Université Libre de Bruxelles
École polytechnique de Bruxelles

Master en Sciences de l'ingénieur, orientation ingénieur civil
Section Informatique

ELEC-H-473

Résumé

June 2, 2014.

Vincent Radelet
William Chelman

Contents

1 Introduction **5**

2 IC manufacturing technology perspective **5**
2.1 Porte NOT 5
2.2 Scaling 5

3 Computing systems performance **6**
3.1 Performance 6

4 Example of poor usage: Data Centers **6**

5 What could happen in the future **7**

6 Basic computer architecture concepts **7**

7 Minimum system architecture **9**

8 Instruction execution cycle **9**

9 ISA **10**

10 RISC vs CISC **11**

11 Instruction execution cycle **11**

12 Pipeline execution **12**

13 Pipeline hazards **13**
13.1 Name depedency 13
13.2 Data depedency 14
13.3 Control dependency 15
13.4 Resource conflicts 15

14 Loop unrolling **16**

15 Memory technology **16**

16 Memory organisation **18**

17 Memory hierarchy **19**

18 Cache principals **20**

19 Virtual memory **23**

20 Data parallelism **23**

21 Classification **24**
21.1 Flynn's classification 24

22 SIMD Intel x86 **26**

23 How to use SIMD in Intel CPU? **27**

24 Sandy and Ivy **27**
24.1 Core architecture 29
24.2 Power and thermal management 31

List of Figures

1 Porte logique NOT . . . 5
2 Délais . . . 5
3 Gain de puissance dissipée . . . 6
4 Architectures . . . 7
5 Relation CPU-Mémoire . . . 8
6 Vue rapprochée du CPU . . . 8
7 Micro architecture . . . 9
8 Addressing Mode . . . 10
9 Suite d'instructions sans pipeline . . . 11
10 Subdivision joints par Flip-Flop . . . 12
11 Suite d'instructions avec pipeline . . . 12
12 Pipeline stall . . . 13
13 Dépendance en noms . . . 14
14 Forwarding . . . 14
15 Out-of-Order . . . 15
16 For subdivision . . . 16
17 Bistable . . . 16
18 SRAM . . . 17
19 DRAM . . . 17
20 Comparaison des mémoires volatiles . . . 17
21 Comparaison des tailles et temps d'accès des mémoires . . . 18
22 Comparaison génération DDR . . . 18
23 Alignement . . . 18
24 CPU and Memory frequencies . . . 19
25 Cache system . . . 20
26 Direct mapping . . . 21
27 Block Identification . . . 21
28 2 ways . . . 22
29 Cache performance . . . 22
30 Data packing . . . 24
31 Data alignment example . . . 24
32 SIMD chart . . . 25
33 Performance ou aisance de programmation . . . 27
34 Planar vs. Multi-gate . . . 27
35 SoC . . . 28
36 System agent . . . 29
37 Sandy Bridge pipeline . . . 30
38 Front end close up . . . 30
39 Cache Hierarchy Overview . . . 31
40 Sleep States . . . 32
41 C-States . . . 32
42 P-States examples . . . 33
43 Turbo Boost . . . 33
44 Consommation au cours du temps . . . 33

1 Introduction

2 IC manufacturing technology perspective

2.1 Porte NOT

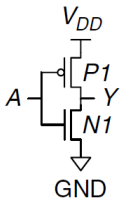


Figure 1: Porte logique NOT

2.2 Scaling

Principe

Réduction de la taille des transistors afin d'en faire tenir le plus possible sur une même surface et ainsi gagner en performances.

Effets collatéraux

Apparation du *More than Moore paradigm* voire inversion des propriétés de scaling :

Coûts Coûts et durée de production augmentant avec diminution de la taille (Les moyens de production n'évoluent pas aussi vite que la technologie). Le coût des IC (circuits imprimés), est calculé comme $\frac{\text{Moule+Test+Packaging}}{\text{Rendement du test final}}$ qui est en constante augmentation avec le temps.

Gain Les temps de commutation des portes diminuent mais pas ceux de propagation dans les fils, on calcule donc rapidement mais la transmission de l'information est lente (cf. figure 2).

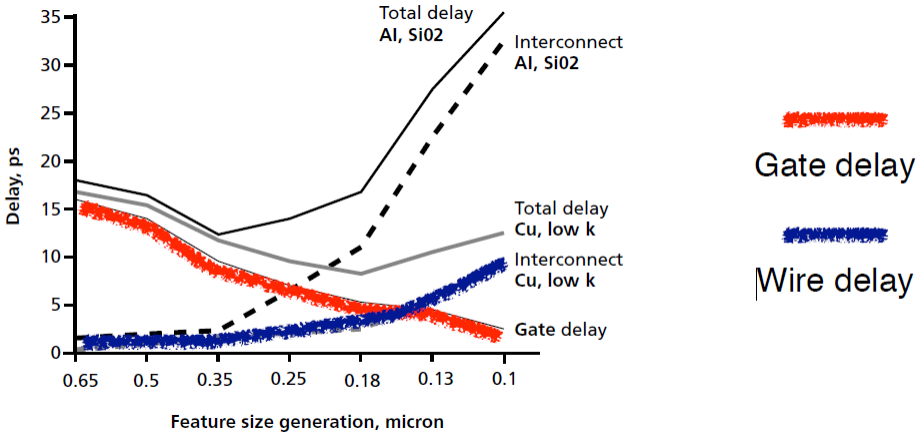


Figure 2: Délais

Puissance On constate un switch dans les types de puissances dissipées et les gains de puissances dissipées ne sont plus aussi intéressants qu'avant (cf figure 3). Au final, la fréquence des CPU n'augmente plus, on

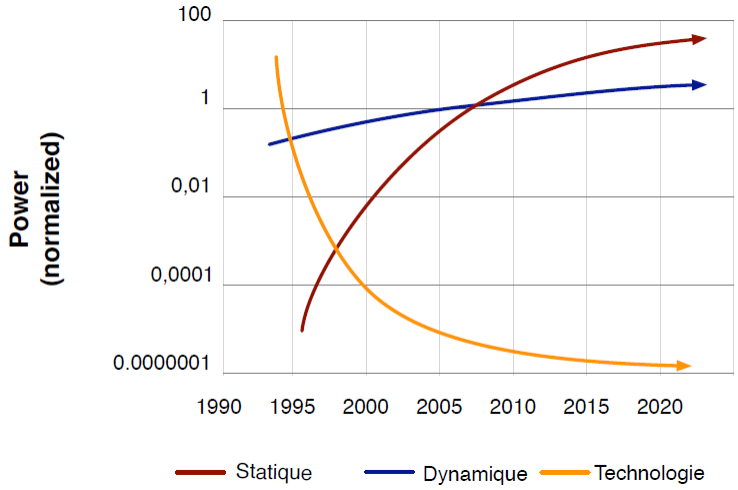


Figure 3: Gain de puissance dissipée

finit donc par augmenter le parallélisme. Malheureusement, la fréquence des CPU a augmenté bien plus vite que celle des bus, ainsi le parallélisme est lui-aussi limité.

3 Computing systems performance

3.1 Performance

Clock cycle (Clk)

Le cycle d'horloge est présent car les CPU sont des circuits logiques synchrones dont les états sont stockés dans des flip-flops. Le Clk est utilisé comme horloge commune pour tous les flip-flops.

On peut mesurer le nombre de cycles requis pour exécuter toutes les instructions d'un programme ainsi que le nombre d'instructions exécutées et ainsi calculer le CPI (*Cycles per instruction*) moyen du programme. Pour chaque CPU, on peut avoir une *data sheet* reprenant les CPI par opération de base (addition, multiplication, etc).

ICP (*Instructions per cycle*) est l'inverse du CPI et se calcule a posteriori pour une application. S'il est plus grand que 1, on a du parallélisme.

Le temps CPU est calculé comme le produit du CPI, du nombre d'instruction et du Clk. Ainsi, pour augmenter les performances, il faudrait augmenter la fréquence du CPU, mais on devient limité de ce point de vue là. On peut aussi reduire le CPI, il faut donc plus de parallélisme inter et intra CPU ou réduire le nombre d'instructions exécutées (algorithmes, SIMD, ...)

4 Example of poor usage: Data Centers

Globalement, les *data centers* utilisent des processeurs classiques multi-usages alors qu'ils font très peu de processing mais surtout du transfert de données, ainsi une architecture classique de processeur ne convient pas. De plus, les processeurs ne communiquent pas assez entre eux pour justifier un cache commun important.

Un seul type d'architecture globale pour tous types d'ordinateur était la seule solution viable économiquement et a plutôt bien marché pendant un certain temps. Nous sommes bloqués au niveau des fréquences, il est maintenant nécessaire d'avoir des architectures dédiées aux différents types d'applications.

5 What could happen in the future

voir slides.

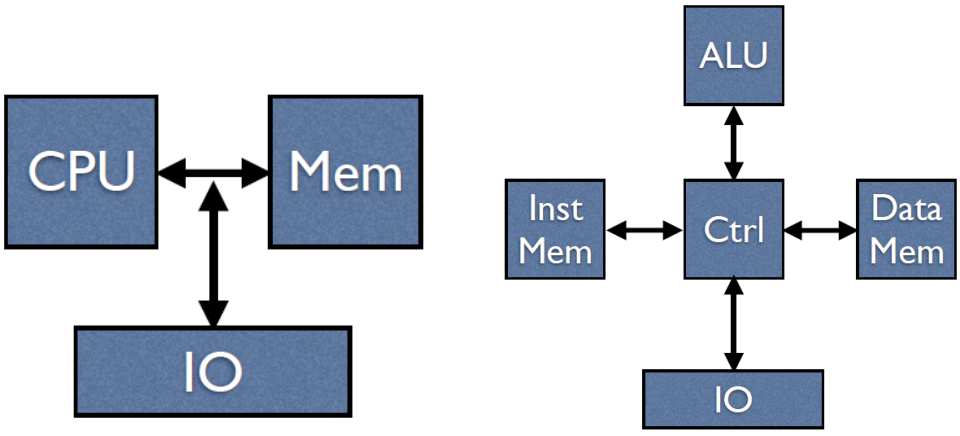
6 Basic computer architecture concepts

Il existe deux architectures prédominantes: von Neumann et Harvard. Cependant, les machines modernes ont une architecture d'Harvard modifiée qui cumulent les avantages des deux autres.

von Neumann Ce type d'architecture ne possède qu'une seule mémoire qui contient les instructions et les données. Le CPU se constitue de l'ALU, l'UC et des registres. Elle souffre d'une perte d'efficacité liée au fait qu'on ne peut pas lire ou écrire des instructions et des données simultanément. La communication entre les différentes unités se fait par un bus commun. La différence entre un bus et une connexion point à point est l'aspect collectif d'une ressource partagée comparativement à un médium de transmission exclusif.

Harvard Dans l'architecture d'Harvard, la mémoire d'instruction est scindée de celles des données. Cette architecture permet un parallélisme entre le chargement des données et de l'instruction. L'introduction de pipelining permet une accélération des calculs.

Harvard modifié Dans cette architecture, les instructions sont stockées comme étant des données. Ainsi, nous avons un système dont les instructions peuvent être réécrites.



(a) von Neumann (b) Harvard

Figure 4: Architectures

Les données et les instructions étant codées en binaire dans les mémoires, l'ordinateur ne fait aucune distinction entre ces deux types. Les mémoires sont des tables de correspondances associant une adresse à une donnée. Dans un système à n bit d'adressage, nous avons un espace de 2^n adresses pointant chacune vers m bits (typiquement 8). Il y a donc m câbles en sortie de mémoire, permettant donc un chargement parallèle des bits. Ceci détermine la bande passante de la mémoire.

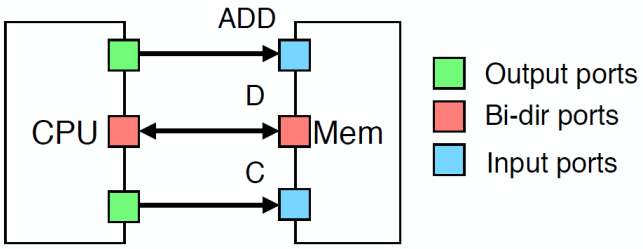


Figure 5: Relation CPU-Mémoire

Les données pouvant être écrites ou lues, les m bus d'accès sont bi-directionnels, ce qui n'est pas le cas des bus d'adressage et de contrôle (servant à spécifier "lecture" ou "écriture" sur la mémoire). La convention courante est de considérer l'input pour l'écriture et l'output pour la lecture. La vision adoptée est donc celle de la mémoire. De plus, il faut que la vitesse d'accès à la mémoire soit adaptée au nombre d'instructions par seconde que peut exécuter le CPU. Le cas échéant, elle en limiterait l'efficacité.

Les connexions entre le CPU et la mémoire étant en cuivre, il se forme un circuit RLC parasite dont les effets néfastes sont perceptibles à haute fréquence. De ce fait, la bande passante de cette connexion est impactée par la réalité géométrique du médium. Nous savons que la résistance d'un matériaux donné est fonction de ces paramètres ($R = \rho\frac{L}{S}$). Il faudra veiller à disposer les éléments d'architectures de la façon la plus compacte possible. De plus, il faut tenir compte de la vitesse de propagation de l'information dans le câble. Ainsi, la longueur du câble plafonne la fréquence.

La relation entre le CPU et la mémoire est de type Maître-Esclave. Le maître peut initialiser des opérations sur un ou plusieurs esclaves alors que ceux-ci ne peuvent qu'accepter ou refuser une opération. Les mémoires ne peuvent donc pas initialiser un transfert spontanément. Cependant, l'initialisation de transfert est couteuse en temps et ralenti donc le CPU. Les machines actuelles ont trouvé une alternative à ce problème qui sera étudiée plus en détail ultérieurement.

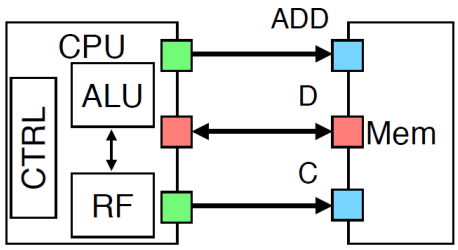


Figure 6: Vue rapprochée du CPU

Comme mentionné précédemment, le CPU se décompose en trois entités

1. L'unité arithmétique et logique (ALU) qui est en charge des aspects calculatoires du déroulement.
2. Les registres qui contiennent les opérandes nécessaires à l'ALU.
3. L'unité de contrôle qui joue le rôle de chef d'orchestre.

Les registres sont typiquement des SRAM, des mémoires ne pouvant effectuer qu'une opération d'écriture ou de lecture par cycle. Ils sont généralement multi-ports ce qui permet à des données multiples d'être accédées en parallèle. Bien qu'il puisse il y avoir plusieurs ports de lecture, il n'y a cependant qu'un seul port d'écriture afin de se prémunir d'éventuels conflits lors de tentatives d'écriture concourantes. Si ce n'était

pas le cas, si deux écritures concourantes se faisaient à la même adresse, la seconde opération écraserait la première. Même dans le cas de systèmes synchrones, les temps de propagations et autres délais induisent toujours un léger décalage.

7 Minimum system architecture

Les ordinateurs ne peuvent, toutefois, pas se contenter de ces unités de bases. Il leur faut bien une architecture de plus haut niveau pour gouverner le tout. En particulier, l'architecture globale d'un ordinateur englobe la partie logicielle (système d'exploitation).

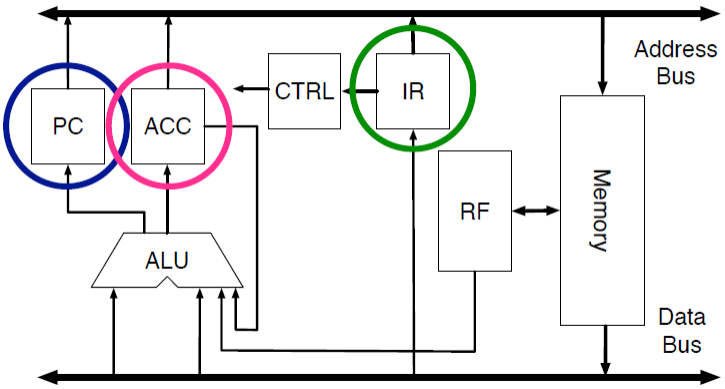


Figure 7: Micro architecture

L'architecture minimum qui englobe le système étudié précédemment est appellé micro-architecture. Elle conserve l'adresse qui contient l'instruction suivante dans le PC (Program Counter), l'instruction en cours dans l'IR (Instruction Register) et introduit des registres dédiés au stockage des résultats de calcul de l'ALU, l'ACC (Dedicated register). L'utilisation de registres permet d'introduire un pipelining augmentant l'efficacité. En effet, cela permet une utilisation plus intensive des composantes et de maximiser leur temps d'activité.

L'ALU possède à présent trois sources de données potentielles. Elle peut se baser sur le résultat de la dernière opération effectuée en chargeant le contenu de l'ACC, se servir du RF comme d'un proxy ou aller chercher directement les données en mémoire. On préfèrera bien évidemment éviter ce dernier cas de figure au profit de l'utilisation du RF (*Register File*).

8 Instruction execution cycle

D'un point de vue général, un programme est une liste d'instructions lues et exécutées une par une de façon séquentielle. Le traitement d'une instruction peut se découper en 5 étapes.

1. **Fetch instruction** : Rapatrier l'instruction dans l'IR.
   - L'UC met l'adresse contenue dans le PC sur le bus d'adressage.
   - L'UC ordonne la lecture de cette adresse en mémoire qui écrit le résultat sur le bus des données.
   - L'IR récupère l'information écrite sur le bus des données.
2. **Decode instruction** : Traduire l'instruction en terme de signaux de contrôle.

- L'UC Charge le contenu de l'IR.
- L'UC Traduit cette instruction en signaux appropriés et sollicite l'ALU.

3. **Fetch operand** : Rapatrier les données relatives aux opérandes dans les registres si ces dernières ne s'y trouvaient pas déjà.
4. **Execute instruction** : Effectue l'instruction
   - L'ALU effectue l'opération
   - Le résultat est écrit dans l'ACC
   - Le PC est mis à jour en concordance avec le résultat (typiquement un incrément unitaire, mais il peut s'agir de JUMP).
5. **Write result** : Ecrit le résultat en mémoire ou dans un registre selon le type d'architecture utilisée et de l'instruction qui peut potentiellement spécifier la destination du résultat.

Nous pouvons distinguer 3 types d'architectures basée sur des mémoires RW.

1. **Load/Store** : Les opérandes doivent impérativement se trouver dans les registres. Tout calcul est précédé d'opération de lecture et/ou d'écriture entre le registre et la mémoire.
2. **Register/Memory** : Les opérandes peuvent se trouver en mémoire ou dans les registre. Il s'agit, ici, d'un ou exclusif.
3. **Register+Memory** : Les opérandes se trouvent dans les registres et/ou la mémoire.

9 ISA

L'ISA (Instruction Set Architecture) est le lien entre la micro-architecture et le programmeur (e.g l'assembleur ou le compilateur). L'ISA ne détermine donc pas la façon dont la machine fait les choses, mais bien ce qu'elle fait. Un ordinateur est donc la mise en commun de la micro-architecture et de l'ISA. Le set d'instructions exprime les mouvements de données (chargement, sauvegarde, copie, I/O), l'arithmétique (entier, virgule flotante), le décalage (n bits vers la gauche/droite), logique (AND, NOT), le contrôle (jump, branch), appel de sous routine (call, return) et les interruptions.

Addressing mode	Example	Meaning
Register	Add R4,R3	R4 ← R4 + R3
Immediate	Add R4,#3	R4 ← R4 + 3
Displacement	Add R4,100(R1)	R4 ← R4+Mem[100+R1]
Register indirect	Add R4,(R1)	R4 ← R4+Mem[R1]
Indexed / Base	Add R3,(R1+R2)	R3 ← R3+Mem[R1+R2]
Direct or absolute	Add R1,(1001)	R1 ← R1+Mem[1001]
Memory indirect	Add R1,@(R3)	R1 ← R1+Mem[Mem[R3]]
Auto-increment	Add R1,(R2)+	R1 ← R1+Mem[R2] R2 ← R2 + d
Auto-decrement	Add R1,–(R2)	R2 ← R2–d R1 ← R1+Mem[R2]
Scaled	Add R1,100(R2)[R3]	R1 ← R1+Mem[100+R2+R3*d]

Figure 8: Addressing Mode

L'étude d'un code permet de déterminer la fréquence d'occurrence des différents types d'instructions. Ainsi, lorsque des systèmes embarqués ont des fonctions spécifiques, il est intéressant d'adapter l'architecture aux type d'instructions les plus représentés pour en maximiser le rendement.

10 RISC vs CISC

Toutes les instructions ne sont pas utilisées à la même fréquence, certaines le sont même assez rarement. Certaines utilisations se limitant à un très faible nombre d'instructions différentes, il est utile de créer deux set d'instructions. Le RISC est un set épuré alors que le CISC est un set plus complexe. Le choix de l'un ou de l'autre sera entièrement déterminé par le type d'utilisation à laquelle il est destiné.

RISC Le set étant plus restreint, il est beaucoup plus simple de l'optimiser de façon matérielle. Typiquement, les instructions du RISC ne nécessiteront qu'un nombre très limité de cycle (idéalement un seul). Des instructions plus complexes peuvent être obtenues sous forme de sous-routines. Il faudra cependant découper les instructions plus haut niveau en plusieurs instructions. Par exemple, les opérations dont les opérandes se trouvent en mémoire devront être découpées en chargement, exécution et écriture. Bien que les RISC soient essentiellement utilisés de façon académique on peut les retrouver dans des applications réelles telles que les smartphones, super-calculateurs et serveurs.

CISC A contrario du RISC, le CISC possède des instructions beaucoup plus complexes. Il est par exemple possible de charger, exécuter et sauvegarder en une seule instruction. Bien sûr, le coût de cette liberté est une optimisation bien plus pauvre menant à un montant de cycles par instructions beaucoup plus important.

De nos jours, nous cumulons les avantages en entremêlant les deux sets. Ainsi nous avons des architectures principalement CISC avec des sous-sets basés sur le RISC afin d'en améliorer l'efficacité.

11 Instruction execution cycle

En l'absence de pipelining, nous devons attendre que les 4 étapes de l'exécution s'achèvent avant de lancer la suivante.

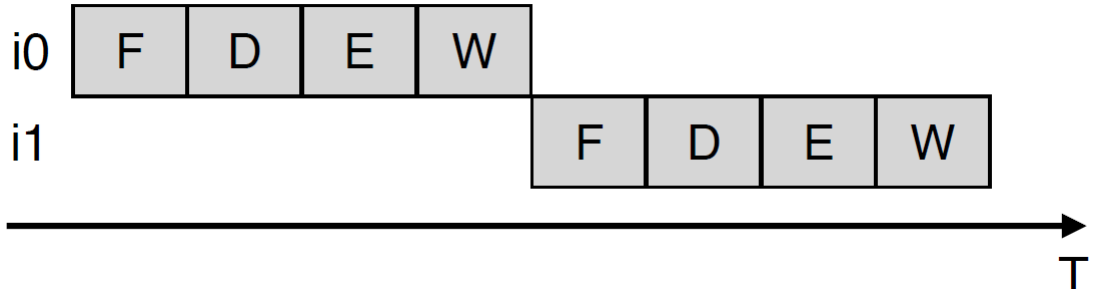


Figure 9: Suite d'instructions sans pipeline

Le chemin critique est le chemin qui contient le plus de délais (transmission, commutation des portes,...). Au plus gros est le circuit, au plus nous nous exposons à des chemins critiques de grande taille. Les petits circuits sont donc par nature plus rapide que les gros. Il est possible d'accélérer un gros circuit en le subdivisant en plusieurs petits inter connectés par des Flip-Flop.

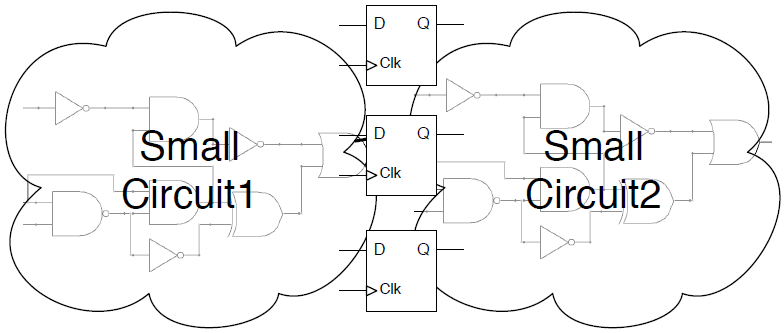


Figure 10: Subdivision joints par Flip-Flop

Les résultats obtenus après les opérations du premier circuit seront stockés dans les Flip-Flops et pourront, de ce fait, être traités lors du prochain cycle d'horloge par le circuit suivant. Nous pouvons, dès lors, introduire du pipelining car le circuit 1 peut se remettre à travailler sans avoir à attendre que le circuit 2 ait terminé de traiter sa sortie. Cependant, cette découpe en circuits ajoute du délais car un circuit prenant un cycle d'horloge en prendra autant de fois plus qu'il y a de sous-circuits. Néanmoins, le gain introduit par le pipelining justifie amplement son utilisation malgré ce fait.

12 Pipeline execution

Le pipelining permet de considérer F,D,E et W comme des circuits distincts. Cela induit donc une latence de 4 cycles dans le circuit, ce qui signifie que la sortie correspondant à un input n'est produite que 4 cycles plus tard.

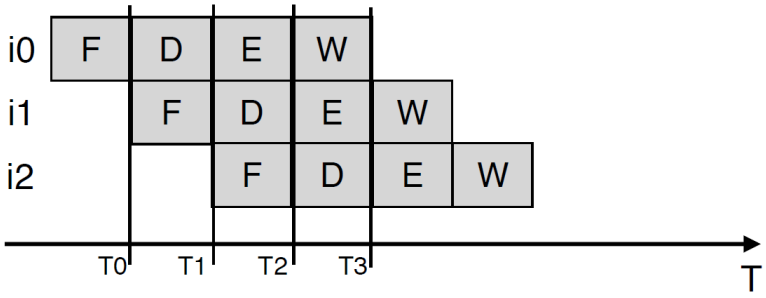


Figure 11: Suite d'instructions avec pipeline

Une fois le système en régime (3 cycles d'initialisation), chaque cycle d'horloge exécute un Fetch, un Decode, un Execute et un Write. Pour N instructions et n étapes de pipeline, le nombre de cycles requis devient $C_p = n + N \cdot t$ au lieu de $C_s = N \cdot n \cdot t$. L'accélération obtenue est donc $\frac{C_p}{C_s} = \frac{1}{N \cdot t} + \frac{1}{n}$. Nous pouvons voir que pour un grand nombre d'instructions, cette accélération tend vers $\frac{1}{n}$. Il y a donc un compromis à trouver entre le fait d'utiliser des pipelines, donc maximiser l'emploi des composant de l'architecture, et la latence qu'on introduit. Il ne serait en effet pas recevable de devoir attendre 50 cycles d'horloges pour avoir le résultat d'une instruction quand bien même l'efficacité générale du système serait accrue. En particulier, si les instructions ne sont pas indépendantes, une instruction peut bloquer le pipeline en attendant le résultat d'une autre. Malgré tout, fragmenter le chemin critique permet d'augmenter la fréquence car le cycle d'horloge dépendra alors du plus grand chemin critique parmi les sous circuits. Il existe, dès lors, des architectures

plus ou moins parallélisées dont le taux de parallélisation dépendra de l'utilisation qu'on en a. Le pipelining se fait de façon matérielle de sorte que le programmeur n'ai pas à se soucier de son implémentation.

13 Pipeline hazards

L'accélération due au pipelining, directement proportionnelle au taux de parallélisme, est uniquement possible si toutes les étapes prennent un temps identique, fixe et pré-déterminable et qu'il n'y a aucune dépendance entre les instructions successives. Au mieux, on ne peut dépasser un rendement de une instruction par cycle.

Bubbling Un problème d'exécution introduit une bulle dans le pipeline. Ces problèmes peuvent apparaitre quand des instructions ne prennent pas le même temps, quand une situation imprévisible se produit ou quand il y a une dépendance calculatoire entre 2 instructions.

Pipeline stall Dans le cas où une instruction serait trop longue (nécessite plus d'un cycle d'horloge), l'UC peut déterminer la présence de ce problème et insère des NOP (No OPeration) dans la chaîne d'instructions. Ainsi, un NOP est juste une attente. S'il y a plus de NOP que d'étapes de pipelining, on dit qu'il est *flushed*.

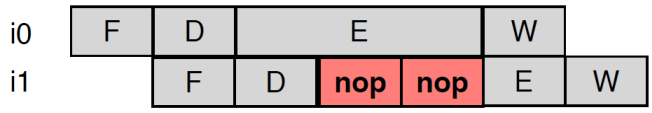


Figure 12: Pipeline stall

Le ordinateurs exécutent, bien généralement, moins d'une instruction par cycle. Pour améliorer le rendement, il est donc nécessaire de comprendre ce qui induit du stalling. Nous pouvons en regrouper les causes en 4 sections majeures:

- Name dependency
- Data dependency
- Control dependency
- Ressource conflicts

13.1 Name depedency

On parle de dépendance en noms lorsque deux instructions distinctes souhaitent effectuer une opération sur le même registre de source ou de destination. Il ne s'agit pas réellement de dépendance car il n'y a pas de séquences de calcul reliant ces deux instructions. On parlera néanmoins d'anti dépendance lorsque l'instruction $i+1$ écrit sur le registre qu'utilise l'instruction i et de dépendance en sortie lorsqu'elles essayent d'écrire sur le même registre de destination. Lorsque de telles dépendances sont détectées, il s'agit simplement de redistribuer les instructions sur d'autres registres. L'ISA contiendra donc les adresses des registres afin de les rendre accessibles nominativement. Bien qu'il puisse s'agir d'une relation 1-1 avec l'adresse physique établie qu'une seule fois au niveau matériel, il est aussi possible d'utiliser une table dynamique (Register Alias Table - RAT) afin de pouvoir faire des renoms à chaud. Les noms statiques peuvent être utiliser par le programmeur et le compilateur alors que les noms dynamiques ne sont utilisés que par le CPU lors de l'exécution.

```
--
1. MOV     CX, [Mem1]
2. ADD     CX, BX
3. MOV     CX, [Mem2]
4. ADD     CX, BX
```

```
--
1. MOV     CX, [Mem1]
2. ADD     CX, BX
3. MOV     JX, [Mem2]
4. ADD     JX, BX
```

Figure 13: Dépendance en noms

Dans le cas illustré ci-dessus, nous pouvons voir qu'il n'était initialement pas possible d'introduire du parallélisme car les deux instructions souhaitaient travailler avec le registre C_x. En déviant la seconde addition sur un autre registre, nous retrouvons la possibilité d'exécuter ces deux opérations simultanément. Avec un adressage dynamique, cette redirection peut être faite à la volée.

13.2 Data depedency

Considérons les deux opération suivantes : $C = A/B$ et $D = A + C$. La dépendance aux données exprime le fait qu'il faille attendre que la division ai été effectuée avant de pouvoir calculer la somme. Outre les bulles générées dans le pipeline lors de ce cas de figure, nous sommes exposés à trois autres risques.

1. **Read after Write** Une instruction lit le contenu d'un registre alors qu'il a été modifié par une autre instruction. Ce problème est courant dans tous les types de pipeline.

2. **Write after Read** Une instruction modifie le contenu d'un registre avant qu'une autre instruction n'ai eu le temps de le lire. Ce problème s'applique uniquement dans les architectures dotées de plusieurs ALU.

3. **Write after Write** Deux instructions écrivent dans le même registre; la seconde écrase, de ce fait, l'écriture de la première. Ce risque ne s'applique, lui aussi, qu'au architectures dotées de plusieurs ALU.

Ainsi, quand bien même nous utilisons du matériel capable d'effectuer une division en un cycle d'horloge, il persiste une bulle dans le pipeline car il faut attendre l'écriture du résultat dans le registre. Cependant, il est possible d'aller chercher directement le résultat à la sortie de l'instruction précédente plutôt qu'à son registre d'écriture. Cette solution est appelée *Forwarding*.

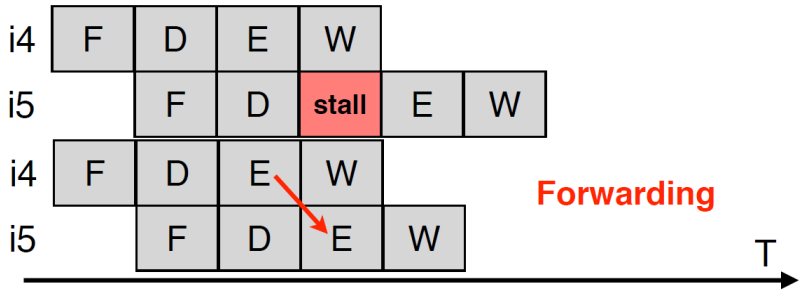


Figure 14: Forwarding

Deux instructions sont dites indépendantes si elles peuvent être exécutées simultanément par deux ALU distinctes. De telles instructions n'introduiront pas de bulles dans le pipeline, nous chercherons donc à

organiser les calculs de sorte que les opérations parallèles soient indépendantes. Néanmoins, il reste important que des instructions dépendantes en données soient proches pour que le résultat soit disponible aussi tôt que possible. Il existera donc un ordonnancement minimisant le stalling et maximisant, par voie de conséquence, l'exécution du programme. Une micro-architecture suivant l'ordre des instructions imposé par le programme est appellée *in-order*. Pour ce type d'architecture, il incombera au programmeur ou au compilateur d'optimiser l'ordonnancement. Un effet collatéral est qu'un code ne pourra être optimisé que pour une architecture donnée. Il est donc intéressant de créer du matériel capable d'exécuter les instructions *out-of-order* pouvant modifier l'ordonnancement en fonction de l'architecture sur laquelle est supposée tourner le code. En effet, ce matériel permet d'affranchir le programmeur et le compilateur de l'ordonnancement. La complexité et le coût de tels matériels sont, bien évidemment, plus élevés. Dans l'exemple ci-dessous, les trois premières instructions sont dépendantes en données alors que les trois suivantes sont indépendantes. Une optimisation est donc possible en déplaçant une des soustractions.

```
--
DIV R2, R1
ADD R3, R2
MUL R4, R3
SUB R8, R7
SUB R10, R9
SUB R12, R11
```

Cycle	1	2	3	4	5	6	7	8
Div R2, R1	Fetch	Decode	Execute			Write		
Add R3, R2		Fetch	Decode	Wait		Execute	Write	
Sub R8, R7			Fetch	Decode	Execute	Write		
Mul R4, R3				Fetch	Decode	Wait	Execute	Write
Sub R10, R9					Fetch	Decode	Execute	Write
Sub R12, R11						Fetch	Decode	Execute

Figure 15: Out-of-Order

13.3 Control dependency

Ce type de dépendance est induit par la possibilité d'effectuer des branchements. Ceux-ci peuvent être inconditionnels (e.g `GOTO`) ou conditionnels (e.g `IF`). Lors d'un branchement, le flux de contrôle est bouleversé et nous démarrons une autre séquence d'instructions. Les ISA incluent donc la possibilité de faire des sauts conditionnels (JZ : Jump if Zero, JNZ : Jump if Not Zero) à des endroits du codes labellisés. Un usage excessif de saut non conditionnels engendre un code illisible, difficile à déboguer, que l'on référence comme du code spaghetti. Lors d'un test de condition, `IF`, le PC est incrémenté d'une unité si la condition est vérifiée (fonctionnement normal). Si ce n'est pas le cas, le PC saute à la ligne suivant le bloc du `IF`. Du fait de ce fonctionnement, on ne peut pas *fetch* l'instruction suivante avant de connaitre le résultat du `IF`. Il y a 2 solutions pour gérer ces cas :

- La manière la plus simple est de *flush* le pipeline jusqu'à ce que le résultat du `IF` soit connu.
- Une autre méthode est de *fetch* la suite des instructions sans attendre le résultat du `IF` en espérant qu'il soit vérifié, si ce n'est pas le cas on *flush* le pipeline et on reprend le PC à l'endroit voulu.

13.4 Resource conflicts

Jusqu'à maintenant, on ne considérait le cas que d'une seule unité d'exécution (ALU). Idéalement, on peut penser une architecture possédant plusieurs unités d'exécution n'ayant pas forcement les mêmes fonctionnalités. Ainsi, 2 instructions indépendantes peuvent s'exécuter sur 2 unités d'exécution différentes, ce qui permet d'éviter les *pipeline stall*. Nous arrivons donc à un système ayant de multiples pipelines d'exécution, et donc avec la possibilité d'avoir un IPC plus grand que 1. On finit par arriver à un résultat encore meilleur avec un meilleur ordonnancement automatique ou un code mieux écrit. Il faut faire attention, si nous avons 2 ALU dont une seule est capable de `MULT` et que nous souhaitons effectuer 2 de ces opérations d'affilée, il y aura de nouveau des problèmes de *pipeline stall*.

14 Loop unrolling

Le but ici est d'augmenter la probabilité de possibles instructions se chevauchant afin d'aider l'ordonnanceur à éviter les *pipeline stall*. Par exemple, la boucle suivante est subdivisée afin que, si un délais est présent sur l'instruction, on puisse ne pas multiplier ce délais par le nombre d'itération.

```
for(i=0; i<1000; i++)
    A[i] = A[i] + B[i];
```

```
for(i=0; i<250; i++) {
      A[i + 0*250] = A[i + 0*250] + B[i + 0*250];
      A[i + 1*250] = A[i + 1*250] + B[i + 1*250];
      A[i + 2*250] = A[i + 2*250] + B[i + 2*250];
      A[i + 3*250] = A[i + 3*250] + B[i + 3*250]; }
```

Figure 16: For subdivision

Nous avons maintenant 4 calculs indépendants qui peuvent renommer des registres, qu'on peut ordonner pour minimiser le *pipeline stall* et si on est dans la situation d'une architecture super-scalaire, nous avons la possibilité d'exécuter les 4 instructions sur 4 ALU différents.

15 Memory technology

Nous commençons par introduire le *bit-cell*, le plus petit élément de mémorisation, ne contenant qu'un bit. Pour construire des structures de mémoires plus grandes, nous utilisons des tableaux de mémoire 2D composés de n lignes et m colonnes. Chaque ligne est identifiée par une adresse unique et une seule ligne peut être accessible en R/W à la fois.

Bit-cell Il y en a deux classes principales : les volatiles et les non-volatiles. Parmi les volatiles, on retrouve principalement les Flip-Flop (stockage basique pour tout circuit logique séquentiel, le plus rapide des trois mais la plus chère en terme de place), les SRAM (Static Random Access Memory, on parle de Random car quelle que soit l'adresse spécifiée, son accès prend le même temps, le rapport performance/prix étant intermédiaire) et les DRAM (Dynamic Random Access Memory, la moins chère en terme de coût par bit mais la plus chère en terme de temps d'accès).

Latches & Flip-flops Un bistable est un circuit séquentiel possédant seulement 2 états stables qui sont contrôlables par ses entrées. Si celui-ci est sensible à un palier, on parle de *latch*. Au contraire, s'il est sensible aux flancs, on parle de *flip-flop*.

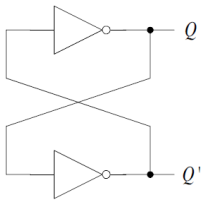


Figure 17: Bistable

SRAM cell Le bit d'information est stocké dans des inverseurs à couplages croisés. Cette cellule possède 3 états : IDLE, READ ou WRITE. Deux transistors nMOS contrôlent les états R/W : quand *wordline* est actif, les deux transistors sont ON et l'état *bitline* est transféré depuis/vers le couple d'inverseurs (plus

précisement, pour l'écriture on écrit d'abord sur bitline puis on active *wordline*, et pour la lecture on active *wordline* et on lit la valeur stocké sur *bitline*). On parle d'auto-rafraichissement car la SRAM est immunisée aux bruits grâce aux inverseurs qui peuvent restaurer la valeur qui a été possiblement perdue. On parle de configuration 6T car il y a 6 transistors : en plus des 2 de contrôle, il y en a 2 par porte NOT.

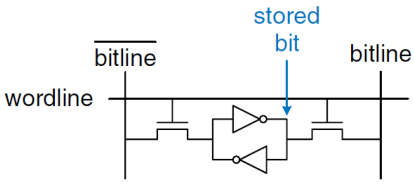


Figure 18: SRAM

DRAM cell Lorsque wordline est actif, donc le nMOS, le bit est stocké depuis le bitline vers la capacité (WRITE) ou placé sur le bitline depuis la capacité (READ). La capacité se vidant au cours du temps, la donnée est perdue au fil du temps. De plus, l'opération de lecture vide la capacité, supprimant ainsi son information. Nous parlons donc de mémoire dynamique car elles nécessitent un rafraichissement régulier même lorsqu'aucune lecture n'est effectuée.

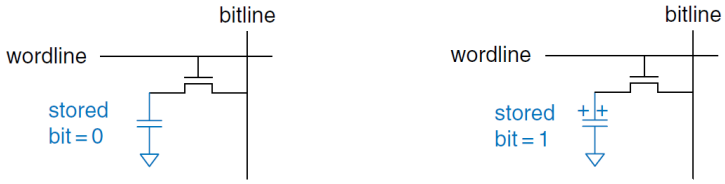


Figure 19: DRAM

	FF	SRAM	DRAM
Transistor count / bit	≈ 20	6	1
Speed(ns)	< 0.5	$\approx 1 - 10$	100

Figure 20: Comparaison des mémoires volatiles

Il est à noter que les temps d'accès d'une SRAM sont directement liés à sa taille. Une grande mémoire SRAM sera donc lente par nature.

De nos jours, les PC utilisent des DDRs DRAM (**D**ouble **D**ata **R**ate **s**ynchronous **D**ynamic **R**andom **A**ccess **M**emories). On parle de double taux car elles sont sensibles aux fronts montant et descendant.

	Reg File	L1	L2	L3	Main	Disk
Typical size	< 1kB	< 256 kB	< 2 MB	< 24 MB	< 512 GB	> 1 TB
Access time (ns)	< 0.5	$0.5 - 25$			< 250	$5 \cdot 10^6$
Managed by	compiler/core	HW			OS	

Figure 21: Comparaison des tailles et temps d'accès des mémoires

Names	Frequency	Transfer Rate	Max Bandwidth
DR	100 MHz	200 MT/s	1.6 GB/s
DDR2	400 MHz	800 MT/s	6.4 GB/s
DDR3	800 MHz	1600 MT/s	12.8 GB/s

Figure 22: Comparaison génération DDR

Comme nous l'avons vu, nous ne pouvons pas augmenter infiniment la fréquence de nos mémoires à cause des effets néfastes des circuits RLC. Une tendance actuelle est d'empiler les DRAMs au dessus d'un étage qui sert d'unité de contrôle pour la mémoire. On parle alors d'*Hybrid memory cube*.

16 Memory organisation

Comme énoncé précédemment, les adresses mémoires pointent vers m bits. Ce nombre résulte d'un compromis entre le coût et la performance fait au niveau de l'architecture. Ainsi, le choix de ce nombre au niveau matériel défini la taille d'une unité de mémoire au niveau logiciel. Les allocations mémoires seront, dès lors, faites en multiple de m. On utilise également le vocable *mot* pour désigner les m bits contenus à une adresse. La nature protocolaire de la relation entre le CPU et la mémoire entraine un ralentissement à chaque transfert, on cherchera donc a maximiser le nombre de données rapatriées par accès mémoire. Nous chercherons également à minimiser le nombre d'accès en mémoire nécessaires pour rapatrier un mot. Une adresse est dite alignée si elle est un multiple de la taille du mot. Ainsi, l'accès à une donnée peut être fait en un seul rapatriement. L'utilisation d'adresse non alignée nécessitera le rapatriement des deux blocs sur lesquels le mot est à cheval. Pour favoriser l'alignement des adresses, lorsqu'un type de donnée est plus court que la taille d'un mot l'architecture comble automatiquement le vide en mettant des *dummy values*. On parle de *data structure padding*. Pour illustrer ceci, considérons une structure comprenant 4 char, 1 double et 1 int, soit 16 bytes. Si nous ne respectons pas l'alignement, il nous faudra un accès supplémentaire pour rapatrier cette structure sur un bus de 64 bits (8 bytes).

Type	Bytes	Allocated	Total
char	1	8	24
double	8	8	
int	4	8	
char	1		
char	1		
char	1		

(a) Adresses non alignées

Type	Bytes	Allocated	Total
double	8	8	16
int	4	8	
char	1		
char	1		
char	1		
char	1		

(b) Adresses alignées

Figure 23: Alignement

Il n'est, en effet, pas possible de véhiculer un char et un double simultanément sur un bus de 64 bits. Il faudra donc rapatrier le char de façon isolée ce qui représente une perte d'efficacité en plus d'une perte de

mémoire (utilisation d'un espace de 8 bytes pour stocker 1 byte d'information utile).

17 Memory hierarchy

Soit un registre directement connecté à la mémoire centrale (voir figure 25a). Si le CPU a besoin de données, celui-ci doit les récupérer directement de la mémoire centrale. Pour conserver une exécution équilibrée, et pour ne pas *stall* le processus ou la mémoire IO, il nous faut calculer et accéder aux données à la même vitesse, ce qui n'est malheureusement pas le cas, les fréquences de fonctionnement de la mémoire centrale n'ayant pas autant évolué que celles des CPU.

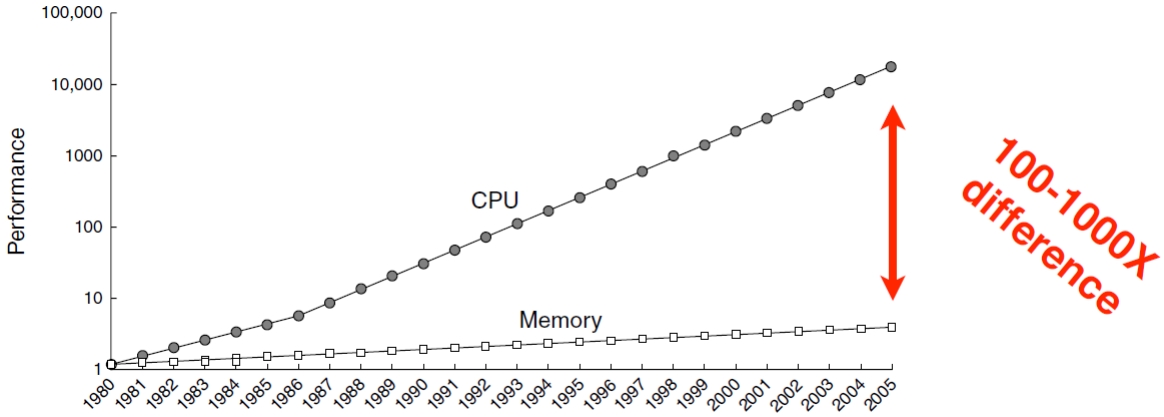


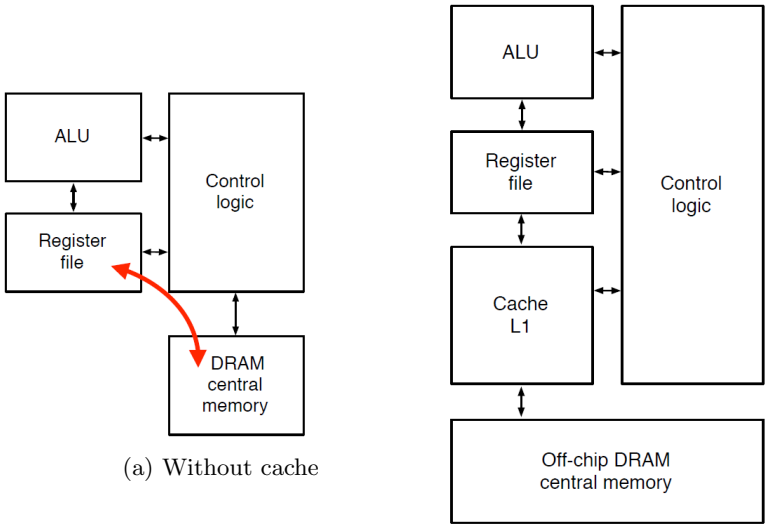
Figure 24: CPU and Memory frequencies

La solution est d'insérer des mémoires (caches) plus petites mais plus rapides sur le chemin entre le CPU et la mémoire centrale (voir figure 25b). Les caches conservent une copie de la mémoire centrale et peuvent la délivrée au CPU bien plus vite. Quand le CPU veut accéder à une nouvelle donnée, il va d'abord regarder dans le cache :

- Si le cache possède cette donnée, elle est simplement transmise au CPU (*cache hit*). On se prémunit ainsi du *stall* du CPU.
- Si ce n'est pas le cas, le cache charge la donnée nécessaire depuis la mémoire centrale et le CPU est *stalled*.

Les *cache hits* sont rendus possibles grâce aux localités temporelles et spatiales. En effet, certaines données sont utilisées plusieurs fois à des temps très proches (e.g compteur de boucle) tandis que d'autres font partie de structures qu'il est fréquent de parcourir entièrement (e.g array). Ces dernières dépendent du problème en question, de l'algorithme utilisé pour résoudre le-dit problème, de l'implémentation du programme ainsi que de la manière dont le cache est contrôlé.

De manière évidente, il faut maximiser les *cache hits* afin de ne pas avoir de pertes de performance. Malheureusement, la solution à ce problème est d'augmenter la taille des caches, ce qui va à l'encontre d'un autre objectif qui est la minimisation des coûts. Ainsi, des compromis doivent être trouvés, mais la taille de la mémoire l'emporte du à la gourmandise des applications en terme de mémoire.



(a) Without cache

(b) With L1 cache

Figure 25: Cache system

Pour mesurer la performance des caches, on calcule le Memory stall cycles = IC × Misses/Instruction × Miss cost. Les coûts d'échec dépendent du temps d'accès à la mémoire, le taux de réussite, la latence et la bande passante.

18 Cache principals

Les entrées caches sont structurées en 3 champs : le *tag* qui est une partie de l'adresse mémoire d'origine de la donnée, le bloc de donnée en lui-même et les *flags* qui en indiquent le statut (*valid* si la bonne donnée a été chargée, *dirty* si celle-ci a été modifiée par le CPU). Les transferts de données entre la mémoire centrale et le cache utilisent des *caches lines* dont la taille est laissée à la discrétion du hardware.

Direct mapping On map 2^{30} espaces de la mémoire centrale sur 2^3 espaces du cache. Les mots en mémoire centrale ainsi que leurs adresses sont codées sur 4 bytes, les 2 derniers bits d'adressage étant utilisés comme offset et les 3 bits suivant servent à l'adressage dans le cache.

Block identification Lors d'une demande de donnée, on vérifie la correspondance entre l'adresse donnée et la concaténation du tag, du numéro de registre et des 2 bits d'offset. Si un match est trouvé, l'accès à la donnée est fourni.

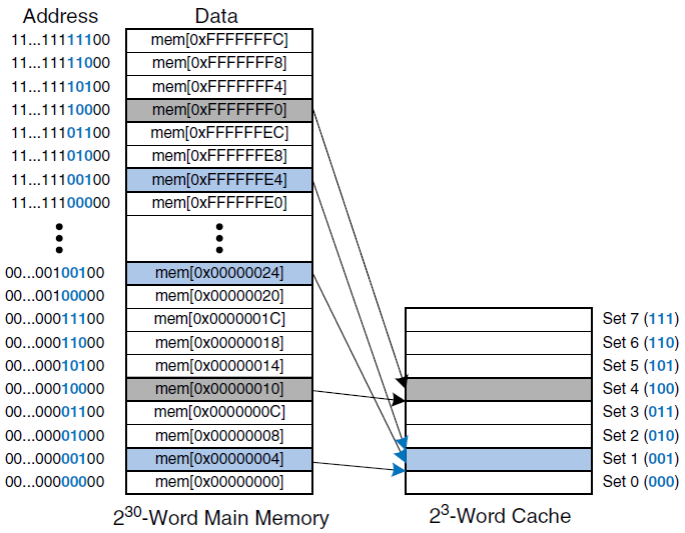


Figure 26: Direct mapping

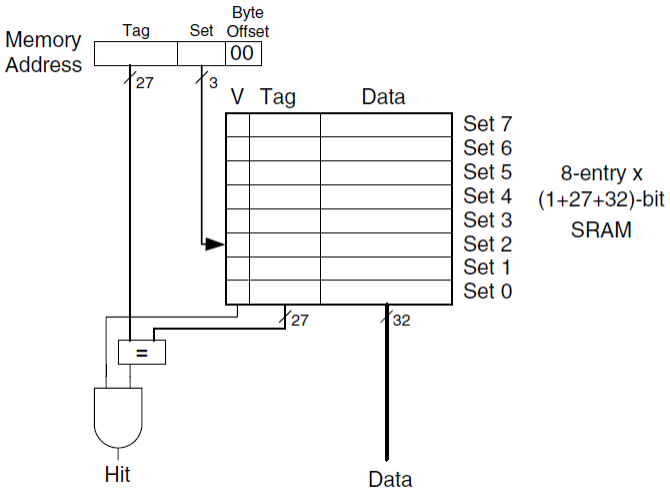


Figure 27: Block Identification

Direct mapping cache - problem Si 2 données ont des adresses qui leurs fourniraient la même entrée en cache et que celles-ci sont appelées régulièrement, elles vont s'écraser mutuellement du cache, ce qui entrainera des échecs de cache réguliers. Pour palier à ce genre de problème, il faut augmenter l'associativité des espaces mémoires, i.e augmenter le nombre d'entrées possibles où elles peuvent se trouver mapper en cache. On a ainsi 3 types d'associativité : *direct mapped cache* qui, comme on a pu le voir, a une associativité de 1, *m-set* qui peut associer m entrées différentes à un espace mémoire et *fully associate* où les espaces mémoires peuvent être associés à n'importe quelle entrée du cache. Un exemple de *2-way set associate cache* est donné ci-dessous.

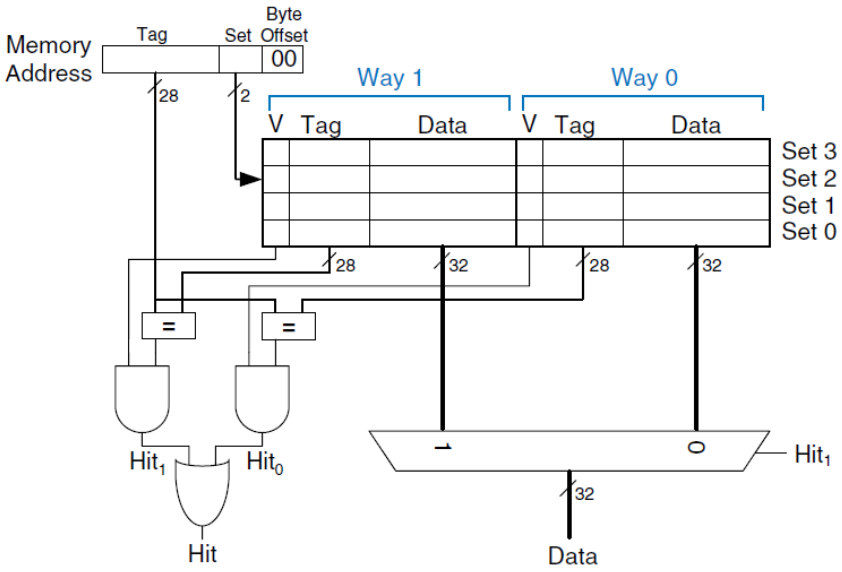


Figure 28: 2 ways

La taille du cache et son associativité sont tout deux des facteurs améliorant le taux de réussite du cache comme on peut le voir sur le graphe ci-dessous. On peut aussi remarquer que doubler l'associativité a le même effet que de doubler la taille du cache.

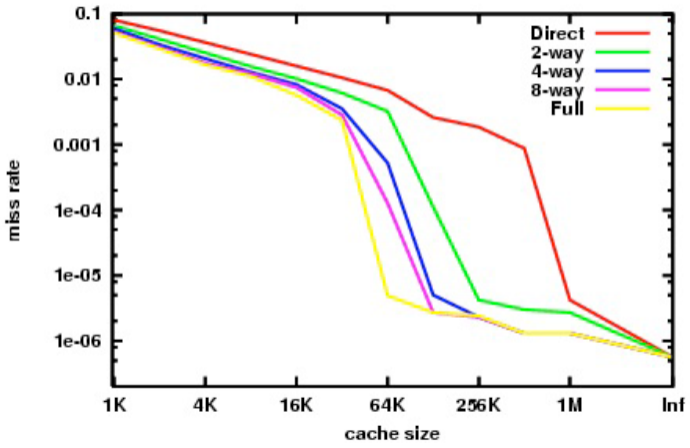


Figure 29: Cache performance

Replacement policy Plusieurs solutions sont disponibles pour choisir quelle entrée écraser : au hasard, celle qui a été utilisée la moins récemment (LRU) et en FIFO (plus simple à implémenter que LRU). Le LRU est le plus efficace à petite taille de cache (16 KB) et les différences de performances ne sont pas significatives pour des tailles de cache plus élevées.

Write policy Deux politiques d'écriture peuvent être vues : Soit dès qu'une donnée est modifiée, on répercute directement sa modification dans la mémoire, soit on attend que la valeur de donnée qui est en cache soit supprimé de celui-ci avant d'actualiser sa valeur en mémoire. Une optimisation commune est l'utilisation de buffer entre le cache et la mémoire centrale. Celui-ci est rempli par le CPU et vidé par la mémoire (en FIFO). Comme il est indépendant du CPU, il n'occasionnera pas de *stall*. Sa qualité dépend bien sûr de sa taille, qu'il est malheureusement coûteux d'augmenter. Si on n'arrive pas à le vider suffisamment vite, nous nous retrouvons en saturation et donc confronter au même problème. La solution est donc d'insérer un cache L2 qui peut se vider à la fréquence imposée par le CPU.

Il est a noter que nous avons principalement parlé de cache de données mais les mêmes principes s'appliquent aussi aux caches pour instructions, à cela près que celles-ci sont uniquement en lecture.

Malgré tous nos efforts, nous n'aurons jamais un taux d'échec nul. D'une part, le cache est initialement vide, donc tous les premiers appels débouchent sur un échec. De plus, il y aura plus de requêtes que l'associativité du cache ne le permet. Ce facteur dépend de l'application mais aussi des choix faits dans l'implémentation. Finalement, si la taille du cache est trop petite, les blocs de données seront systématiquement écrasés puis rechargés (ce facteur est le plus important des trois).

19 Virtual memory

Bien que nous ayons l'embarras du choix quant à la gestion de nos adresses, nous sommes inexorablement ramenés à la nécessité de stocker ces informations quelque part. De plus, une question reste en suspens : Quelle partie de la mémoire l'OS allouera-t-il à un processus donné? L'entièreté? Une fraction de la mémoire physique? La mémoire virtuelle est une façon élégante de permettre à chaque processus d'avoir un champ d'adresse aussi vaste que celui de la mémoire physique.

La mémoire virtuelle consiste à laisser penser au processus qu'il a accès à l'entièreté de la mémoire physique alors qu'en réalité, une partie seulement se trouve en RAM tandis que le reste a été transféré sur le HDD. Une Look Up Table permet d'établir la correspondance entre l'adresse virtuelle et son équivalent physique. Ce procédé amène également le principe de pagination qui consiste à définir des blocs d'une certaine dimension (typiquement 4KB). Les accès disques étant beaucoup plus lent, on essaye de minimiser leurs occurrences en ne rapatriant vers la RAM non pas des données seules mais des collections entières.

20 Data parallelism

Les ALU visent une efficacité maximale pour les types courants de données. Pourtant, ceux-ci sont de tailles variables, or la complexité de l'ALU dépend du nombre de bit des opérandes concernés. Un compromis possible entre la performance de l'ALU et son coût est une architecture à 16 bits. Des opérations nécessitant de plus grands opérandes seront subdivisées en opérations plus petites. Cependant, les ALU destinées à un usage généraliste chercheront à satisfaire un large éventail de types de données. Au fil des années, elles ont pu évoluer de 4 vers 64 bits grâce aux améliorations apportées aux circuits intégrés. Toutefois, bien que la possibilité de pouvoir traiter des données de 64 bits puisse avoir l'air attrayante, l'utilisation d'une ALU surdimensionnée débouche sur un gaspillage de ressources. En guise d'exemple, le traitement de `char` n'exploite une ALU 64 bits qu'à 12.5%. Une solution est de créer des agrégats de plusieurs données dans un type de taille maximum.

Cette observation concerne également les bus de données. Nous chargerons autant de données qu'il est possible d'en véhiculer sur un bus afin de maximiser son utilisation. L'emploi de paquet de données, appelé mode vectoriel, est aussi référencé comme *sub-word parallelism*.

```
unsigned char A, B, C;
int i;
for (i = 0 ; i < 1000000; i+=4) {
C[i+0] = A[i+0] + B[i+0];
C[i+1] = A[i+1] + B[i+1];
C[i+2] = A[i+2] + B[i+2];
C[i+3] = A[i+3] + B[i+3];
}
```

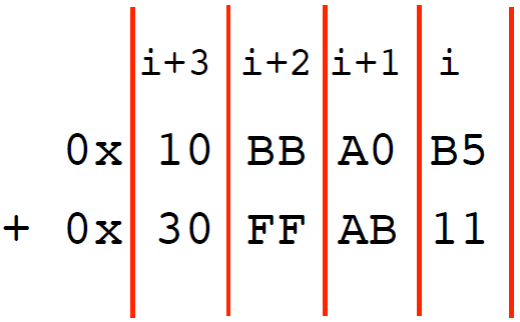


Figure 30: Data packing

Dans l'exemple ci-dessous (figure 31), on peut voir que le cas de droite n'est pas aligné. En effet, dans un contexte de *data packing*, nous aurions un problème d'alignement:

```
unsigned char *a, *b, *c;
int i;
for (i = 0; i< N; i++)
   a[i] = b[i] + c[i];
```

```
unsigned char *a, *b, *c;
int i;
for (i = 0; i< N; i++)
   a[i] = b[i+1] + c[i+2];
```

Figure 31: Data alignment example

	i	i+1	i+2	i+3	i+4	i+5	i+6	i+7
b	b[i]	b[i+1]	b[i+2]	b[i+3]	b[i+4]	b[i+5]	b[i+6]	b[i+7]
+								
c	c[i]	c[i+1]	c[i+2]	c[i+3]	c[i+4]	c[i+5]	c[i+6]	c[i+7]

Comme au chargement dans l'ALU, les vecteurs b et c ne sont pas alignés pour le calcul voulu (i+1 et i+2), on ne peut pas directement faire la somme par bloc, il faut alors corriger cela, ce qui est coûteux en performance.

Cache line split Si les données ne sont pas alignées en mémoire, il se peut qu'une donnée soit découpées sur plusieurs entrées du cache, et donc son accès demandera plus d'une opération de rapatriment auprès du cache, on parle alors de *cache line split*.

21 Classification

A dessein de garder la complexité de l'ALU faible, nous limitons généralement les opérandes au nombre de deux. Les calculs impliquant plus d'opérandes que cela sont subdivisés en sous calculs plus simples et effectués de manière séquentielle.

21.1 Flynn's classification

La classification de Flynn exprime le niveau de parallélisme présent dans une architecture.

Tron Impliqué dans un sérieux imbroglio avec `MCP`. *Come on, Kevin!*

Single Instruction Single Data (SISD) À chaque cycle d'horloge, une seule instruction est exécutée avec une seule paire d'opérandes.

Single Instruction Multiple Data (SIMD) À chaque cycle d'horloge, une seule instruction est exécutée sur une collection d'opérandes. Les super calculateurs des années 60 à 80 étaient basés sur ces architectures. Ils se sont progressivement dirigé vers des processeurs plus généraux. L'avènement du multimédia, dans les années 90, a ramené la technologie SIMD dans nos ordinateurs qui constitue à présent le GPU. En effet, le traitement de signaux (1,2 et 3D) est un candidat idéal du SIMD.

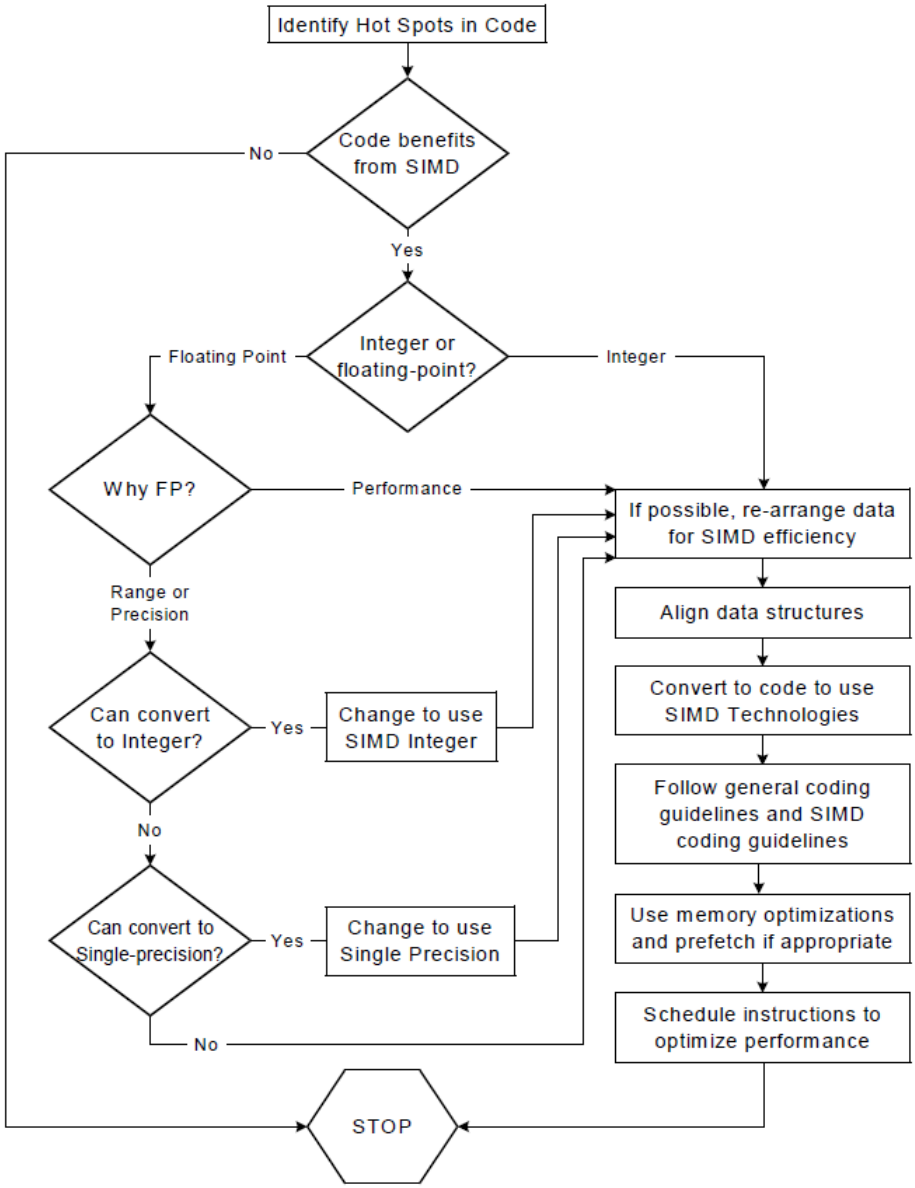


Figure 32: SIMD chart

Cette architecture souffre, néanmoins, de quelques désavantages.

- Tous les algorithmes n'en tirent pas profit.
- Les registres sont coûteux géométriquement.
- Il existe déjà une toute petite surcouche de l'ALU capable de travailler comme un SIMD.
- Les références discontinues aux données nuisent gravement à l'efficacité.
- Son emploi ne peut être automatisé, une vectorisation est nécessaire et ne peut être faite que par des compilateurs spécifiques.

Toutefois, lorsqu'il est possible d'y avoir recours, les gains en temps de calculs sont gargantuesques. Des bibliothèques ont été développées pour faciliter au maximum son implémentation.

Multiple Instruction Single Data (MISD) A chaque cycle d'horloge, une collection d'instructions est exécutée sur une paire d'opérandes.

Multiple Instruction Multiple Data (MIMD) Une collection d'instructions sont exécutées sur une collection de données. Ce qui peut arriver dans des environnements multi-CPU.

22 SIMD Intel x86

AVX, la troisième génération de SIMD, est rétrocompatible avec ses deux version antérieure SSE et MMX. Au cours du temps, la taille des opérandes est passé de 64 à 256 bits et de nouvelles instructions ont vu le jour.

Matrix Math eXtension La première génération de SIMD utilisait les registres en virgule flottante. Bien que ceux-ci soient en 80 bits, nous n'en utilisions que 64 pour stocker 1 entier de 64 bits, 2 de 32, 4 de 16 ou 8 de 8. Il n'était, de ce fait, pas possible d'exécuter simultanément des opérations en virgule flottante et SIMD. On comprend donc ici que bien que la mémoire ne soit pas sensible au type de données, les instructions le sont.

Streaming SIMD Extension De nouveaux registres en 128 bits(xmm0 à xmm7) sont apparus en 99 sur la troisième génération de Pentium. Il y a eu quatre vagues de SSE jusqu'en 2006, amenant chacune de nouvelles instructions en virgule flottante.

Advanced Vector eXtensions Les registres de l'AVX sont 256 bits et il est à présent possible d'impliquer 3 opérandes. Cependant, la gestion de l'AVX requiert l'aide de l'OS.

23 How to use SIMD in Intel CPU?

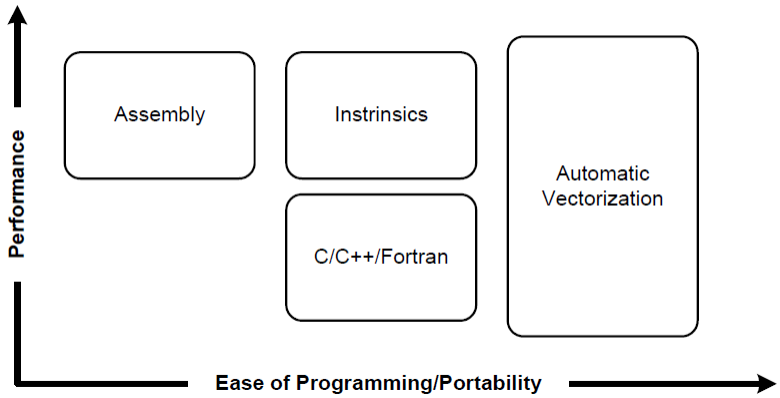


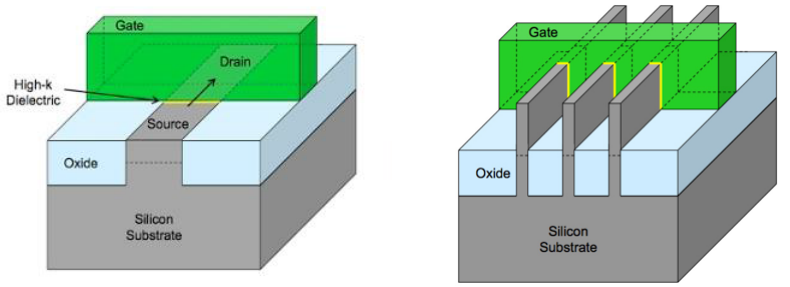
Figure 33: Performance ou aisance de programmation

Les hauts niveaux de langages offrent plus d'aisance au programmateur mais s'éloignent de la réalité matérielle de la machine. Il est parfois bon de sacrifier un peu de cette abstraction au profit de la performance. Il est cependant possible d'imbriquer des bouts de code en langage bas niveau dans des bouts de code plus haut niveau (e.g Assembleur dans du C++). De plus, le langage Intrinsics offre un niveau de langage proche du C tout en offrant des performances similaires à l'assembleur.

24 Sandy and Ivy

L'architecture Sandy and Ivy est apparue en deux temps. En 2011 d'abord, Sandy a été proposée pour les core i3, i5 et i7. Il s'agissait alors de technologie 32nm. Un an plus tard, l'utilisation de lasers 22nm pour graver les circuits imprimés a apporté de nouvelles caractéristiques telles que les transistors multi-gate et un gain principal de performance par Watt de 50%.

Comme nous l'avons mentionné précédemment, la disposition géométrique des composantes fait partie de l'optimisation de l'architecture. Il est, cependant, difficile de maximiser toutes les aires simultanément. Les transistors multi-gate, proposent de quitter le domaine 2D. Ainsi, les 3-D transistors, aussi appelés tri-gate, rajoutent une surcouche au silicium, diminuant ainsi la surface utilisée au détriment du relief.



(a) Mono gate (b) Tri gate

Figure 34: Planar vs. Multi-gate

Les système à puces (System On Chip - SoC) comportent 7 zones.

1. Chaque coeur physique correspond à deux coeur logiques. On peut donc attribuer 2 thread à chaque coeur physique.
2. De la mémoire RAM DDR3.
3. Des caches L3 (Least Level Cache - L3).
4. Des ports permettant d'interconnecter d'autres SoC.
5. Un GPU embarqué.
6. Un contrôleur.
7. Des interfaces IO.

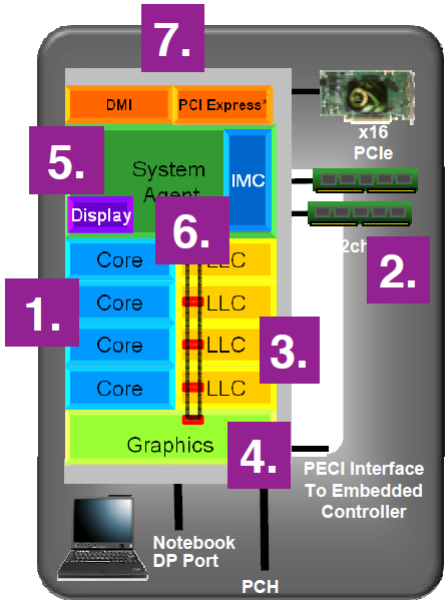


Figure 35: SoC

Turbo Boost La puissance dissipée étant fonction de la tension d'alimentation des transistors et de la fréquence ($P = C \cdot V_{dd}^2 \cdot F$), nous ne pouvons pas nous permettre de tourner continuellement à très haute fréquence. Le Turbo Boost permet aux coeurs de tourner au delà de leur fréquence nominale pendant un court instant et de dissiper la chaleur au travers de tout le circuit imprimé.

DDR Les DDR3 sont, initialement, peu gourmandes en énergie mais il reste possible de les overclocker. Ceci permet de monter jusqu'à 2800 MT/s au lieu de 2133. De plus, une interface IO embarquée (*power gating*), permet de diminuer la consommation d'énergie lorsque les RAM ne sont pas sollicitées.

LLC Le cache de plus bas étage (LLC) est partagé entre tous les coeurs, le GPU et le CPU. Chaque agent pouvant accéder aux données du cache sans permission requise, un contrôleur devra s'assurer du respect de l'intégrité des données. On parle, alors, de domaine de cohérence :

- **IA** Lorsqu'un mot est modifié en mémoire, tous les agents sont notifiés par un message broadcasté. On parle de *snooping*.
- **Graphic** Le CPU met des données dans le LLC et invoque le GPU. Lorsque celui-ci a exécuté les calculs, il notifie le CPU que les données sont prêtes.
- **Non-Coherent** Le CPU met les données dans le LLC et invoque le GPU qui renvoie le résultat vers l'affichage.

On-die Interconnect Le médium utilisé pour la communication n'est, cette fois, pas un bus. Il s'agit de quatre anneaux totalement parallèles: Données, Requête, Accusé et Snoop. Ces anneaux sont adaptés aux coeurs en fréquence, voltage, bande passante, latence et puissance. Ces anneaux sont disposés, géométriquement, au dessus des LLC et gérés par un protocole sophistiqué de gestion du médium.

System agent C'est un arbitre qui gère tous les accès du *ring domain* et des I/O (PCIe et DMI) et achemine les accès aux bons endroits. Les contrôleurs PCIe permettent la connexion à des dispositifs PCIe externes et ont différentes possibilités de configuration. Le contrôleur DMI, quant à lui, est connecté au chipset PCH (*Platform Controller Hub*) qui contient un moteur d'affichage intégré, un *Flexible Display Interconnect* et un *Display port*, servant pour les opérations graphiques internes. Au final, le *System agent* gère aussi le contrôleur mémoire.

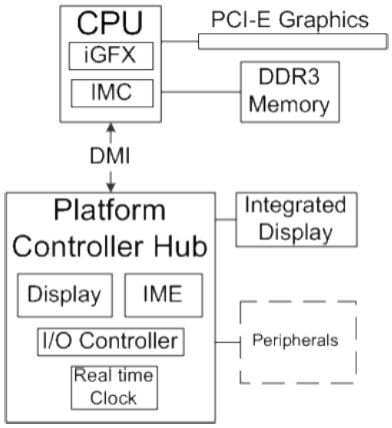


Figure 36: System agent

Graphics Processor C'est un GPU intégré au processeur et ses fonctionnalités sont orientées vers le multimédia, le gaming ainsi que l'encodage/décodage vidéo.

Dies Pour la plupart des configurations denses, on a jusqu'à 2 milliards de transistors sur un seul *die* et $435mm^2$ de surface. Il est à noter que la taille du cache L3 a un impact déterminant sur la surface utilisée.

24.1 Core architecture

La partie *in-order front-end* rapatrie les instructions et les décode en *micro-ops*. Celle-ci fournit au pipeline suivant un flux constant de *micro-ops* dans l'ordre le plus probable d'exécution.

Un moteur d'exécution *out-of-order* superscalaire répartit jusqu'à 6 *micro-ops* à l'exécution par cycle. Le bloc se chargeant d'allouer/renommer réarrange les *micro-ops* en suivant l'ordre du flux de données de manière à ce qu'elles puissent s'exécuter aussi tôt que leurs sources sont prêtes et que les ressources d'exécution sont disponibles.

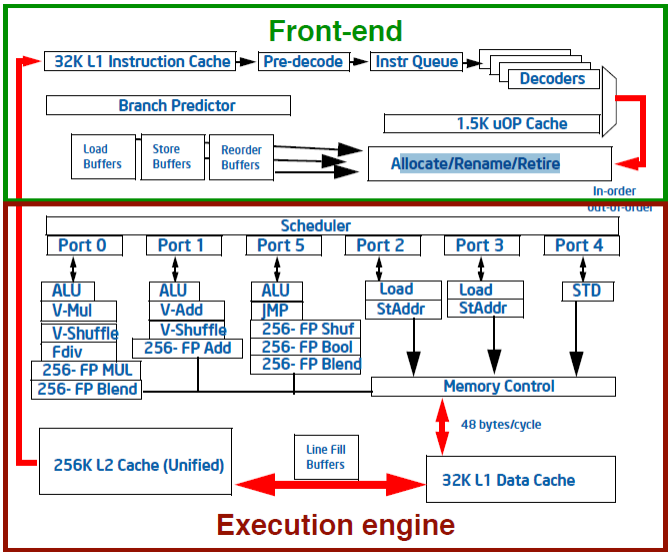


Figure 37: Sandy Bridge pipeline

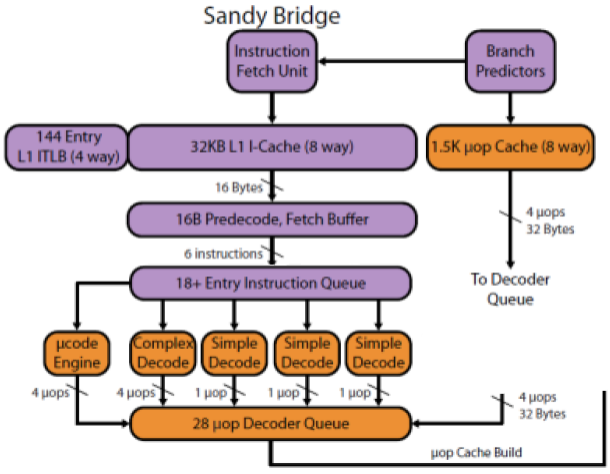


Figure 38: Front end close up

Front-end Le Branch Prediction Unit (BPU) indique le prochain bloc de code à exécuter à l'unité de Fetching qui transmet alors les instructions au cache L1. Une première étape de pré décodage est ensuite effectuée et le résultat est bufferisé pour être vidé progressivement par un système de file d'attente. À la sortie de celle-ci, nous avons 3 unités de décodage rudimentaires ainsi qu'une autre plus évoluée.

OoO Engine Une détection automatisée de chaînes de dépendances permet une exécution out-of-order tout en maintenant le flux de donnée. Lorsqu'une instruction dépendante est en attente de ressources, une autre chaîne d'instruction est sortie du cache de second niveau. Le design architectural est soucieux de la consommation d'énergie. L'engine se compose de trois entités importantes :

- **Renamer** Rapatrie les micro-opérations depuis le front-end vers le coeur d'exécution et détermine les dépendances afin d'activer l'OoO.
- **Schedule** Gère les files d'attentes jusqu'à ce que tous les opérandes d'une opération soient prêts et distribue les micro-opérations disponibles vers les unité d'exécutions concernées selon un ordre FIFO (autant que faire se peut).
- **Retirement** Retire les instructions ainsi que les micro-opérations et gère les exceptions.

Execution Core Le coeur d'exécution est une unité suprascalaire OoO. Il se constitue de trois stacks correspondant chacune à un type de données. Une GP (Général Purpose) travaillant sur des entiers, une SIMD (FP ou Int) et une pour le calcul en virgule flottante. Il est directement connecté aux caches mais c'est le scheduler qui gère l'ordonnancement. Celui-ci peut répartir jusqu'a 6 micro-opérations par cycle.

Cache Hierarchy Le premier niveau contient deux caches, un pour les instructions (L1), l'autre pour les données (L1D). Chaque coeur physique possède ses propres caches de premier niveau, mais dans le cas d'Hyper Threading les coeurs virtuels se les partagent. Ils possèdent tous également un cache L2 commun pour les instructions et les données. Celui-ci à recourt aux Translation Lookaside Buffer (TLB -Instruction, Data ou Shared) pour la traduction des adresses logiques en adresses physiques. Le cache L3 (ou LLC) est un cache commun à tous les coeurs tel que nous l'avons vu précédemment. La cohérence de tous les niveaux de caches est maintenue par le protocol MESI.

Level	Latency (cycles)	Bandwidth (per core per cycle)
L1D	4	$2 \cdot 16$ bytes
L2	12	$1 \cdot 32$ bytes
LLC	26-31	$1 \cdot 32$ bytes

Figure 39: Cache Hierarchy Overview

Lorsqu'une instruction lit des données d'une mémoire pouvant être modifiée, le processeur vérifie l'état des caches et de la mémoire. Lorsqu'une donnée est labellisée comme étant *dirty*, il est nécessaire de mettre à jour le cache. Si une entrée n'est pas dans le cache (miss), le processeur remonte la hiérarchie jusqu'à, dans le pire des cas, la mémoire. Dans certains cas, le processeur à accès au caches L1 et L2 des autres coeurs et peut vérifier si la donnée qu'il recherche est clean dans l'un d'eux.

24.2 Power and thermal management

La puissance dissipée est un sujet de premier plan. Il est donc crucial de limiter les fréquences à tous les étages afin de diminuer la consommation et les dégagements thermiques. Il existe une composante statique, exprimant les pertes et les imperfections des transistors réels, ainsi qu'une composante dynamique liée à la commutation des portes logiques. Jusqu'à récemment, la composante dynamique était notre préoccupation principale mais les pertes statiques entrent, peu à peu, en compte. Leur diminution peut être obtenue par une réduction de la période d'horloge et de la tension d'alimentation des transistors (V_{DD}). La plupart des IC actuels le font.

La gestion de la puissance, qui était jusque là gérée par le BIOS, est devenu la tâche de l'OS. L'*Advanced Configuration and Power Interface* (ACPI) est un standard ouvert permettant le Power Management et Monitoring qui tourne sur un coeur dédié. Le CPU peut être placé dans plusieurs états de torpeur lui permettant de jouer sur sa performance et sa consommation.

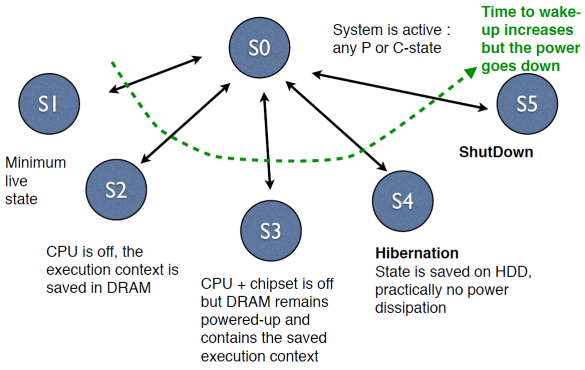


Figure 40: Sleep States

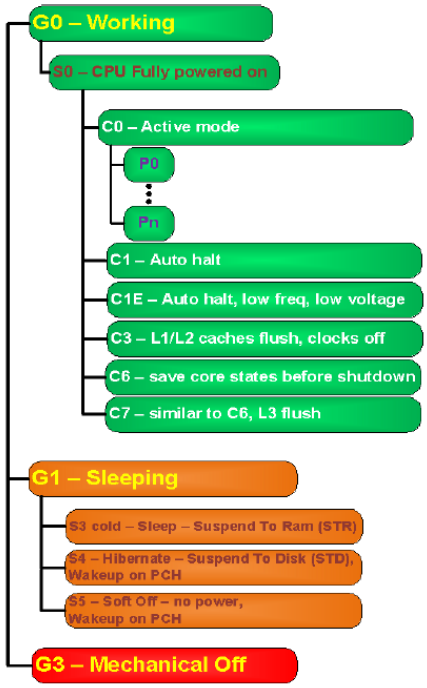


Figure 41: C-States

Lorsque le CPU est dans l'état S_0, il est alimenté à puissance maximale. Il existe cependant 7 états diminuant de plus en plus sa consommation. En C_0, les coeurs sont dans un couple (V_{DD}, C_{lk}) donné(Dynamic Voltage and Frequency Scaling - DVFS), on parle de P-State. En C_1, les signaux d'horloge sont désactivés, le coeur n'est pas fonctionnel. En C_3, les caches sont vidés mais le coeur reste actif sous une tension V_{DD}. En C_6, l'alimentation est coupée pour éviter les fuites et l'état du système est stocké dans le LLC pour une consommation proche de 0W.

P-States Le Thermal Design Power (TDP) est la puissance maximale que le système de refroidissement est capable d'assumer. Cependant, il est possible, sous Turbo Boost, de dépasser cette limite pour de courtes durées. P_0 est la fréquence maximale possible, P_1 est la fréquence nominale, et P_n est la fréquence de consommation minimale. Ces gammes de fréquences sont très larges (en GHz). Les niveau 0 et 1 ne peuvent être activés que par le matériel tandis que les niveaux 1 à n sont géré par l'OS et dépend des préférences de l'utilisateur.

P-State	Frequency	Voltage	Power
P0	1.6 GHz	1.484 V	25 Watts
P1	1.4 GHz	1.420 V	~17 Watts
P2	1.2 GHz	1.276 V	~13 Watts
P3	1.0 GHz	1.164 V	~10 Watts
P4	800 MHz	1.036 V	~8 Watts
P5	600 MHz	0.956 V	6 Watts

Figure 42: P-States examples

L'utilisation de Turbo Boost est maximale sur un seul coeur, il est possible d'accélérer tous les coeurs simultanément mais le gain n'est pas celui escompté.

N° de cores	N° of Turbo Steps	Max F [GHz]
3 or 4	7	3.2 = 2.5+7*100
2	9	3.4 = 2.5+9*100
1	10	3.5 = 2.5+10*100

Figure 43: Turbo Boost

En règle général, les périodes de Turbo Boost sont contrebalancées par les périodes d'inactivité de sorte qu'au fil du temps, la puissance dissipée est relativement stable autour de la valeur du TDP.

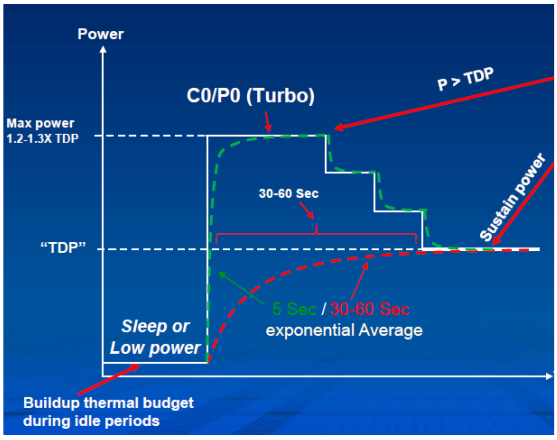


Figure 44: Consommation au cours du temps

Il y a 12 capteurs par groupe (CPU, Graphics, Ring, System Agent), qui s'assurent que les composantes ne sortent pas de la plage de températures $50 - 100$C. Ces températures sont relevées pour tout le chip ainsi que par unité fonctionnelle. On se protège ainsi d'échauffements critiques en régissant la consommation de puissance et le système de refroidissement.