1: Dissection d'ordinateur conceptuel

Martin Quinson

Épisode précédent

I) Présentation du module

- Ce cours vise à constituer une première introduction sur les ordinateurs et les systèmes informatiques. L'objectif principal est de donner une culture générale et de combattre quelques idées fausses.
- Dans sa version L3 info, c'est un module de 10 semaines couvrant principalement le langage C et le réseau. Le langage C est une excuse pour regarder comment fonctionne un ordinateur sans mettre les mains dans le cambouis. Les réseaux sont un exemple de grand système informatique qui dure depuis 4 décennies car il était bien pensé à la base.
  - Cette version n'insiste pas trop sur le M99 en cours, même si les ressources sont distribuées en cours pour jouer à la maison.
  - MCC: un projet en binôme en semaine 4 à 6, puis un examen sur table à la fin.
- Il existe aussi une version très raccourcie, pour le tronc commun d'agregation SI ou en DIU EIL. Objectifs en 4 heures seulement:
  - Exposer les principes directeurs d'un ordinateurs (première séance sans la partie C, en insistant sur le M99)
  - Présenter ce qu'est un système d'exploitation, à quoi ça sert, et comment ça marche (seconde séance, sans le C, qui est vu en L3 mécatro)
- En 20-21 L3info, je n'ai passé que 5mn sur le M99, ce qui n'est pas assez

- TODO: préenter les résumés à rendre à chaque semaine, par groupes de 5 ou 6. Mettez tout le groupe en CC pour qu'ils aient mes commentaires.

I.1) Introduction à la recherche en informatique appliquée

- Pas de panique, ce n'est pas un cours d'ingénierie (même si c'est sympa).
- Objectif1 est de montrer le large spectre de la recherche en informatique (luter contre la spécialisation)
- Objectif2: mieux comprendre les vrais systèmes informatiques pour que les *problem-solvers* trouvent des problèmes sympas à résoudre. Eviter les affres de la page blanche
- Et y'a même de la recherche en génie logiciel, en particulier à Rennes (Diverse).

I.2) Quelques vérités sur les ordinateurs

I.2.a) Idée fausse 1: Les ordinateurs manipulent des nombres mathématiques

(inspiré du cours Intro to computer systems de CMU)

- Int $\neq$ entiers: $\exists$ n $\in$ `Int` tel que $x^2 < 0$: $50\,000 \times 50\,000 < 0$ (pas de pb avec les flottants)
- Float $\neq$ réels: (x+y)+z $\neq$ x+(x+z) parfois (pas pb avec les entiers)
  - $\boxed{\text{(1e20 + -1e20) + 3.14 = 3.14}}$ mais $\boxed{\text{1e20 + (-1e20 + 3.14) = 0}}$
- `Int` n'est pas $\mathbb{N}$ mais c'est un anneau (commutativité, associativité, distributivité)
- `Double` n'est pas $\mathbb{R}$ mais respecte relation d'ordre (monotonie, signe)

I.2.b) Idée fausse 2: La mémoire d'un ordinateur est uniforme (RAM)

- Pas sans limite: taille max mais aussi précision max
- Source de bugs pénibles (malloc)
- La vitesse dépend de l'endroit où sont stockées les données
  - $\forall$ i $\in$ [0; 2028]
$\forall$ j $\in$ [0; 2028]
dist[i][j] = src[i][j]

    $\forall$ j $\in$ [0; 2028]
$\forall$ i $\in$ [0; 2028]
dist[i][j] = src[i][j]
  - La seule différence est l'ordre des boucles: on parcourt les i avant les j ou le contraire
  - L'un met 4ms et l'autre 80ms. On verra plus tard pourquoi :)

I.2.c) Idée fausse 3: Les ordinateurs ne font que calculer

On trouve pleins d'autres problèmes de recherche intéressants qui ne sont pas directement liés au calcul

- Performance dissimétrique des I/O $\Rightarrow$ algo out of core (moins actif actuellement)
- Compatibilité
  - Internet (TCP/IP) inventé pour $\approx$ 100 noeuds et fonctionne pour 4+ milliards de IPv4
  - Démarrage (BIOS, MBR, OS) fonctionne depuis les années 80 pour des matériels très disparates
  - Recherche: Génie log, équipe Diverse à Rennes
- Concurrence: quand on coordonne différentes entités, il se pose une foule de problèmes intéressants. Recherche: preuve de programme (ariane, voiture), modèle-checking, algo très amusante (wait-free)
- Usage fiable d'un média non fiable/faire avec ce qu'on a: les réseaux fonctionnent avec des cables de mauvaise qualité, et le logiciel améliore/compense cela. C'est pour ça que les ordis sont binaires (setun, ternaire préférable d'un point de vue théorique).

  Recherche: OS (domaine moins actif en OS, pour l'instant), réseau (beaucoup), traitement d'image (énormément)

II) Langage C

C'est la partie plutôt pratique de ce cours. On en a besoin pour le prochain TP. En 20-21, je l'ai fait en 45 minutes en détaillant tout bien.

II.1) Historique

- Inventé dans les années 1970 par Brian Kernighan et Dennis Ritchie, du laboratoire Bells de AT&T, pour abstraire un peu l'écriture de programmes de l'ordinateur cible. Compilateur pour traduire le code source des humains en langage machine des ordis $\rightsquigarrow$ ne pas tout réécrire à chaque fois.
- Mais ça reste un langage très proche de la machine (compilos très limités à l'époque)
- La première machine cible est le PDP-11: 24ko de mémoire $\rightsquigarrow$ KISS
  - La mémoire est un tableau d'octets sans structure
- (le nom est une mauvaise blague car c'est une variante du langage B de laboratoires Bells)
- Le C++ est une extension du C, semblable en apparence mais bien différente (on ne confond pas)

II.2) Pourquoi étudier le C?

- Intérêt historique et culturel
  - Très répandu (tous les OS, 50% de Debian)
  - Bcp de langages s'en inspirent (C++, Java, C#, Python, Ruby, Rust, PHP)
  - Permet de faire des programmes efficaces (mais c'est pas automatique)
- Intérêt pédagogique
  - Comprendre le C permet de comprendre la machine
    - C'est le langage évolué le plus proche de l'architecture
    - Devenu un modèle opérationel concret de la machine, par co-design (le hard s'y conforme)
  - Comprendre C et machine pour mieux programmer en <insert language name>
    - Les VM cachent plus ou moins bien les détails (CAML vs. Perl)
    - Faut connaître les pointeurs C pour bien comprendre les références Python/Java
- Avantages et inconvénients du C
  - (+) C'est un langage très simple, très KISS. Syntaxe en une seule page A4
  - (-) AUCUN garde fou, les outils font une confiance aveugle aux humains, c'est fatiguant
  - (+) On contrôle tout (optimisation) (-) il faut tout contrôler
  - (+) On peut comprendre les méandres (-) 1001 façons de se tirer dans le pied
  - Presque backward compatible avec 1970: (+) stable et pérène (-) qques héritages regrettables
  - Pas de bibliothèque standard: (-) il faut tout refaire soi-même
  - $\Rightarrow$ C'est un langage à connaître, mais à n'utiliser qu'au besoin

II.3) Syntaxe de base du langage C

- **On utilise le pense bête, distribué à chacun.e**
- C'est très similaire au Java ou C++ (forcément), et assez similaire au Python où on marque les blocs
- Il faut déclarer les variables à l'avance, qui sont typées pour aider le compilo (pas pour la sémantique).
  - `int` et `double` pour tout faire (booléen : entiers — vrai $\neq 0$)
- Opérateurs arith (+ - * % ++ +=) logiques (&& || !) relationnels (< <= != ==) binaires (& | ^ «)

- `if () {} else {}` `while () {}` `for (init; cond; inc) {}` `switch () {case:  break; }`
- Préprocesseur: les lignes commençant par # passent dans une machine à cherche-remplace améliorée
- Lecture ensemble du code "Premier programme" du pense bête. Include, fonction, scanf/printf.
- Définition de fonctions et appels de fonction
- Tout le reste (ou presque) c'est en option, pas dans le langage
  - conçu en 1970, c'est à l'humain de s'adapter aux outils
  - messages d'erreur parfois cryptique

II.4) Survivre en C

- Workflow attendu
  - On écrit son source dans un fichier `toto.c` (avec geany)
  - On compile ce fichier (= traduction en binaire exécutable) `gcc toto.c -o binaire`
  - On exécute ce binaire `./binaire`
- Compiler ? C'est quoi en vrai ?
  - Phase 1: préprocesseur (define, include, ifdef)
  - Phase 2: compilation = traduction en assembleur
  - Phase 3: édition de liens = puzzle des ptits bouts d'assembleur (si +ieurs fichiers)
- Premières bétises possibles:
  - Exécuter le source : impossible, c'est pas interprété (le processeur ne parle que des 0 et des 1)
  - Compiler sans resauvegarder (de l'intérêt d'utiliser un IDE)
  - Exécuter sans recompiler (idem)
  - Ne pas lire les erreurs ou warning (clang plus compréhensible)
- Pour survivre, connaît tes outils
  - `-Wall -Wextra -Werror -g`
  - Écrire du C moderne (éviter certaines choses autorisées par le compilo – place des déclarations)
  - gdb et valgrind pour chasser les bugs (y'a un TP spécifique)
  - KISS, c'est trop simple d'écrire du code illisible (on y revient)

III) Ordinateur

C'est la partie plutôt connaissance générale de ce cours. C'est la raison d'être du module, et la partie pratique ne sert qu'à renforcer ces apprentissages. En 20-21, je l'ai fait en 1h15 en allant vite mais en disant tout. On doit pouvoir raccourcir par exemple en n'évoquant pas les différentes méthodes pour calculer toujours plus vite.

III.1) Qu'est ce qu'un ordinateur?

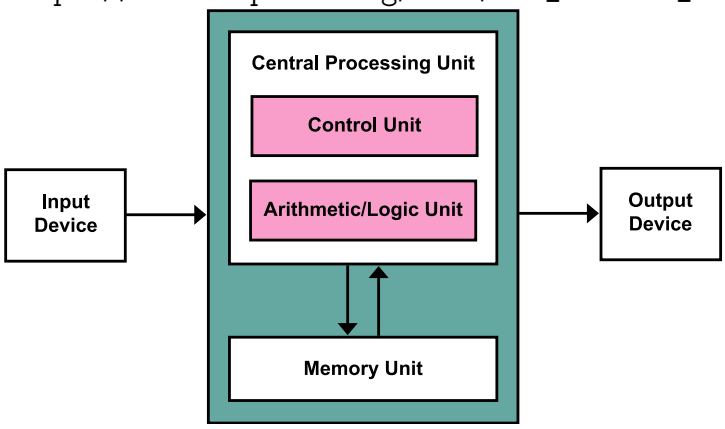
- Machine à calculer universelle.
  - La différence avec un calculateur est qu'il n'y a pas à le modifier physiquement pour lui faire calculer autre chose
  - Le nom FR est un vieux mot FR tombé en désuettude pour désigner Dieu, celui qui range les choses.
- Depuis quand?
  - 1936: Article fondateur de Turing sur la calculabilité et la Machine de Turing

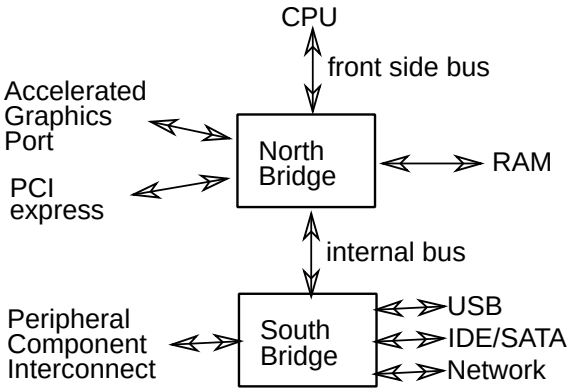
- 1946: Construction de l'ENIAC.
  * Environ 350 flops, ie 2500 fois plus rapide qu'un humain
  * Changer le programme prend jusqu'à 3 semaines pour tout recabler (c'est donc plutôt un calculateur programmable)
  * Pour info, les supercalculateurs modernes sont exaflopiques, 10^{18} flops. 1 milliard de milliard de milliard. Âge de l'univers en seconde: $\approx 4{,}3{\cdot}10^{17}$
- 1962: Premier département d'informatique à l'université de Purdue, près de Chicago. Fork du département de génie civil
- 1886: fondation de Tabulating Machine Company, qui fait des cartes perforées et deviendra IBM par la suite.

III.2) Contenu d'un ordinateur

- Prise de représentation sur les différentes parties d'un ordinateur. On attend CPU, GPU, disque, écran, clavier, bus, ... poussière.
- Modèle de Von Neumann (1945) suffisament simple pour être indémodable.

`https://en.wikipedia.org/wiki/Von_Neumann_architecture`

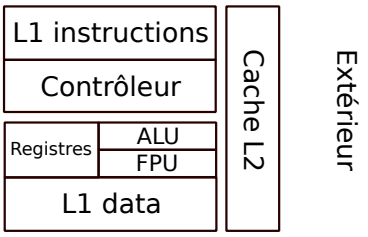




- Les cartes mères actuelles ont des puces qui implémentent grosso modo le modèle de droite
  - Les détails techniques changent beaucoup et tout le temps, mais l'idée est là.
  - De nos jours, le northbridge est souvent directement sur la puce du CPU pour gagner en vitesse
  - PCI et AGP ont disparu au profit du PCIe, maintenant assez rapide pour tout faire
  - Dans les laptops, tout est soudé directement donc la puce du bus remplace le PCIe

III.2.a) Le processeur

- Composants
  - Unités fonctionnelles, qui font les calcul.
    * ALU: Arithmetic/Logical Unit: logique booléenne et travail sur les entiers
    * FPU: Floating Point Unit: calculs sur les nombres à virgule flottante
  - Contrôleur, qui coordonne les calculs
  - Zones de stockage, certaines pour les instructions, d'autres pour les données, ou encore mixtes



- Principe général (à 3 Ghz)
  - *Fetch:* Le contrôleur récupère l'instruction suivante du programme
  - *Decode:* Le contrôleur décode l'instruction, sépare les opérandes
  - *Execute:* L'ALU ou FPU exécute l'instruction
  - Le résultat est stocké dans un registre
- Loi de Moore (ex-président d'Intel)
  - Le nombre de transistors double tous les 18 mois.
  - C'est une prédiction autoréalisante puisque Moore est un ex-PDG d'Intel et que la companie a comme business plan que tous les ordinateurs sont remplacés tous les 18 mois.
  - 1971: 2000 transistors sur un 4004; 2010: 2.6 milliards sur xéon; 2020: 40 milliards AMD epyc
- Gravure de plus en plus fine (`Wikipedia:32_nm_process`)
  - 1970: 10μm ($10{\cdot}10^{-6}$); 1982: 1,5μm; 2001: 130nm; 2014: 14nm; 2017: 10nm; 2020: 5nm
  - Arduino à 40nm est une technologie de 2008 pour Intel, d'où le prix actuel
  - Un cristal de silicium est de l'ordre du nanomètre
  - En dessous de 7nm, les électrons fuitent des transistors par effet tunnel...
  - Les CPU modernes sont multi-couches, ce qui rend le pb de placement encore plus complexe
- Types de processeurs
  - GPU: unité de calcul spécialisée dans le calcul vectoriel sur nombres flottants. Parallélisme de masse. Contient des ALU mais vraiment pas efficace pour les calculs génériques. Comparable à une formule 1: très rapide si le calcul est très régulier, nul sinon.
  - RISC vs. CISC: Reduced/Complex Instruction Set CPU. Plus ou moins d'instructions (de mots au vocabulaire). Plus de mots: plus de séquences directement inscrites dans le silicium, donc plus rapide et plus gros/complexe
  - FPGA: aucune instruction inscrite dans le silicium, tout est interprété
  - Little-Big Endian: ordre des bits pour écrire les nombres entiers. Les ingés Intel ont eu des goûts bizarres, mais soit. Le nom vient des voyages de Guillivers. AMD/x86 est little, ARM est (souvent) big.

III.2.b) La mémoire

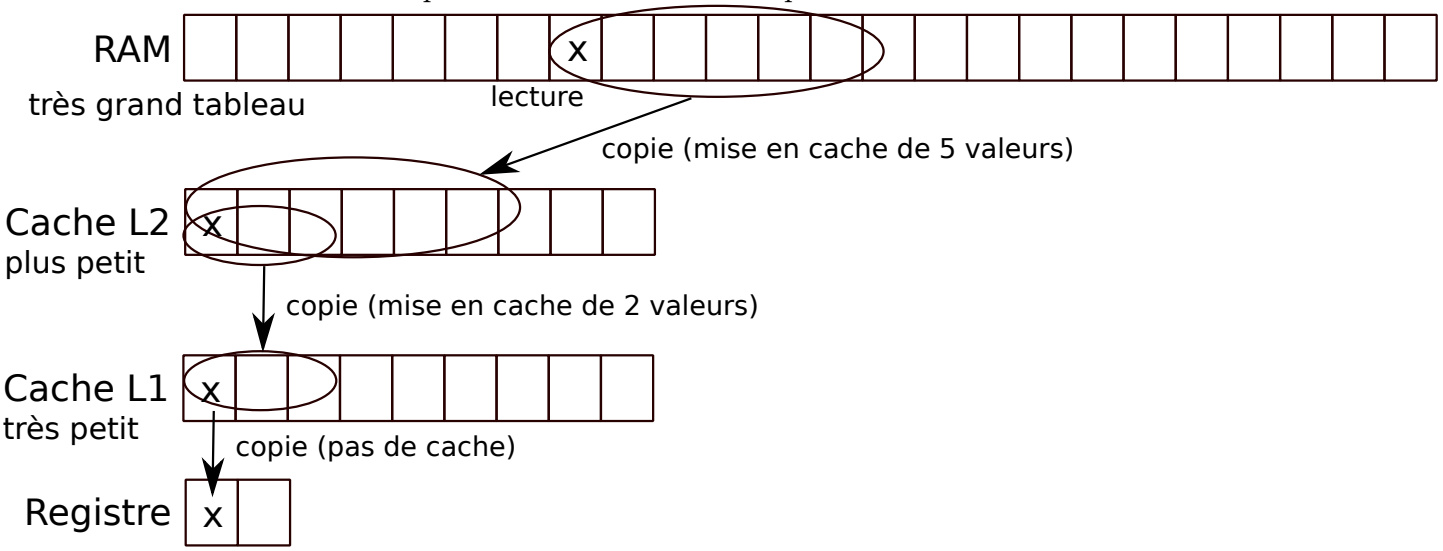
- Pyramide d'accès: On peut représenter les types de mémoire par un triangle avec les ordres de grandeurs suivants
  - Registre: <512 octets pour <1ns
  - Caches: ko-Mo pour 5ns
  - RAM: Go pour 10-30 ns
  - Stockage de masse: To-Po pour 10ms
- Memory wall:
  - Latences mises à l'échelle, pour les ordre de grandeur. Imaginons qu'un cycle fasse une seconde

1 cycle CPU	0,3ns	1s
cache L1	1ns	3s
cache L2	3ns	9s
cache L3	13ns	43s
RAM	120ns	6mn
SSD disk	50-150μs	2-6 jours
disque rotationel	1-10ms	1-12 mois
Internet Oslo/Madrid ou SF/NY	40ms	4 ans
Internet transcontinental	180ms	18 ans
TCP retransmit	1-3s	100-300 ans

- Le CPU va très, très très vite: 0,3ns par cycle, ça fait 10 cycles quand la lumière parcourt 1m. Le défi est donc de nourrir le CPU de données sans pause
- C'est aussi pour ça que les registres ont une petite capacité: ils doivent être petits en taille pour être proche des unités fonctionnelles. S'ils étaient plus gros, la lumière mettrait plus de temps
- Le memory wall s'aggrave: Vitesse CPU croit de 55%/an et celle mémoire de 10%/an.

III.2.c) Les mémoires caches

- L'étymologie vient des trappeurs nord-américains. Ils apportaient du matériel sur un chariot jusqu'à une sorte de camp 0, puis ils cachaient ce qu'ils ne pouvaient pas porter pour l'expédition.
- Dans l'ordi, on a un pb de latence entre la mémoire et l'ALU, mais pas de bande passante. On va donc rapatrier plus de données que demandées, au cas où on aurait ensuite besoin de la donnée juste après en mémoire.
- une mémoire cache est une petite réserve de mémoire proche du coeur du CPU



- La fois suivante qu'on accède aux données, on regarde d'abord si elle n'est pas déja stockée dans un niveau de cache intermédiaire. Si oui, on a économisé la latence. Si non, on charge la ligne de cache nécessaire, en virant une autre ligne au besoin. Pleins d'algos différents de gestion des caches, mais LRU (least recently used eviction) marche bien en pratique.
- "Coup de chance", les accès mémoire sont effectivement souvent linéaires. En fait, on programme exprès pour, et le compilo tente de démeller les accès et les données pour fluidifier.
  - En HPC, on cherche même à faire des structures de données *cache oblivious*, cad optimisées pour n'importe quelle taille de cache.
  - C'est ce qui explique la différence de code entre les deux programmes de copie de matrice (4ms vs.

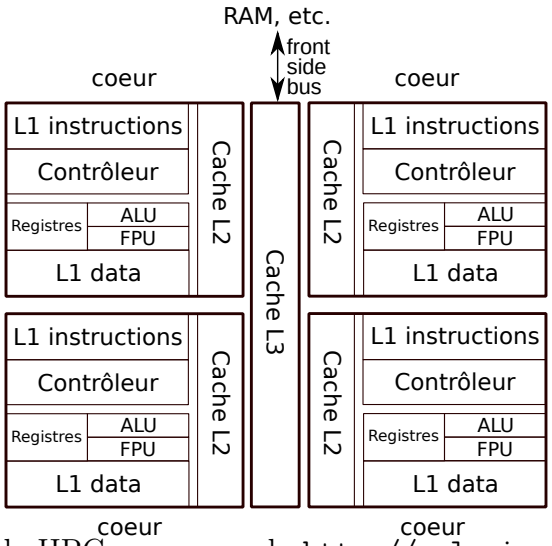
80ms pour une matrice 2048): l'un utilise très efficacement les caches, l'autre passe son temps à les invalider après chaque accès.

III.2.d) Comment calculer plus vite (L3 info: seulement schéma multicoeur sans détails)

- **Miniaturisation**, pour que la lumière porte l'info plus vite. Moteur principal jusqu'au années 1990, moins efficace maintenant car c'est vraiment petit. C'était une solution hardware, trobien.
- **Fréquence**, augmenter la fréquence augmente mécaniquement le nombre de top d'horloge. hardware.
  - Quand la fréquence croit, la quantité de calculs croit linéairement
  - Mais la quantité d'énergie croit quadratiquement. 50% plus vite, 2 fois plus d'énergie.
- **Pipeline**: faire plusieurs choses à la fois niveau silicium
  - Parfois, les instructions CPU prennent plus d'un cycle (surtout en CISC mais pas que). Imaginons que la multiplication prenne 4 cycles. Si on veut faire 3 multiplications (X Y et Z ci dessous), ça fait 12 cycles.
  - On découpe ces opérations complexes en morceau qui prennent un cycle chacun. Les sous-instructions sont chainées, mais du coup on peut commencer l'opération suivante avant la fin de la première, en réutilisant cette zone du circuit. On fait alors les 3 multiplications en 6 cycles.

-
  - La difficulté est ensuite d'assurer que les pipelines restent bien pleins, ie d'éviter les *pipe holes* qui apparaissent quand on a besoin de la fin d'un calcul pour savoir quoi faire ensuite. Dans ce cas, on fait de la prédiction de branche et du calcul spéculatif. Si on a bien prédit, on a déjà commencé le calcul suivant. Si on s'est trompé, on annule le calcul spéculatif et c'est pas grave.
  - Les compilateurs font énormément de transformations du code pour rendre le code plus sympa pour les pipelines, et certains savent même benchmarker quelques exécutions pour faire des binaires où la prédiction de branches est précise pour aller plus vite.
  - Ce n'est plus seulement une solution hardware, mais faut améliorer les compilateurs
  - TODO: manque la notion de writeback ici.
- **Multicoeur**: puisqu'on peut plus faire d'unité plus rapide, multiplions les unités
  - idée inverse de la fréquence: deux coeurs underclockés consomment autant qu'un seul coeur, mais produisent 150% de calculs
  - c'est une solution terriblement software, car ça demande de reprendre l'intégralité des programmes. Peut-être aussi d'inventer des langages où le parallélisme est moins méchant, mais je m'égare.

- Toujours cette vue simplifiée, idéalisée
  - Caches partagés entre les cores → pb de cohérence
  - NUMA (non uniform mem archi): mémoire plus ou moins proche: on connecte des cartes mères.
  - Intel/AMD voudraient fondre des systemes tjs plus gros (SoC: System on Chip) mais on a du mal à les programmer
  - Multicore hétérogène: le processeur Cell (2005) pour Playstation3: un PowerPC peu puissant avec 8 gros coeurs type GPU. De beaux jouets, mais ultra durs à programmer efficacement. La PS4 a un design plus classique (CPU + GPU classiques sur même puce)



- Pour aller plus loin, on pourra se référer à un cours complet de HPC, par exemple `http://polaris.imag.fr/arnaud.legrand/teaching/2015/M2R_PC.php`

III.3) Un ordinateur en papier : M99

- On distribue l'activité M99, on lit ensemble les exos en discutant de ce qu'il faudrait faire, mais sans le faire (pas le temps), puis on saute à la partie "Réalisme" à la fin pour bien insister sur le fait qu'un ordinateur fonctionne comme ça:
  - La mémoire est un grand tableau de cases de largeur 1 octet
  - Les variables sont à coté du programme, dans un ordre déterminé par le compilateur
- Les élèves sont libres de faire l'exo chez eux ou pas, comme ils veulent.