e coût sera en 2^n.

Une autre propriété utile, car elle sous-tend toute l'arithmétique binaire :

$$2^n - 1 = \sum_{i=0}^{n-1} 2^i.$$

Vérifiez pour les petits n :
$1 + 2 \ = \ 3 \ = \ 4 - 1$

$1+2+4 = 7 = 8-1$
$1+2+4+8 = 15 = 16-1$.

En fait c'est aussi une propriété que vous manipulez, dans sa version décimale, depuis que vous savez compter : c'est $999+1 = 1000$. Vous avez donc déjà rencontré cette dernière propriété sous la forme suivante : en binaire, $10000000 = 01111111 + 1$. Réfléchissez une minute pour vous convaincre que c'est la même chose. Sinon, la preuve est par récurrence en utilisant $2^{a+1} = 2 \times 2^a$.

Enfin, la fonction 2^n croît très vite. Cela se voit dans les deux propriétés ci-dessus.

$2^0 = 1$, $2^1 = 2$, $2^{16} = 65536$, $2^{32} \approx 4.10^9$ (4Giga), $2^{64} \approx 16.10^{18}$. Peut-être plus parlant : $2^{64} \approx 16\,000\,000\,000\,000\,000\,000$.

Quant à la fonction $\log_2 n$, c'est l'inverse de 2^n :

$$x = 2^y \quad \Longleftrightarrow \quad y = \log_2 x \quad .$$

On peut donc reformuler à l'envers toutes les phrases dans lesquelles on avait un 2^n, par exemple : *pour coder n informations différentes, il faut $log_2 n$ bits*.

Attention, si n n'est pas une puissance de 2, $\log_2 n$ ne sera pas entier. Il faut l'arrondir à l'entier supérieur. La phrase correcte serait *pour coder n informations différentes, il faut $\lceil log_2 n \rceil$ bits*. Mais ce sera souvent implicite, surtout dans les raisonnements à la louche.

A.2 Raisonner à la louche

Quand on évalue les coûts, on ajoute souvent les coûts de plusieurs composants. Mais quand on raisonne à la louche, on ne s'intéresse qu'au plus important. Les autres coûts sont rassemblés dans une formulation de type "et des chouillas".

Tout cela peut s'appuyer sur des mathématiques propres. Si vous voulez creuser, allez wikipédier *comparaison asymptotique* et *complexité asymptotique*.

Par exemple, quand *pour obtenir truc$(n+1)$, j'assemble deux truc(n)*, j'ai souvent en fait besoin d'un chouilla en plus. Pour obtenir un multiplexeur pour des adresses n bits, on assemble deux multiplexeurs pour des adresses de $n-1$ bits, et un multiplexeur de taille 1. Ce dernier est un chouilla. On obtient quand même un coût total en 2^n.

Quand on a un doute on peut toujours faire une vraie récurrence où l'on voit apparaître une des formules précédentes. On peut partir d'un schéma pour un petit n. Par exemple, la figure 3.7, p. 30, fait apparaître la récurrence $c_{n+1} = 2c_n + 1$, avec $c_1 = 1$ (ici a est le nombre de bits d'adresse). Calculez les premiers termes (1, 3, 7, 15, 31) et intuitez-en que $c_n = 2^n - 1$, ce que vous démontrerez ensuite en utilisant la propriété déjà vue $2^{a+1} = 2 \times 2^a$.

En général, si on a une somme de deux fonctions de la liste ci-dessus, à la louche on peut négliger la plus petite. Je dirai donc que le coût de la gare de triage de la p. 30 est en 2^n. Quel est le coût en fonction du nombre de cases adressées k ? On voit qu'il faut $1 + 2 + 4 + 8 + ... + k/2$ aiguillages, donc $k - 1$.

Pour des raisons similaires, on ne s'intéresse pas aux constantes qui peuvent multiplier ces fonctions. Si je compare une solution qui coûte 2^n et une solution qui coûte n^2, je ne suis pas à un facteur 3 près sur ces coûts : le n^2 peut même être un $17n^2$, dès que n sera assez grand ce sera moins cher que 2^n.

Si les constantes sont importantes pour vous, c'est que vous ne raisonnez plus à la louche — je ne dis pas que c'est mal.

Pour conclure, les fonctions listées ci-dessus ont également les propriétés à la louche suivantes :

- Si le coût est en $\log_2 n$, il faut que le problème double de taille pour que le coût commence à augmenter. C'est donc pas cher. C'est bien.
- Si le coût est en n, il est proportionnel à la taille du problème.
- Si le coût est en 2^n, l'augmentation d'une unité de la taille du problème fait doubler le coût. On n'ira pas très loin.