ASR1 2006
Protopoly

Florent de Dinechin
avec des figures de
R. Bergasse,
A. Derouet-Jourdan,
A. Friggeri,
B. Grenet,
F. Givors,
C. Keller,
E. Lassalle,
P.E. Meunier,
O. Schwander,
P. Vannier

18 décembre 2006

Table des matières

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 de 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.

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 sont les précurseurs de l'internet en s'envoyant à distance du texte codé par un code à 5 bits.

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).

C'est quoi un ordinateur moderne ? C'est cela :

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

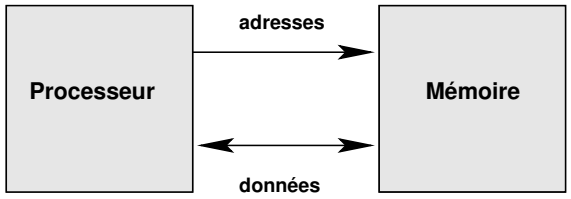

FIG. 1.1 – S'il te plait, dessine-moi un ordinateur

- 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 ar 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 :

- $\frac{\text{Masse de l'univers}}{\text{Masse du proton}} \approx 10^{78}$
  (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.
  (Actuellement, un fil ou un transistor font quelques dizaines d'atomes de large...)

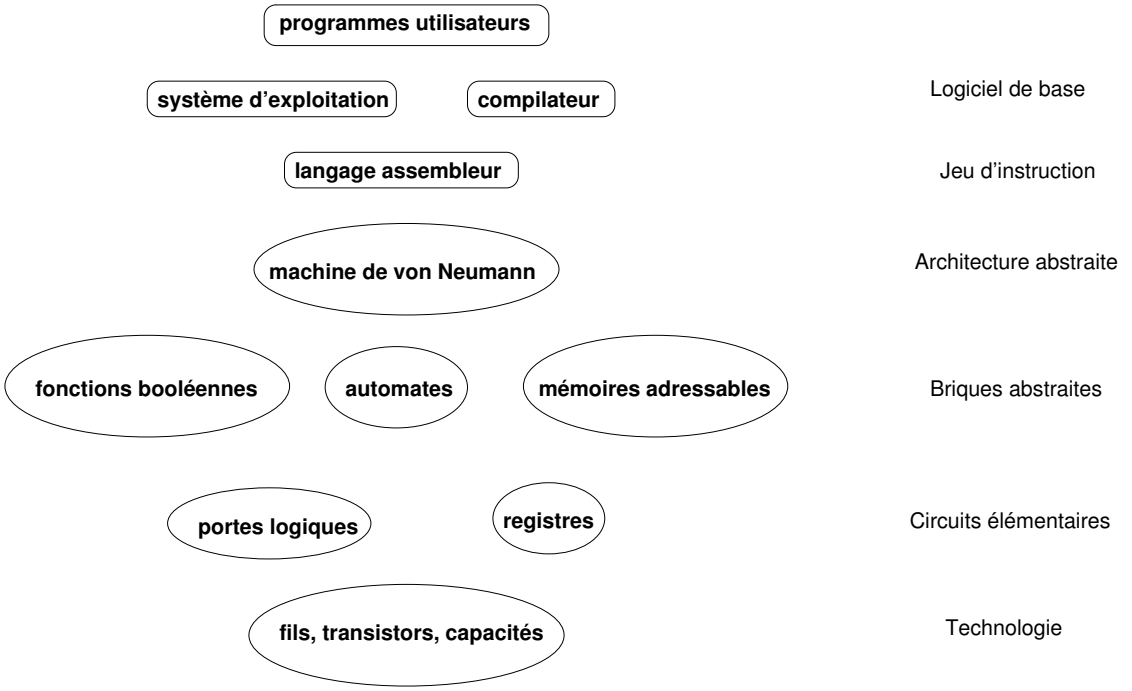

FIG. 1.2 – *Top-down* ou *bottom-up* ?

- 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 de gérer des fréquences au Hz près dans les 600MHz

1.4.2 La technologie en 2006

Les ordinateurs actuels sont construits en assemblant des transistors (plein). 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 Intel dans les années 60 et jamais démentie depuis, qui décrit l'augmentation exponentielle de la quantité de transistors par puce (la taille de la puce restant à peu près constante, de l'ordre de 1cm^2).

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. 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

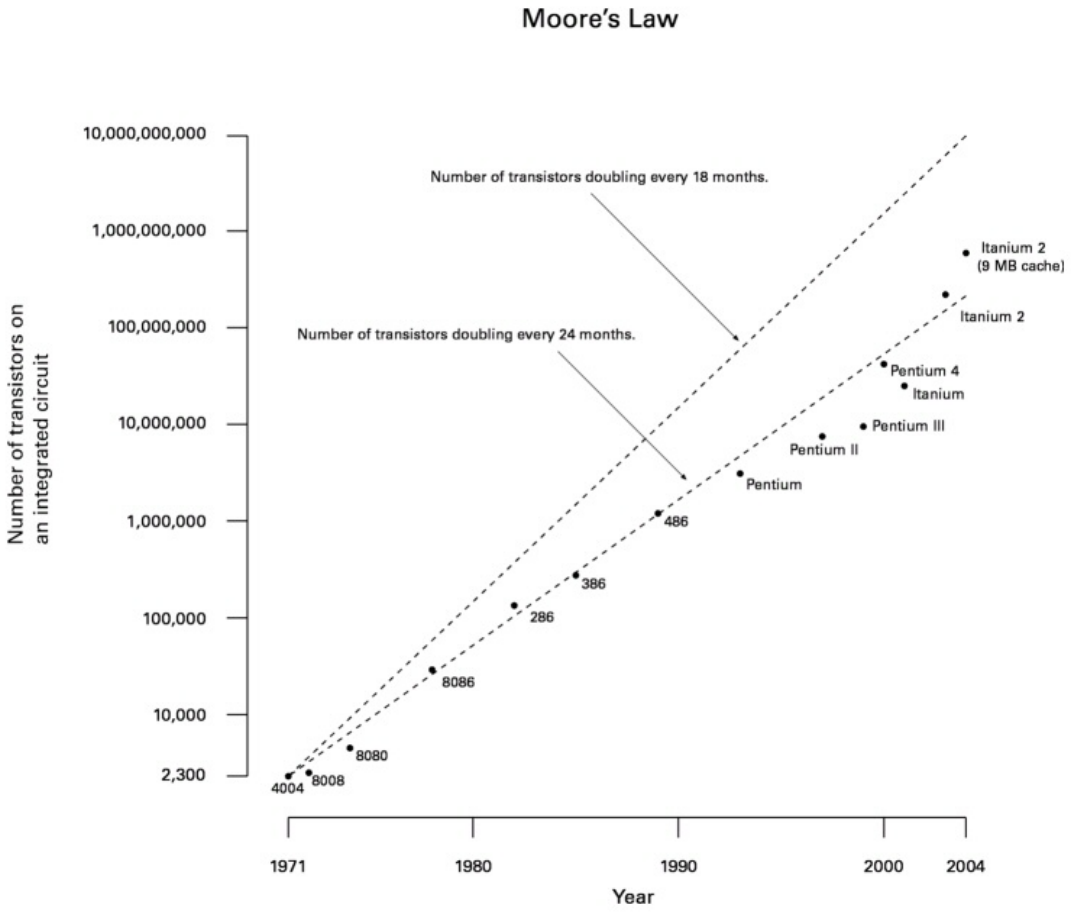

FIG. 1.3 – La "loi" de Moore (©Wikipedia/creative commons)

la loi de Moore pour le silicium.

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 chanceux 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.

1.4.3 Et maintenant, un peu de modestie

Mon théorème préféré :

Si un graphe ne possède pas de sous-graphe isomorphe au graphe complet d'ordre 4, et si, pour tout coloriage de ses arêtes à l'aide de deux couleurs, il existe un triangle monochromatique, alors le nombre de ses sommets est supérieur à $F = 10^{10^{10^{10^{10}}}}$ *(nombre de Folkmann).*

Courrez toujours, un ordinateur construit avec tous les atomes de l'univers et tournant jusqu'à la fin de l'univers n'arrivera pas à énumérer un tel graphe.

1.4.4 Ya pas que les PC dans la vie

Juste deux 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 comparable à celle du Pentium cinq ans avant, 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).

Première partie

L’information et comment on la traite

Chapitre 2

Coder

2.1 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.

2.2 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. Les transistors actuels savent commutent tellement vite qu'on peut tout faire en numérique. Ce fut une vraie révolution, mais personne n'a rien remarqué.

2.3 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).

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 bijection entre un tel vecteur et une grandeur dans un autre domaine.

2.4 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.

Toutefois, le précepte qui reviendra tout le temps, c'est : un bon code est un code qui permet de faire facilement les traitements utiles. 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. Pour cela, on a inventé les codages de position comme par exemple celuis 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 $X = \sum_{i=0}^{n-1} 2^i x_i$.

Si on veut gérer des entiers relatifs, on peut ajouter un bit qui contient l'information de signe. On verra qu'il y a des codages un peu moins intuitifs qui permettent de faire les additions/soustractions plus facilement.

Lorsqu'on construit un calculateur, on est souvent appelé à superposer des codages. Par exemple,le boulier code des grands nombres en base 10, et chaque chiffre est représenté par un mélange de binaire et d'unaire. Encore une fois, la motivation est de faciliter le calcul (on peut reconnaître les chiffres d'un coup d'œil, les additionner d'un mouvement du doigt, etc). Les calculettes fonctionnent également en base 10, avec chaque chiffre codé en binaire. Mon papier sur *single electron counting* reconnait qu'il faut limiter le nombre maximum d'électrons dans un condensateur, et utilise ces condensateurs comme chiffres.

2.5 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, 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 unifié dans un seul code, sur 16 bits, l'ensemble des écritures de la planète. 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.

2.6 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'œuil 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 programme de dessin vectoriel (CorelDraw, XFig) on ne code pas des images, mais des figures construites à partir de primitives telles que point, droite, cercle...

2.7 Codes correcteurs d'erreur

En TD

2.8 Compression d'information

En TD

2.9 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.

Chapitre 3

Transformer

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'opérateurs et de propriétés.
- On utilisera les opérateurs pour construire des expressions booléennes, et on utilisera les propriétés pour les manipuler.
- En 3.2, on implémentera ces expressions sous forme de circuits (dans une technologie donnée), 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 $(\top, \bot)$.
- 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 : $\neg a \vee b == (\neg a) \vee b$.

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

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 : 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'exemple précédent montre que 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, 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

Voici les dessins des portes ET, OU, NON.

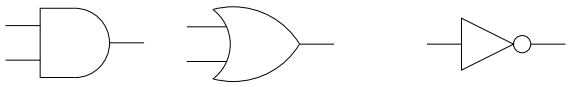

FIG. 3.1 – porte ET, porte OU, porte NON (inverseur)

On peut mettre des petits ronds sur les entrées et les sorties pour dire qu'on les inverse.

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.

3.2.4 Circuits combinatoires

Définition : un circuit combinatoire, c'est une réalisation physique d'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 3.2.

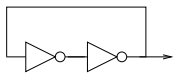

FIG. 3.2 – Le bistable

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 3.3.

3.2.5 Circuits combinatoires bien formés

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 des sorties d'un CCBF à des 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 3.3,
– connecter des sorties entre elles (que se passerait si une sortie est 1 et l'autre 0 ?)

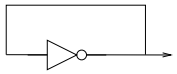

FIG. 3.3 – Le pas-stable

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

Il faut insister sur le "une" (réalisation) dans la définition 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 $[t, t + \tau]$. On apprendra à construire des circuits dans lesquels on garantit que ces états transitoires sont ignorés.

(petit dessin de chronogramme)

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.

Dans la vraie vie, le délai d'une porte de base peut être une fonction des transitions sur les entrées, et le calcul du délai d'un CCBF devient un peu plus compliqué.

3.2.7 Quelques circuits utiles

Not, et, ou, non-et, non-ou, xor, multiplexeur.

On se donne une brique de base appelée *aiguillage*, qui en fonction d'une information binaire de *direction* (entrée du bas sur la figure), transmet l'information soit selon un chemin, soit selon l'autre.

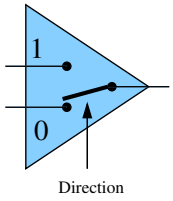

FIG. 3.4 – L'aiguillage

En assemblant $1 + 2 + 4 + ... + 2^{k-1} = 2^k - 1$ aiguillage, on sait construire une gare de triage complète. L'adresse est donnée bit à bit sur les fils de direction.

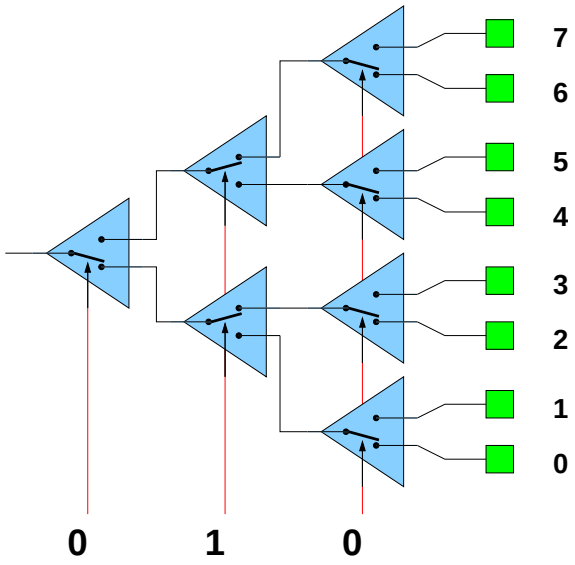

FIG. 3.5 – Une gare de triage permet de construire une mémoire adressable

L'aiguillage, en technologie train électrique, marche dans les deux sens. En CMOS, on distinguera le multiplexeur (2^n en 1) et le démultiplexeur (1 vers 2^n).

Les gares de triage sont construites par une approche algorithmique. On va définir maintenant une approche automatique.

3.3 D'une fonction booléenne à un circuit combinatoire

3.3.1 Formes canoniques

Théorème de Shannon : 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 le théorème de Shannon dual de celui-ci. J'ai perfidement utilisé la notation que j'ai dit que je n'aimais pas.

On appelle un monôme canonique un ET de toutes les variables apparaissant chacune soit sous forme directe soit sous forme complémentée.

On appelle monal canonique le dual du précédent.

Les théorèmes de Shannon, appliqués 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 somme, mais je vous dis que je n'aime pas) de monomes
- Forme canonique conjonctive : ET de monals (monaux ?)[2]

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.

3.3.2 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.

On peut appliquer des règles de réécriture.

3.3.3 Arbres de décision binaire

Construction de l'OBDD

Réductions (ROBDD)

- Si deux sous-arbres sont identiques, on les fusionne
- Si un nœud a deux fils identiques, on le court-circuite et on le supprime.

Problème : quel ordre des variables va donner les simplifications les plus avantageuses ? Problème difficile. Heuristiques.

3.4 Circuits pour le calcul

3.4.1 Des petits cailloux aux bouliers

3.4.2 Addition et soustraction binaires

Pour construire un additionneur, la bonne technique est de faire le plus d'algorithmique possible, et de ne recourir à l'approche automatique que pour les petits blocs. C'est *toujours* la bonne technique.

On part de l'algo d'addition en décimal, puis en binaire.

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.

3.4.3 Multiplication binaire

En TD

[2] C'est sans doute pour cela qu'on préfère habituellement la disjonctive...

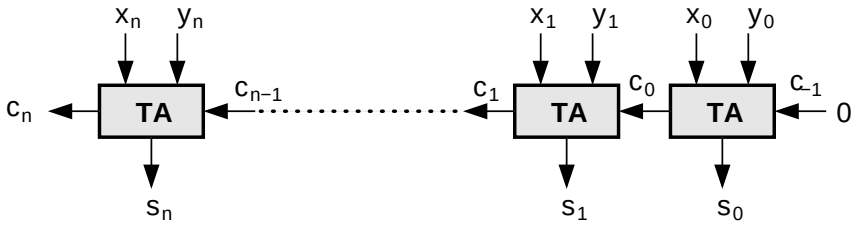

FIG. 3.6 – L'algo d'addition déroulé dans l'espace. TA c'est "table d'addition".

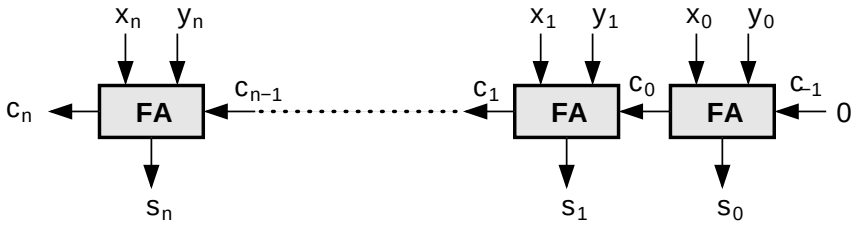

FIG. 3.7 – Additionneur *binaire* à propagation de retenue.

3.4.4 Division binaire

En TD

3.5 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 des opérateurs de calcul.

Plus tard, on abstraira de même le comportement séquentiel d'une machine par un automate, on dérivera une technique universelle d'implémentation des automates, et on constatera à nouveau qu'un peu d'intelligence permet d'obtenir les meilleurs résultats de cette technique.

3.6 Annexe technologique contingente : les circuits CMOS

3.6.1 Transistors et processus de fabrication

3.6.2 Portes de base

Inverseur, non-et, non-ou, porte de transmission.

3.6.3 Vitesse, surface et consommation

Logical effort.

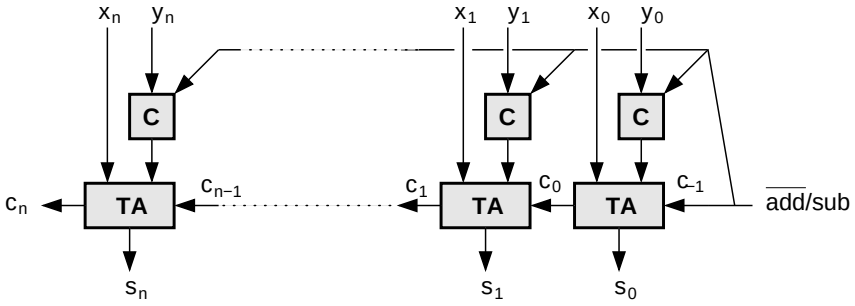

FIG. 3.8 – Additionneur en complément à 2.

Chapitre 4

Memoriser

Enchainement des calculs, accumulateur. Même dans un calcul : retenue, ce qu'on retient.
Re-exemple de déroulements spatiaux et temporels de la somme de n nombres.

4.1 Organes de memorisation

4.1.1 Le registre ou mémoire ponctuelle

On part du bistable double-inverseur.
On ajoute deux portes de transmission. Ou deux portes "ou" (exo). Avantages comparés des deux dessins ?
Dessin d'un registre :
Mémorisation sur front (changement de valeur). 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).
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. Exemple : le Pentium a 4 registres entiers et 8 registres flottants alors que l'Itanium a 128 de chaque.

4.1.2 Mémoires à accès séquentiel : piles et files

Pile ou LIFO : primitives
- empiler(info)
  ajoute un étage
- dépiler()
  enlève un étage : destruction de la donnée dans la pile
- test pile vide

File ou FIFO : primitives
- 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".

4.1.3 Mémoires adressables

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

– 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.

On appelle aussi la mémoire adressable RAM *(random access memory)*, parce que c'est plus court.

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 :

En TD

4.2 Construction des mémoires

4.2.1 Point mémoire

Le registre dessiné précédemment est une mémoire statique (l'information est stable) mais volatile (elle disparaît quand on coupe l'alimentation).

On sait faire des points mémoirqe *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 statique, qui peut fournir plus de courant (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. Mémoire statique et non volatile, haute densité, lecture non destructrice (puisque le transistor reste bloqué ou passant). Inconvénient : écriture relativement lente (haute tension), seulement 100 000 cycles d'écriture.

4.2.2 Mémoire adressable

On sait construire une RAM avec des multiplexeurs, des démultiplexeurs, et des registres. En vrai, il y a plus économique : on verra en TD.

On n'a pas trouvé mieux qu'utiliser des RAM et des compteurs pour implémenter les piles et les files.

4.2.3 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.

4.2.4 Une loi de conservation des emmerdements

Une mémoire de grande capacité bon marché est lente, une mémoire rapide est limitée en taille, et chère.

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)

Remarques :

– Ce tableau 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 loin comment construire des mécanismes qui font croire au processeur que toute sa mémoire vive est aussi rapide que les registres (mémoire cache) et aussi grande que son disque (mémoire d'échange). Mais c'est de l'optimisation, pour construire un ordinateur simple (disons une Gameboy) on n'en a pas besoin. Et cela nécessite d'abord d'avoir construit un système d'exploitation, on n'en est pas encore là.

Chapitre 5

Transmettre

5.1 Medium

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

Notion de canal.

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.

5.2 Liaison point a point

5.2.1 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.

5.2.2 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 serbo-croate antique).

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" 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 l'horloge est incluse.

Ce type de protocole ne marche pas avec les circuits Normaliens. 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*. Il faut trois canaux : un de données (D), un de demande (R), un d'acquittement (A) *(acknowledge)* – les initiales viennent du breton.

Au repos, tout le monde est à 0. Voici comment se passe une émission pilotée par l'émetteur :

– 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".

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, le repose 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.

5.3 Bus trois états

porte trois etats. Arbitrage.

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.
- 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.

5.4 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.

5.4.1 Les topologies et leurs métriques

Considérons deux topologies possibles de réseaux pour relier n ordinateurs : l'anneau, représenté figure 5.1, et le tore bidimensionnel, représenté figure 5.3 qui est une extension de la grille bidimensionnelle (représentée figure 5.2) dans laquelle tous les nœuds ont le même nombre de canaux.

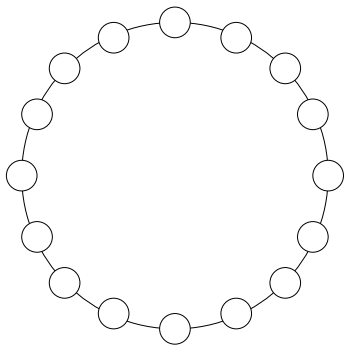

FIG. 5.1 – Graphe anneau

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

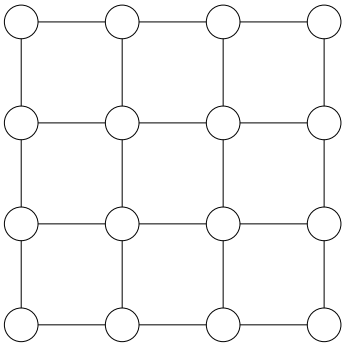

FIG. 5.2 – Graphe grille 2D

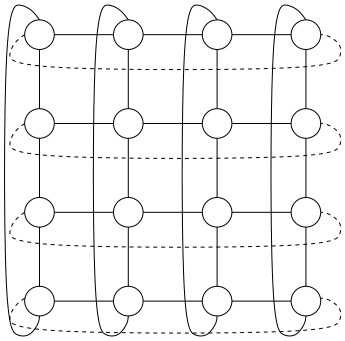

FIG. 5.3 – Graphe tore 2D

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, elle est définie comme le plus petit nombre de fils qu'il faut couper pour transformer le graphe en deux sous-graphes disjoints.

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 5.5 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 $2d$: c'est toujours logarithmique en le nombre de nœuds.

5.4.2 Routage statique et routage dynamique

Commutation de ligne ou commutation de paquets.

Le avantage de la communication de paquets, c'est que la question du routage peut devenir décentralisée.

Commutation de paquets : store and forward, wormhole (flit).

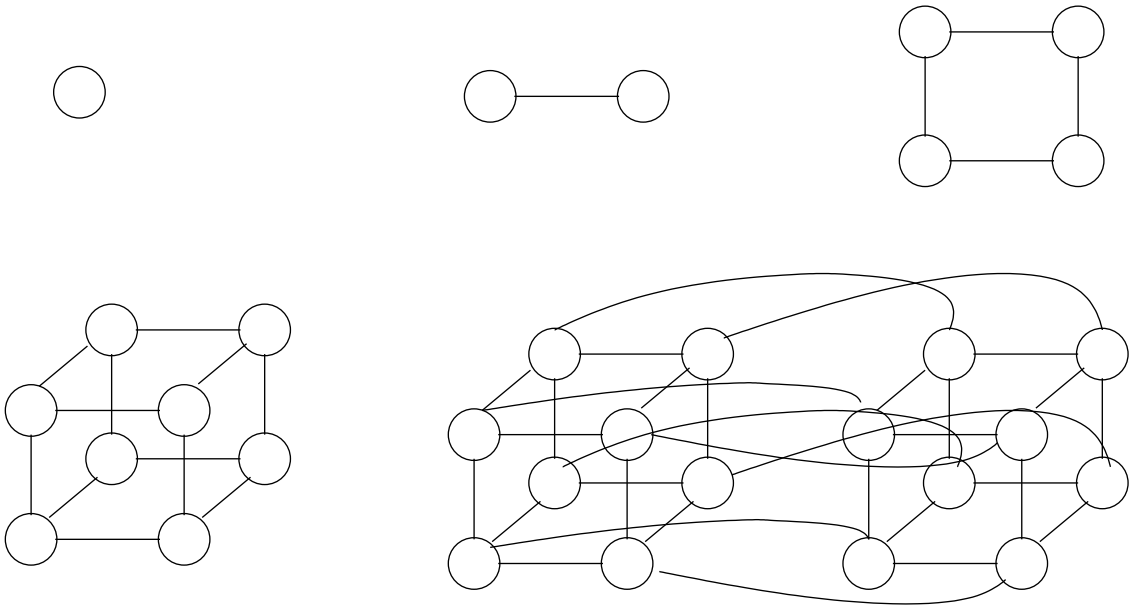

FIG. 5.4 – Graphes hypercubes

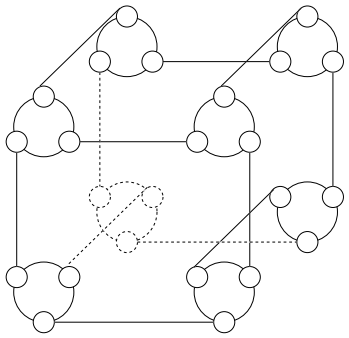

FIG. 5.5 – Graphe *cube-connected cycles*

Problèmes : famines, deadlock (interblocage). On les évite par des protocoles.
Exemples de preuve que pas d'interblocage et pas de famine :
– anneau (protocole Token Ring),
– routage 2D Manhattan,
– routage hypercube.

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

5.5 Exemples

5.5.1 Le téléphone à Papa

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

5.5.2 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.

5.5.3 FPGAs

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

5.5.4 Machines parallèles

Vous verrez avec Eddy Caron ou Yves Robert, si la fin du pétrole ne les cloue pas à tout jamais dans leurs campagnes.

Chapitre 6

Automates

Le formalisme de la logique booléenne permet de construire les fonctions combinatoires. Il nous faut un formalisme pour des circuits prenant en compte le temps : on veut d'une part des circuits qui réagissent à des événements, d'autre part des circuits dont le comportement dépend de ce qui s'est passé avant (qui ont une mémoire). Exemples :

- Un passage à niveau doit réagir à l'événement "un train approche" en fermant ses barrières et en faisant passer des feux au rouge. Ensuite il doit réagir à l'événement "le train s'éloigne" en les ouvrant etc. Pas de mémoire ici – enfin si : la mémoire que la barrière est fermée est dans la position de la barrière elle-même.
- Un digicode doit réagir aux pressions sur ses boutons, et se souvenir des chiffres déjà tapés.
- Un processeur doit passer répétitivement par les trois étapes du cycle de von Neumann. De plus, dans les détails, il doit conserver une certaine mémoire des cycles précédents – par exemple la valeur du PC, et de ses autres registres internes.

On va définir des *automates finis* (ou FSM ou FSA pour *finite state machine* ou *automaton*) comme une abstraction facile à dessiner du *comportement attendu* d'un circuit. Le comportement, c'est ce qu'on peut décrire de plus abstrait ! C'est le pendant d'une fonction booléenne, qui décrit le comportement d'un circuit logique. Cette abstraction est naturellement asynchrone (l'automate réagit à des événements, sans qu'il y ait nécessairement un Maître du Temps qui commande tout).

Ensuite on verra comment implémenter un tel automate fini par des registres et des fonctions booléennes. On verra que la réalisation physique des automates asynchrones pose des problèmes pratiques qui deviennent vite insurmontables. On les évacuera en approximant les automates asynchrones par des automates synchrones, qui sont plus gentils.

6.1 Événements, états, transitions

Revenons sur la notion d'événement, qui un aspect temporel : un événement, c'est à un instant donné. On repérera cet instant par une *transition* d'un signal. Toutefois, un événement porte aussi une information sur l'état de l'univers à cet instant. Cet état de l'univers sera abstrait par une *formule logique*. La difficulté pratique sera de gérer conjointement ces deux aspects.

Quant à la mémoire, elle est abstraite par la notion d'*état* de l'automate. Le circuit de gestion du passage à niveau a au moins deux états : "barrière fermée", "barrière ouverte".

Allons-y donc pour le formalisme. Un automate d'états finis, tels qu'on les manipule en architecture[1], est un quintuplet (S, I, O, T, s_0) où

- S est un ensemble fini d'états.
- I est un ensemble fini d'entrées binaires,
- O est un ensemble fini de sorties binaires,
- T est une *fonction de transition* de $S \times I \to S \times O$

[1] Vous trouverez chez les théoriciens des définitions subtilement différentes, avec des alphabets et des états acceptants. La notion d'état est la même, mais sinon mieux vaut pas trop mélanger.

– $s_0 \in S$ est un état spécial appelé *état initial*.

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

On fait des dessins avec

– des ronds pour les états
– des flèches d'un état s_i à un état s_j chaque fois qu'il existe une transition de s_i à s_j. On étiquette la flèche avec l'événement qui a eu lieu en entrée, et avec la nouvelle sortie. Celle-ci ne changera pas jusqu'au prochain événement.

Vous remarquez la dissymétrie entre les entrées, par lesquelles entrent des événements, et les sorties, par lesquelles sortent des valeurs.

Exemple : le passage à niveau a deux capteurs, à 300m de chaque côté, qui sont normalement à 0, et sont à 1 lorsqu'un train pèse dessus.

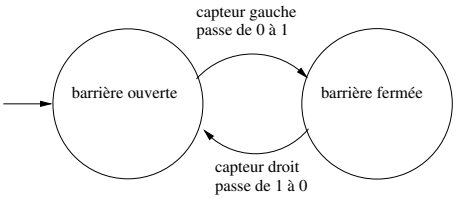

FIG. 6.1 – Un automate pour le passage à niveau

Exemple : le même, mais qui prend en compte que les trains peuvent venir de gauche ou de droite sur la même voie.

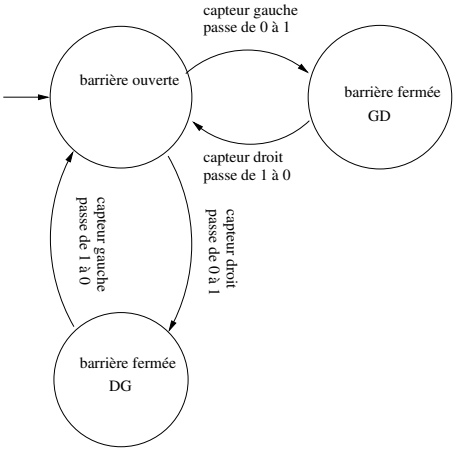

FIG. 6.2 – Un train peut en cacher un autre

6.2 Variations sur le thème de l'automate fini

En définissant les états, on a quand même déjà pas mal fixé l'architecture d'une future implémentation de l'automate. Si on définit un comportement par la fonction qui à une suite (possiblement infinie) d'événements associe la suite de sorties correspondante, on peut avoir plusieurs automates qui implémentent le même comportement. On dira *équivalents* deux automates ayant le même comportement (équivalence comportementale).

– On a utilisé une fonction de transition, ce qui rend l'automate déterministe : pour tout couple (état, événement) il existe au plus une transition. Si on avait une relation de transition (le même événement, partant du même état, peut vous envoyer dans deux états différents) alors on aurait des automates indéterministes. Théorème : il est toujours possible de

construire un automate déterministe équivalent à un automate indéterministe (la preuve est constructive).
- Un automate est dit *complet* ou *réactif*, si pour tout couple (état, événement) il existe au moins une transition dans T. Lemme : dans ce cas l'automate ne bloque pas.
- Les automates décrits ci-dessus, avec $T : S \times I \to S \times O$ sont dits de Mealy. On peut aussi définir des automates de Moore, dans lesquels les sorties ne dépendent que de l'état : au lieu de $T : S \times I \to S \times O$, on a $T : S \times I \to S$ et $R : S \to O$. Dans ce cas, on dessinera bien sûr la sortie dans le rond de l'état, pas sur une flèche. Théorème : pour tout automate de Moore, il existe un automate de Mealy équivalent, et inversement. La preuve est aussi constructive, et l'automate de Moore a en général besoin de plus d'états.
  Certains problèmes sont intrinsèquement plus facile à décrire en Moore, d'autres en Mealy. On saura implanter les deux pareils.

Reste un problème très difficile en théorie, mais pas vraiment gênant en pratique : étant donné un automate, minimiser son nombre d'états.

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 $24 + 60 + 60$.

Un problème lié est de montrer que deux automates sont équivalents.

6.3 Difficile mise en œuvre de l'automate asynchrone

On code l'état, devinez par quoi, par des états de signaux booléens.

Exercice 1 : vous avez droit à la bascule RS et aux portes logiques, bricolez l'automate du passage à niveaux.

Exercice 2 : vous avez droit aux deux registres sur front et aux portes logiques, bricolez l'automate du passage à niveaux.

On est obligé de mettre des portes devant les entrées d'horloge des registres, et il se pose alors des tas de problèmes de *courses* entre les signaux. Ces problèmes sont dificiles à résoudre de manière automatique.

6.4 Synthèse d'un automate de Moore synchrone

Pour évacuer ces problèmes, on munit le système d'une horloge périodique, et on reformule nos automates pour que les seuls fronts soient les fronts montants de l'horloge. On ne dessine même plus ce front sur les flèches.

Du coup, implantation générique par le circuit suivant :

Et on se ramène à des choses qu'on connaît bien : des flip-flops, et des fonctions combinatoires.

6.4.1 Correction d'un automate synchrone

On peut alors énoncer les conditions à vérifier pour que l'automate se comporte bien :
- La fonction de transition doit être bien définie :

  déterminisme : partant du même état, il n'y a pas deux transitions qui arrivent dans des états différents sur les mêmes entrées.

  complétude : dans chaque état, toutes les entrées sont prises en compte (fût-ce à l'aide de *don't care*).
- Les aspects temporels (course entre les signaux) sont maîtrisés :

  synchronisme : tous les flip-flops basculent "en même temps", ce qui signifie ici : dans un intervalle de temps inférieur au plus petit temps de traversée de la fonction de transition (lequel est facile à rallonger si nécessaire).

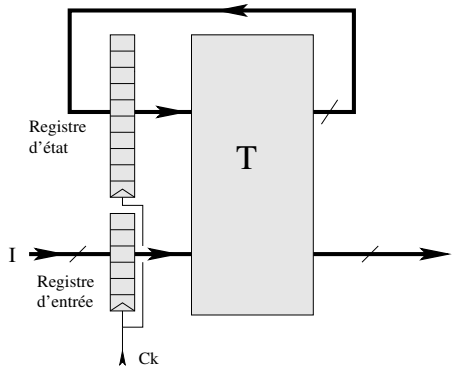

FIG. 6.3 – Implémentation de l'automate de Mealy. Dessinez vous-même à côté l'automate de Moore.

fréquence critique Le plus long temps de traversée de la fonction de transition *(chemin critique)* doit être plus petit que la période d'horloge.

Notez que les deux premières conditions sont des conditions logiques, sur l'automate abstrait, alors que les deux dernières portent sur son implémentation.

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 suivant.

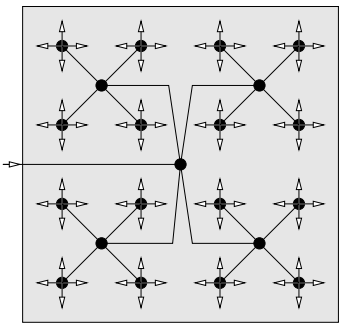

FIG. 6.4 – Arbre d'horloge équilibré. Les ronds noirs sont des inverseurs.

Notez aussi qu'on *échantillonne* les entrées à chaque top d'horloge. Il subsiste un tout petit risque de *métastabilité* en cas de transition d'une entrée trop près d'une transition d'horloge, mais il est bien isolé, et peut alors être géré par des solutions technologiques dans le détail desquelles je ne me risquerai pas.

6.4.2 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). Reprenez l'exemple de la montre à quartz. 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.

– 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.

Deuxième partie

Machines universelles

Chapitre 7

Jeux d'instruction

7.1 Rappels

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

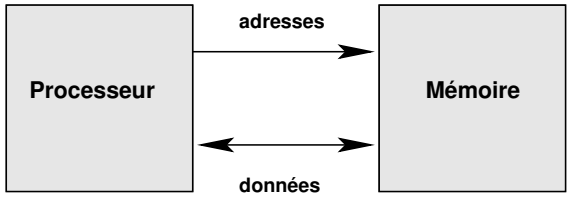

FIG. 7.1 – Mon premier ordinateur

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.

7.2 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 3 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 mémoire 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**.

7.3 Travaux pratiques

Pour vous faire une idée, 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.
Voici les extraits de ce fichier (pour un pentium) qui sont lisibles à ce stade du cours.

```
       (... de la paperasse administrative ...)
movl $17, -8(%ebp)
addl $42, -8(%ebp)
       (... 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).

Voici les lignes correspondant au code assembleur précédent :

```
 Adresses       Instructions binaires   Assembleur
 (...)
 8048365:       c7 45 f8 11 00 00 00    movl   $0x11,0xfffffff8(%ebp)
 804836c:       83 45 f8 2a             addl   $0x2a,0xfffffff8(%ebp)
 (...)
 804837e:       e8 0d ff ff ff          call   8048290 <printf@plt>
 (...)
```

On a une instruction qui fait 7 octets, une qui en fait 4, une qui en fait 5. C'est vraiment n'importe quoi ce processeur, ils ne vont pas en vendre beaucoup.

7.4 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, ...

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. Exemple connu grâce à un marketting efficace : les extension multimedia à IA32, SSE/SSE2/SSE3 chez Intel, 3DNow chez AMD.

7.5 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.

7.5.1 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. La révision du standard ajoutera la quadruple précision. *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 et SSE). On n'en parlera pas plus.

7.5.2 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.

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, utilisée par des ISA datant d'avant que les vitesses de la mémoire centrale et du processeur ne fassent le grand écart, est de ne travailler 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.*

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 ! 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. Notre convention est plus lisible, et comme le mnémonique n'est qu'une représentation textuelle on la garde.

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

- Instructions **à 2 adresses** : `op Rd, Rx -> Rd`. On écrase l'un des opérandes.
- Intructions à une adresse : 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 adresse) : 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 adresses 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 adresses voire plus (voir ci-dessous)
- Avec des instructions à 0, une ou deux adresses, 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 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 adresses, dont une constante et une en mémoire (sur laquelle nous reviendrons).

Voici pour finir un panorama des ISA ARM, PowerPC et IA32.

- ARM est un jeu d'instruction RISC 32 bits à 16 registres. Les instructions sont à 4 adresses : 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` réalise la multiplication 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.
  Anecdote : certains processeurs ARM peuvent fonctionner dans un mode dit *thumb* d'économie de mémoire où toutes les instructions ont 2 adresses 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.
- Power est un jeu d'instruction 32 bits, également à 4 adresses : 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 adresses. 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 permet aussi de passer des paramètres à une fonction appelante. 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.

7.5.3 Instructions d'accès mémoire

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

Test and set atomique nécessaire pour établir des verrous... on verra plus tard

Exemple d'accès à un tableau dans ARM : `LD R1, [R3, R5 LSL#4]`
et avec post-incrément : `LD R1, [R3, R5 LSL#4] !`

7.5.4 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*.

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. La constante ajoutée/soustraite occupe tous les 24 bits restant.

Elles 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*.

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.

Gosub (call)/return *atomique* nécessaire à cause des *interruptions*.

Parenthèse 1 : Pile d'exécution.

Parenthèse 2 : les interruptions. Exemples : le bouton reset du système. L'arrivée d'un paquet sur la carte réseau. La carte son signale qu'elle n'a bientôt plus d'échantillons à jouer. Etc.

7.5.5 Autres instructions

Changement de mode. Modes : au moins *utilisateur* et *superviseur* (système). Les instructions craignos, comme celles qui servent à initialiser la mémoire virtuelle, 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 d'entrées/sorties : de moins en moins. Le plus simple est de les considérer comme des accès mémoire.

7.6 Codage des instructions

Même avec de l'imagination, cela nous fait moins de 256 instructions différentes[1] : 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

[1] Il y en a plusieurs centaines dans IA32...

rendent l'instruction plus puissante. Pour ceux que cela intéresse, cherchez "ARM assembly language" pour avoir une idée de telles options. Voir la Quick Reference Card en annexe.

7.7 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 deux 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, exponentielles... Ce sont des instructions IA32, qui au final sont exécutées par un petit programme stocké dans le processeur lui-même. C'est une caractéristique des ISA CISC. Pour le coup, tout le monde pense que c'était une mauvaise idée.

7.8 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.

Je vous mets en annexe la référence du jeu d'instruction ARM, qui fait 3 pages.

Chapitre 8

Architecture d'un processeur RISC

8.1 Architecture basique

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

L'architecture abstraite d'un processeur capable d'implémenter un jeu d'instructions 16 bits à 2 adresses, construit en TD par vos camarades de 2001 (je crois), est donnée figure 8.1.

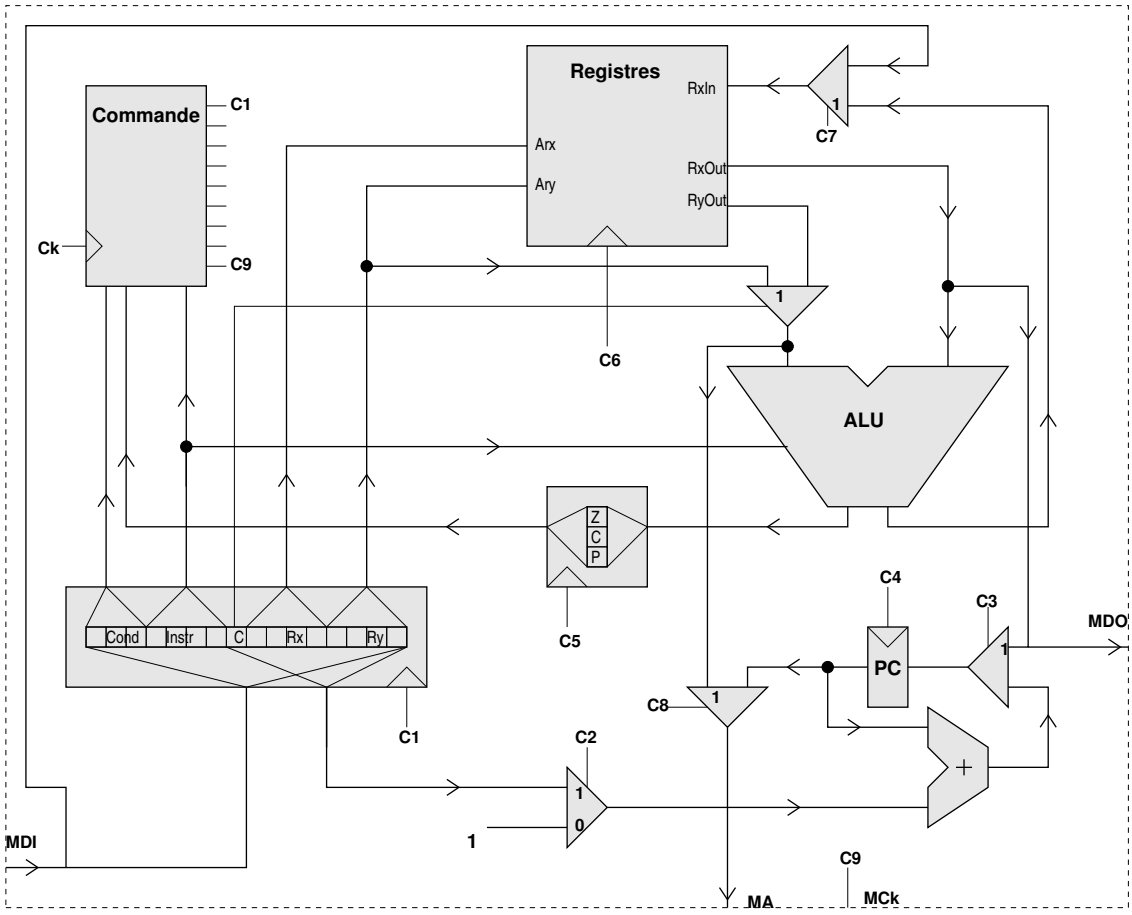

FIG. 8.1 – On ne va pas en vendre beaucoup non plus.

Il faut vous convaincre que vous savez déjà tout construire là dedans. La mémoire est combi-

natoire en lecture.

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. Le traitement d'une instruction peut nécessiter plusieurs cycles, c'est en tout cas certainement le cas des instructions d'accès mémoire. On commence donc par dessiner des chronogrammes qui spécifient ce qu'on veut, par exemple ceux de la figure 8.2.

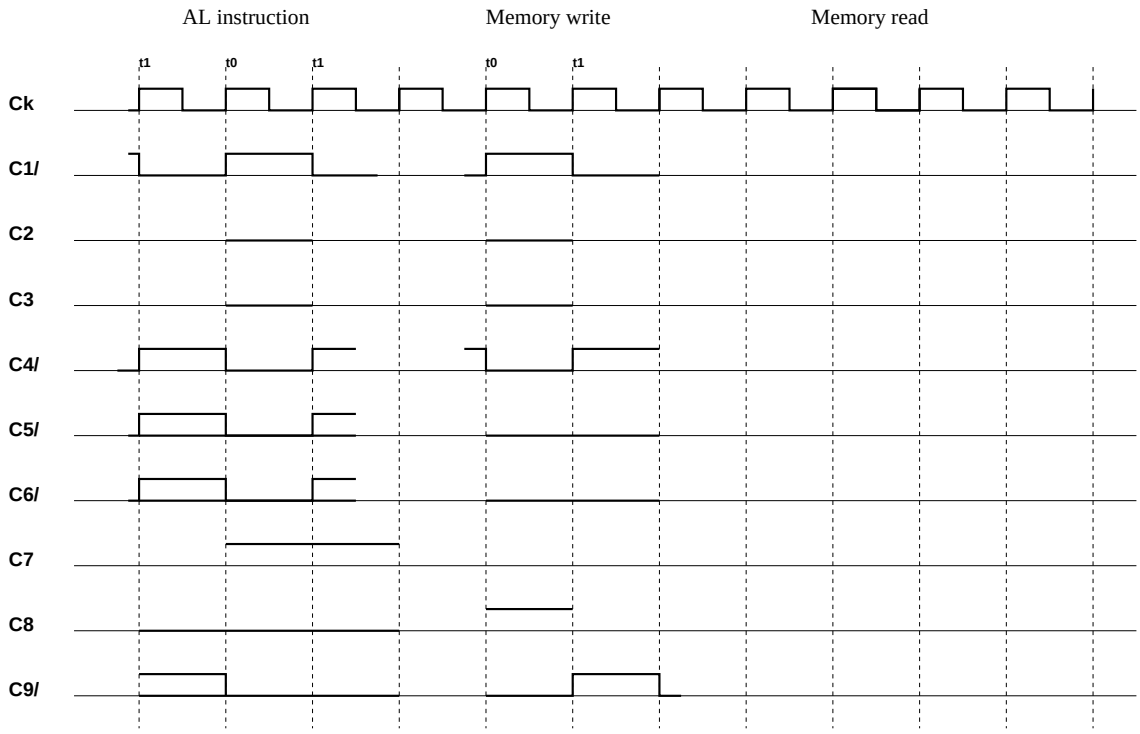

FIG. 8.2 – Les chronogrammes lorsqu'il fonctionne

8.2 Pipeline d'exécution

Les bases

L'idée est simple : c'est celle du travail à la chaîne. 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'instructioon $n-3$ (*execute* ou *EX*), et écrire le résultat de l'instruction $n-4$ dans le registre destination (*writeback* ou *WB*). Ici on suppose qu'on n'a qu'une unité d'exécution – quand on en a plusieurs c'est la section suivante. Et on oublie momentanément les accès mémoire.

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 instructions dans l'ordre du listing du programme) pour le programme suivant :

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

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.

Maintenant parlons des accès mémoire. On va déjà supposer qu'on a un cache séparé programme/donnée, pour pouvoir EX une instruction mémoire en même temps qu'on IF une instruction subséquente. Ensuite on se dit 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 que 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 : cela n'ajoute qu'un cycle au temps d'exécution de tout le programme de 100000 cycles.

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           BNZ 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. On distingue

dépendance de donnée de type *lecture après écriture* ou *RAW*, *écriture après lecture* ou *WAR*, *écriture après écriture* ou *WAW*. Seule la première est une vraie dépendance : les deux autres sont dites "fausses dépendances", car on pourrait les supprimer en renommant les registres (à condition d'en avoir un nombre infini). 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.

dépendance de contrôle , typiquement pour savoir quelle branche on prend dans le BNZ.

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.

On gère les dépendances

- d'abord en les détectant (convainquez-vous que c'est relativement facile, bien que coûteux en matériel – 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).
- 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 ?).
- 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 stockuer 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 (no 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. Les court-circuits se voient bien sur le diagramme temporel.
  - Un court-circuit des drapeaux vers l'étage IF.
- 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). 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.

Parenthèse, 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.

Enfin, il y a des *aleas* qui se traitent comme des dépendances : un *cache miss* va bloquer l'étage MEM, par exemple (on parle d'aléa mémoire). Une division par zéro va nécessiter un vidage du pipeline. Pour cette raison, les dépendances sont souvent appelées *aléas (hazard)* aussi. C'est le cas dans le Patterson et Hennessy.

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 ICCA 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 on négligé les nouveaux petits soucis des techno submicroniques, et le P4 n'a pas pu être poussé jusqu'aux fréquences prévues.

8.3 Exploitation du parallélisme d'instruction

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, on peut les lancer dans le désordre.
- Tant qu'à faire, on peut en lancer plusieurs 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.

Par opposition il existe des processeurs dont le jeu d'instruction a une sémantique exposant explicitement le parallélisme (Intel a inventé le terme *EPIC* pour *explicitely parallel instruction computing*). 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.

8.3.1 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 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 destination et supprime ainsi les fausses dépendances.

En résumé, les registres destination sont 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, elle est marquée comme terminée dans le ROB. 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.

8.3.2 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, 8.3.3.

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 va voir 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) ;

Le jeu d'instruction IA64 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 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.

8.3.3 Multithreading

8.4 Multiprocesseurs

Cohérence de cache MESI.

Chapitre 9

Interfaces d'entrée/sorties

Et le clavier ? Et l'écran ? Et la carte son ? Et la prise pour le modem ?

On ne s'attardera pas sur tous ces gadgets, car moralement ils fonctionnent tous sur le même principe : ils ont le droit d'écrire ou de lire des informations dans certaines zones bien précises de la mémoire, et ce dans le dos du processeur. C'est comme cela que l'extérieur échange des informations avec l'ordinateur.

Pour cela on met en place des *protocoles* connus de l'ordinateur et du gadget. Par exemple, 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 si nécessaire informe le clavier qu'il les a lues (et donc que la zone en question est libre à nouveau).

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.

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 du moniteur, comme jadis.

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 compacte.

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. Toutefois, certaines entrées peuvent envoyer une *interruption* au processeur.

Troisième partie

Système d'exploitation

Chapitre 10

Introduction

10.1 Le rôle de l'OS

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.

10.2 Multiutilisateur ou multitâche, en tout cas multiprocessus

On dit processus, ou tâche.
Les processus se partagent
- la mémoire de l'ordinateur (l'espace)
- le temps.

Le temps n'est pas vraiment partagé, il est découpé 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, est l'impression que les processus s'exécutent en parallèle, mais assez longue pour que le temps du changement de contexte reste négligeable. 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.

Chapitre 11

Gestion de la mémoire

11.1 Mémoire virtuelle

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.

11.1.1 Exemple d'implémentation

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 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 des 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.

(tiens, personne ne me l'a fait. Il est dans le papier distribué).

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 plus de 4Go – jusqu'à 2^{64}. 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 $2^{64} - 1$.

Autre remarque : 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.

Ainsi, la traduction d'une adresse virtuelle en adresse physique n'est pas une bijection, mais une fonction : ainsi on pourra avoir plusieurs processus qui partagent le code de `printf`.

La traduction d'une adresse virtuelle en adressse physique peut se faire en principe par une lecture de table. Deux problèmes pratiques :

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 va être d'une part de faire cette traduction par *pages mémoire*, une page étant une unité de mémoire intermédiaire, typiquement 4 Ko. Et d'autre part d'utiliser un mécanisme de cache (comme vu bientôt) : 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).

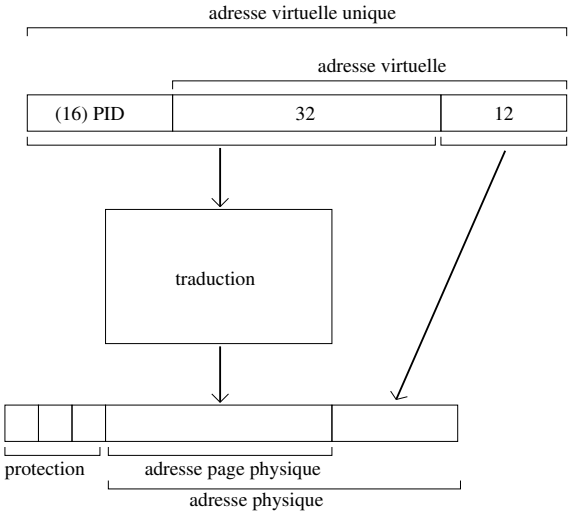

FIG. 11.1 – Vue d'ensemble de l'adressage à travers la mémoire virtuelle

Mais avant de voir les mécanismes de traduction en détail, étendons un peu le cahier des charges de notre mémoire virtuelle.

11.1.2 Avantages de la mémoire virtuelle

Puisque nous allons étendre les adresses virtuelles avec le PID, on peut en profiter pour établir un système de protection de la mémoire plus avancé que la simple isolation. 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 :

- Donner à chaque processus l'impression qu'il est tout seul dans l'ordinateur.
  - impossible d'écrire dans la mémoire des autres processus (isolation mémoire)
  - possibilité d'avoir plusieurs processus qui peuvent chacun allouer toute la mémoire physique.
- 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.
- Donner l'illusion qu'il y a plus de mémoire que la mémoire physique réellement disponible
  - la mémoire qui n'a pas l'air d'être actuellement activement utilisée par l'ordinateur sera jetée sur le disque automatiquement (changement de la traduction virtuelle-physique). La partie du disque correspondant s'appelle alors *mémoire d'échange* ou *swap* en swahili.
  - L'OS ira la rechercher automatiquement quand le besoin s'en fera sentir.
  - LCDE : le Mo sur le disque est bien moins cher, mais bien plus lent que le Mo en mémoire vive.

11.1.3 Table des pages

Tables des pages directe, organisée hiérarchiquement (Fig.2 de l'article distribué).
Accès top-down ou bottom-up.
Pb : adresses virtuelles de 64 bits ?
Solution : table des pages inversées (Fig. 5).

11.2 Mémoire cache

Avec la mémoire d'échange, nous avons un panorama complet de la hiérarchie mémoire d'un ordinateur moderne :

Type	temps d'accès	capacité typique
Registre	0.1 ns	1 à 128 mots (de 1 à 8 octets)
Mémoire cache	1-4ns	4Koctets
Mémoire vive	10 - 100 ns	4 Goctets
Cache disque	100ns	512Koctets
Disque dur	10ms	100Goctets
Archivage	1mn	(illimité)

Il peut en fait y avoir plusieurs niveaux de mémoire cache : le premier sur la puce même du processeur, le second sur une puce séparée, mais tout près tout de même, et avec un bus très large (128-256 bits) avec le processeur. Dans la suite, on ne va considérer qu'un niveau car les mécanismes sont les mêmes lorsqu'il y en a plusieurs.

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.

11.2.1 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.

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) la donnée sera d'abord chargée de la mémoire vers le cache. Une fois la donnée fournie au processeur, le cache chargera aussi les quelques données suivantes.

En fait il chargera une *ligne de cache*, qui est l'unité d'échange entre le cache et la mémoire (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 (petit dessin) : La mémoire physique sera 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.

Une solution un peu moins coûteuse est d'utiliser des mémoires *interlacées* sur un bus de largeur normale. On peut ainsi les lire en pipeline (autre petit dessin) : On envoie les demandes

de lectures en séquence sur le bus (une par cycle), et les mémoires répondent chacun son tour (un certain nombre de cycles plus tard) également en séquence, un mot par cycle.

11.2.2 Fonctionnement 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 :* soit toutes les comparaisons sont réalisées en parallèle (coûteux en matériel). Le cache est alors une mémoire associative, comme un dictionnaire dans lequel l'adresse est la clé.
- Cache n*-way set associative* : 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.

Les gens ont 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)

Un problème se pose : quelle ligne on vire du cache ? (dans le cas où on a le choix : en correspondance directe on n'a pas le choix). En principe on aimerait virer la moins utile dans le futur, mais on ne connaît pas le futur. Première approximation : c'est la moins utilisée dans le passé (*least recently used* ou LRU). Pour cela il faut accrocher une date à chaque ligne. En fait tout ce qu'on veut savoir, c'est qui est le plus récent : 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.

Il y a aussi le pb des écritures mémoire. Elles sont bien moins fréquentes que les lectures (déjà le code est surtout lu, et puis pour les données, considérez le produit de matrice de service : on fait n^2 écritures pour n^3 lectures). Deux stratégies :

- 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 l'envoyer 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.

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. Attention toutefois, les deux caches doivent tout de même être cohérents. Et des caches unifiés pour les niveaux suivants.

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.

11.2.3 Statistiques et optimisation des caches

L'exemple précédent montre qu'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 que la performance d'un cache peut se dégrader rapidement si on augmente juste la fréquence du processeur sans toucher à la hiérarchie mémoire.

11.2.4 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 11.2)
- 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.

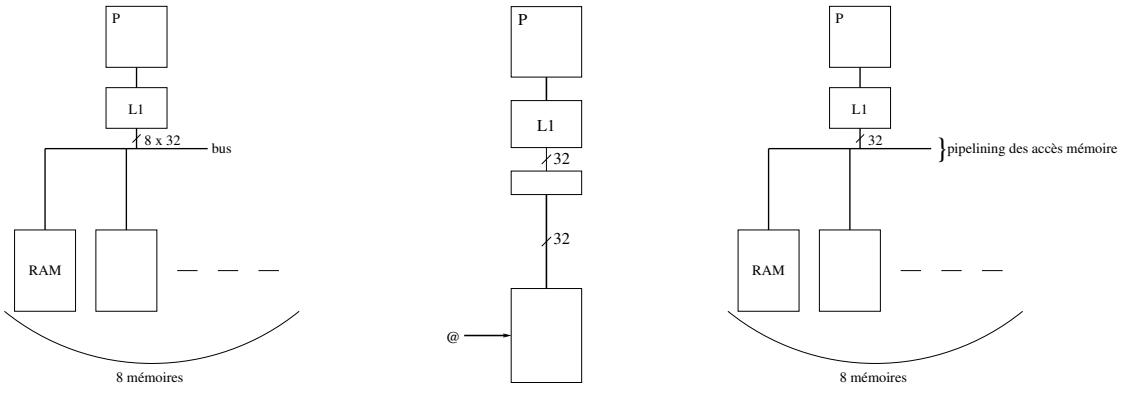

FIG. 11.2 – Entre cache et mémoire physique

11.2.5 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*.

11.3 Une mémoire virtuelle + cache minimale

- Un registre de PID
- Un cache 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 du système. 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 essentiellement en logiciel.
- Les niveaux suivants de la hiérarchie sont indexés physiquement.

11.3.1 Instructions spécifiques à la gestion mémoire

- 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...

11.4 Allocation et récupération

Et nous voilà prêts à écrire la première fonction de notre OS, la fonction `malloc`.

Chapitre 12

Le réseau

12.1 Introduction

12.1.1 Besoins

Différents modes d'échange d'information :
– communication point à point orientée message
– communication point à point orientée flux *(stream)*
– diffusion
– réduction
– multidiffusion

Services annexes :
– adressage : adresse MAC, adresse IP, adresse en texte. Traduction de l'une en l'autre.
– encryption, authentification, sécurité
– localisation (réseaux mobiles wifi et téléphone)
– ...

Partage des ressources physiques entre plusieurs processus/utilisateurs/ordinateurs
– multiplexage temporel (partage du temps, comme les processus se partagent le temps de calcul)
– abstraction possible : canaux logiques, ports
– notion de paquet encapsulant des données
– encapsulation multi-niveaux

Qualité :
– intégrité (pas toujours nécessaire, ex. TV sur friboxe)
– guaranties temporelles (pas toujours nécessaires, ex. courriel)
– performance : débit, latence (performance du médium : "bande passante")

12.1.2 Architecture globale

C'est une architecture mêlant matériel et logiciel.
De haut en bas :
– Pour l'application : appels systèmes par exemple pour
  – demander l'adresse physique de la machine nommée `www.playboy.com`
  – envoyer un mail à `Florent.de.Dinechin@ens-lugdu.fr` (opération de haut niveau)
  – envoyer un paquet sur le port 35 de la machine 11.22.33.44 (opération de bas niveau)
  – attendre un paquet
  – ouvrir un canal temps-réel crypté pour recevoir un flux de données de `www.playboy.com`
– Pour le système d'exploitation :

  - ouvrir/fermer des ports ou des canaux
  - mettre en œuvre les protocoles comme la traduction d'adresse
  - découper/encapsuler les données dans des paquets si nécessaire
  - recopier des tampons, éventuellement sous contrainte de temps réel [1], jusqu'à avoir rempli le tampon de la carte réseau avec les informations qu'il lui faut
  - éventuellement, pour un nœud non terminal du réseau, propager les paquets des autres.
- Pour la carte réseau
  - envoyer et recevoir des paquets sur le réseau.
  - pour cela, implémenter un protocole de communication qui dépend de la techno qu'elle a derrière. Exemples : ethernet, token-ring, ADSL, ATM
  - certains de ces protocoles (ATM) sont eux-mêmes à base de paquets plus petits.

Certains des services de l'Internet demandent une administration centralisée (annuaire des noms de domaine). Toutefois le gros du travail est décentralisé : l'approche décentralisée est plus robuste en cas de panne, supporte mieux la charge là où elle est, et passe mieux à l'échelle. Le routage des paquets est également totalement décentralisé.

12.1.3 En résumé : les défuntes couches OSI

Communications physiques dépendant de la technologie, de si on communique avec son voisin de bureau ou avec Hong Kong.

Mais vue applicative unique : clients, serveurs et services.

Abstraction derrère des notions d'adresse, de canal, de port, de paquet, qui se traduit par des couches logicielles et matérielles. Pour vos données, cela se traduit par des couches d'emballage (encapsulation), qui au passage consomment de la bande passante.

Effort de normalisation par l'ISO *(international standard organisation)* : les couches OSI *(open system infrastructure)*. Jamais vraiment pris racine, parce que pendant les négociations on continuait de construire un internet qui marchait fort bien – entre autre parce qu'il reprenait des bonnes idées de l'OSI. Même le fait que l'état américain rende obligatoire la compatibilité OSI n'a pas réussi à l'imposer.

Bref, les voila :

1. couche physique (envoie des bits de point à point)
2. couche lien de donnée (gère l'emballage et le déballage des bits dans des paquets appelés alors *frame*)
3. couche réseau (gère le routage des paquets)
4. couche transport (gère les communications processus à processus)
5. couche session (gère les différents flux associés à une même application, par exemple synchronise les paquets audio et video d'un film)
6. couche présentation (définit la taille en bits d'un entier, le format d'un flux videlo, ce genre de choses)
7. couche application

Un nœud qui relaye juste un message qui ne lui est pas destiné n'a besoin pour cela que des trois premières couches. La définition des trois couches supérieures est un peu vague, les frontières sont soit mal définies, soit trop bien définies ce qui interdit certaines optimisations.

On va voir que l'internet, lui aussi organisé en couches, est plus flexible.

[1] Ici "temps réel" signifie "en assurant un temps de transmission total inférieur à un maximum préalablement spécifié par le protocole, au vu de la lenteur ridicule des canaux d'entrée/sortie de l'utilisateur humain". Par exemple, moins d'un 25ème de seconde pour recevoir puis décrypter puis décompresser l'image suivante d'un film. Pour transmettre une information en vrai temps réel il faudrait déjà savoir la transmettre plus vite que la lumière. Heureusement, il y a de la marge entre la durée qu'il faut à la lumière pour faire le tour du monde et 1/25ème de seconde. Oui, je sais, la lumière va droit, elle ne fait pas le tour du monde.

12.2 Les couches de l'internet

On n'a que 4 couches :

1. la couche "carte réseau" : elle dépend complètement de la techno qui est derrière, et les couches supérieures ne veulent pas en entendre parler. Une carte réseau va faire intervenir du logiciel et du matériel, et peut éventuellement travailler sur des paquets, etc.
2. la couche IP *(internet protocol)* : elle définit l'abstraction du réseau logique, qui va permettre à plein de technologies différentes de coopérer dans un seul réseau Internet avec un grand I. Pour cela, cette couche définit la notion d'adresses IP, de paquet IP avec son format incluant l'adresse IP du destinataire, etc (voir ci-dessous).
3. Une couche pour TCP, UDP et quelques autres (en gros la couche transport)
4. La couche application

Ces couches sont organisée de manière plus flexible que chez OSI : il est courant que la couche application parle directement à IP, voire à la carte réseau, sans passer par une couche transport.

12.2.1 IP

IP *internet protocol*[2] définit un protocole
– un espace d'adressage
– un service de transmission de paquets sans aucune garantie :
  – un paquet peut se perdre
  – un paquet peut arriver deux fois
  – les paquets peuvent arriver dans le désordre.

C'est une philosophie "best effort" : on fait ce qu'on peut. En terme d'implémentation, la traduction est un protocole assez léger qui par conséquent "run over anything". C'est en fait le bon choix, parce qu'il est facile d'implémenter une couche IP au dessus d'une technologie réseau fiable, alors que le contraire serait difficile.

Si vous voulez une garantie de service, ce sera de la responsabilité des couches supérieures (on va voir TCP et UDP ci-dessous).

Un paquet IP est composé d'un *en-tête* (*header* en impérialiste) qui fait typiquement 20 octets mais peut faire plus si nécessaire. Dans cet en-tête, à des emplacements fixés par la norme IP, on trouve

1. la taille totale du paquet, la taille de l'en-tête
2. l'adresse de destination, l'adresse de départ
3. un *checksum* de 16 bits qui permet de vérifier l'intégrité du paquet
4. des informations indiquant si ce paquet est un morceau d'un paquet plus gros, et si oui quelle est sa place dans ce paquet plus gros (au cours de la transmission d'un paquet IP, il pourra être fragmenté).
5. un champ "TTL" pour *time to live* : il est initialisé à 64 typiquement, et décrémenté à chaque passage par un nœud de routage. Arrivé à zéro, le paquet est jeté sans autre forme de procès. Ainsi, en cas de boucle dans le graphe de routage, les paquets ne s'accumulent pas.
6. un champ donnant la version du protocole IP utilisée (de nos jours IPv4 ou IPv6)
7. un champ "protocol" qui décrit le protocole de la couche supérieure qui a envoyé ce paquet
8. des champs optionnels pour mettre d'autres informations sur le paquet

[2]On peut utiliser le mot "internet" pour '*internetwork*, réseau inter-réseau. Il ne prend alors pas de majuscule. Avec une majuscule, il s'agit de l'Internet omniscient que vous connaissez. Certains ne lui mettent alors plus d'article. C'est passionnant tout cela.

Un gros paquet est fragmenté en paquets plus petits au départ, mais pourra être fragmenté en paquets encore plus petits durant le transport, selon les caprices des nœuds à travers lesquels il passe. Par contre il ne sera réassemblé qu'à destination.

Lorsqu'on doit fragmenter un gros message en plusieurs paquets plus petits, quelle doit être la taille de ces petits paquets IP ? Eh bien la couche IP n'impose rien, c'est la couche supérieure qui choisit. Typiquement, cette dernière peut interroger au préalable la couche "carte réseau", qui pourra lui répondre que ses paquets à lui ne doivent pas dépasser 1500 octets (ethernet) ou 4500 octets (FDDI). La couche supérieure fragmentera alors ses gros paquets en fonction de cela : à tout prendre, c'est le bon choix pour causer avec les voisins. Cela dit, si la couche IP passe des paquets trop gros pour la couche carte réseau, cette dernière les fragmentera et les réassemblera. On y perdra juste un peu en performance.

12.2.2 Routage (commutation de paquets)

IP définit des adresses hiérarchiquement : il y a une part qui identifie un réseau, et une part qui identifie un nœud dans le sous-réseau. Certains nœuds (appelés routeurs) connectent plusieurs réseaux : ils ont alors plusieurs adresses IP, une dans chaque sous-réseau. Un pied dans chaque réseau, quoi. De plus, les routeurs ont une table qui liste, pour tous les sous-réseaux connus, sur quelle interface ils doivent transmettre un paquet.

Il y a un protocole, DHCP (dynamic host condiguration protocol) qui définit la manière dont un ordinateur nouvellement arrivé sur le réseau va recevoir une adresse IP.

Il y a aussi un protocole ami de IP, ICMP (control message) qui en cas d'échec à livrer le message (les raisons ne manquent pas), envoie un message à l'expéditeur, sans garantie qu'il arrive bien sûr. ICMP définit aussi des messages plus positifs, comme "Il existe un chemin plus court de tel réseau à tel réseau" qui permet de mettre à jour les tables de routage.

En fait, la construction de ces tables de routage mérite tout un chapitre à elle seul. Il faut aussi éviter les embouteillages, mainimiser la latence, etc.

12.2.3 UDP

User Datagram Protocol. Sans garantie non plus, mais il vérifie des *checksum*s. Couche qui assure la communication de processus à processus. Se contente de démonter et réassembler des paquets pour les passer par IP.

Comment identifier un processus ? Le PID peut varier d'une session à l'autre. On l'abstrait donc par la notion de *port*. La couche UDP distribue les paquets selon des ports, et chaque processus surveille un port donné.

12.2.4 TCP

Transmission Control Protocol.
Tout comme UDP, mais lui assure des garanties.
Du point de vue applicatif, flux d'octets envoyés et reçus dans l'ordre.
Derrière, système de requête/acquittement + fenêtre glissante.

12.3 Autres réseaux

12.3.1 Réseaux sans fil et mobilité

12.3.2 Réseaux sur puce

Annexe A

Le jeu d'instructions ARM