# Synthèse
# Architecture des ordinateurs Q2

Premier Bloc
Sécurité des systèmes
Année académique 2019-2020

*Rédigé par Sénéchal Julien*

28 Mai 2020

# 1 La virtualisation

## 1.1 Avantages

- Meilleure utilisation du CPU
- Economie en surface et en électricitée
- Load Balancing (changer la charge d'un serveur si celui-ci est trop surchargé)
- Tolérance aux pannes
- Isolation
- Peut servir a déboguer des programmes (ne pas planter l'OS principal)
- Développement logiciel
- Compatibilitée avec un ancien systèmes
- Virtualisation hétérogène
- Affectation des ressources a chaud

## 1.2 Contraintes

- Pas de self-service (Gestion par Admin)
- Surchage du travail du CPU
  - Solution : containers

## 1.3 Methode de virtualisation

- Hyperviseur (Hyperviseur de Type I)
  - Intermédiaire direct entre le matériel et les OS Virtualisés
  - Tourne en espace kernel (rôle de l'OS)
- Modèle hôte invité (Hyperviseur de Type II)
  - VMM (Manager de machine virtuelle) exécuté en espace utilisateur (application) $\Rightarrow$ VM dans l'espace utilisateur.
    Résultat : + lent et + facile a mettre en place
- Dans les 2 cas, le VMM doit donner l'illusion a l'OS invité qu'il a l'accès aux ressources de matériel

## 1.4 Technologies de virtualisation

- Matériel
  - Le matériel fournit un support permettant de faciliter la consctruction d'un VMM et de pouvoir isoler les OS invités.
    Exemple :
    - Pour processeur AMD : AMD-V
    - Pour un processeur Intel : VT-x
    - Etc...
- Logiciel
  - Full virtualisation : Création de faux matériel par l'hyperviseur, auxquels l'OS invité a accès. (L'OS invité doit être de la même architecture que le processeur physique : x86, x64, etc...)
  - Emulation : Création d'un environnement virtuel complet (même le CPU avec une architecture différente) $\rightarrow$ Basses performance
  - Paravirtualisation : L'OS invité sait qu'il est dans une machine virtuelle (peut communiquer directement avec l'hyperviseur)
  - Virtualisation assistée par le matériel (HVM) :
    - Ajout d'extension de virtualisation au processeur
    - Plus d'emulation de mémoire
    - Gestion de leur propre interruptions/contexte par les VM elles-mêmes
    - Accès direct au processeur
  - Isolation : Spécificitée UNIX ! Permet d'isoler une application du reste du monde.
  - Container :
    - Pas d'emulation matériel

- Un répertoire = un OS
- Pas de noyau
- Exemple : Dockers

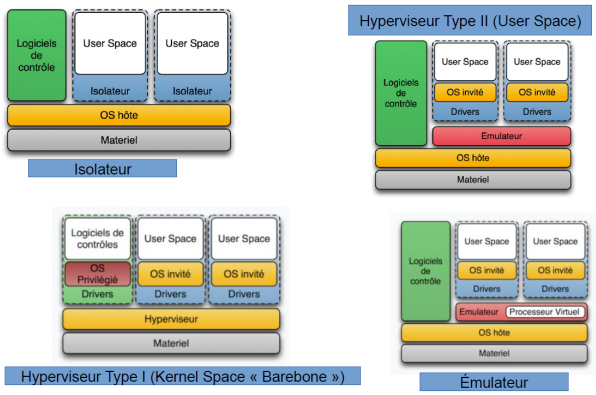


FIGURE 1 – Résumé des différents types de virtualisation

## 1.5 Privilèges d'accès

- L'OS hôte accède directement au matériel
- Niveau de privilège :
  - Mécanisme interne au CPU qui empêche une application d'usurper d'identitée d'un OS hôte
  - *Rings Levels*
    - Exemple : 0 = le plus privilégié = OS hôte
    - Le nombre de niveau dépends de l'architecture du CPU
    - *Ring 3* - User level : Le code est exécuté directement par le CPU mais il ne peut pas accéder directement a la mémoire ou au matériel
    - *Ring 0* - Kernel Level : Accès complet
    - L'OS invité ne vera jamais qu'il n'est pas en *Ring 0*. Pour cela, lorsqu'un programme veut exécuter une instruction pour laquelle il n'a pas de privilège suffisant, une exception va être créé (*trap*), et l'hyperviseur qui écoute chaque *trap* des machines va intercepter les exceptions et reprendre la main en émulant le comportement attendu par la machine invitée.
    - Shadow Structure : mécanisme pour cacher et gérer l'accès aux informations privilégié (copie privée propre a la VM)

## 1.6 Virtualisation de la mémoire

- VMM
  - Partage mémoire physique
  - Allocation dynamique
- Présentation de la mémoire a un OS :
  - La VM voit un espace mémoire non réel
  - Gestion d'une table de correspondance sur la RAM entre les pages [1] mémoire de la machine virtuelle et les pages physique des ressources matériel.
- Virtualisation de la MMU (Memory Management Unit)
  - Shadow page table
    - 2 associations dans cette table : LA VM → PA VM (créé par l'OS invité) & PA VM → Adresse machine - MA (créé par le VMM)
    - Une seule table d'association entre la mémoire virtuelle et la MMU de la machine ce qui est presque aussi rapide qu'en natif.
  - Hardware
    - Extensions x86 pour la virutalisation

1. *Voir "la pagination" en cours de Système d'exploitation*

  - Gain de performance : Pas de perte ou perte légère ($\pm$ 40%)

## 1.7 Cloud

- SAAS
  - Software as a service
  - Ex : Google Doc, etc...
- PAAS
  - Platform as a service
  - Hebergement application
  - Ex : Google Engine, Azure, etc...
- IAAS
  - Infrastructure as a service
  - Ex : OpenStack, etc...
- Publique
  - Via a fournisseur Cloud
  - Ex : Amazon, RackSpace, etc...
- Privé
  - Cloud interne
- Hybride *(Mix de Privé et Publique)*

## 1.8 Container

- Un container $\Rightarrow$ Un OS
- Virtualisation au niveau de l'OS :
  Plusieurs instances du même OS qui tournent avec le même Kernel
- Chaque container contient tout ce dont il a besoin
- Espace fournit par l'OS hôte
- L'OS hôte isole bien chaque container et alloue la RAM/CPU nécessaire
- **Avantages :**
  - Facilité de déployer plusieurs instances d'une même application
  - Isolation
  - Performance propotionnelle a une VM
- **Inconvénients :**
  - Dépends de l'os
  - Isolation moins importante qu'avec une VM

# 2 Les ordinateurs superscalaires

Ordinateurs superscalaires ⇒ Ordinateurs utilisant une puce avec plusieurs *ALU* travaillant en parrallèle.
**Rappel :** L'alu est l'"Arithmetic logic unit". Cette unitée gère les calculs des instructions.

## 2.1 AR2

- 2 *ALU*
- Execution de 2 instructions *arithmétique* en parrallèle
- Même registre pour les 2 *ALU*

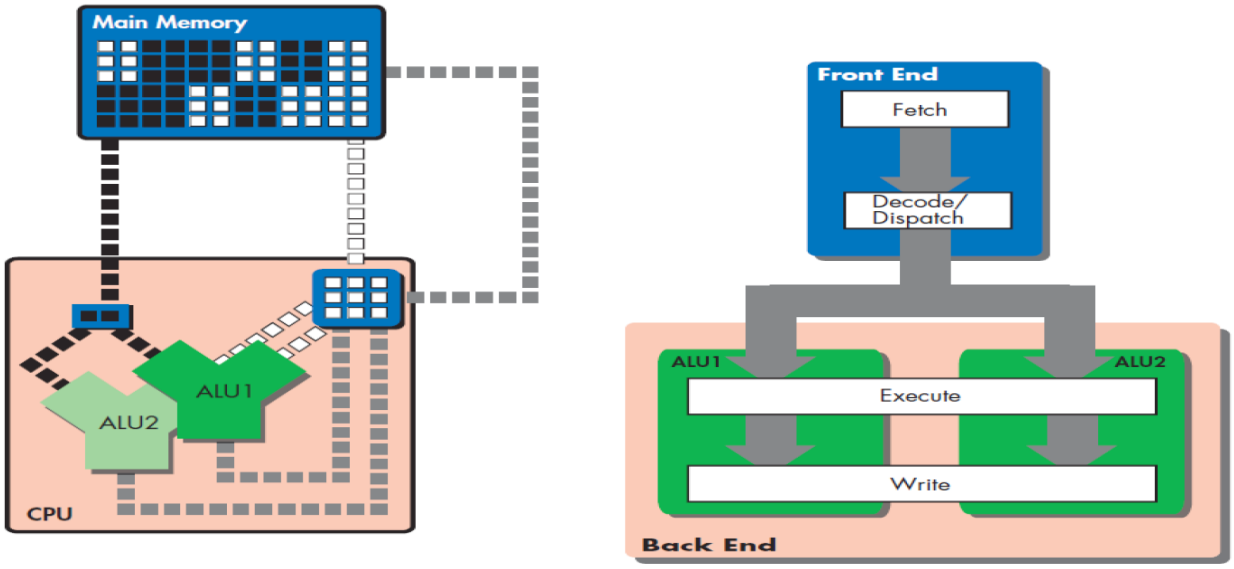


FIGURE 2 – Représentation AR2

### 2.1.1 Decode/Dispatch

Un nouveau circuit vient de s'ajouter a ce que l'on connaissait déjà (voir FIGURE2).
Rôle :

- Déterminer si 2 instructions peuvent être réalisés simultanément
- Si oui, l'unitée de dispatch va répartir les 2 instructions sur les 2 ALU
- Si non, l'unité de dispatch les envoie dans l'ordre du programme vers le premier ALU

### 2.1.2 Programming Model

- Pas de changements
- Illusion d'une execution séquentielle pour le bien du programmeur.

### 2.1.3 Cycle d'exécution

- Capacité de *FETCH* 2 fois simultanément
- Capacité de *Decode/Dispatch* 2 instructions simultanément

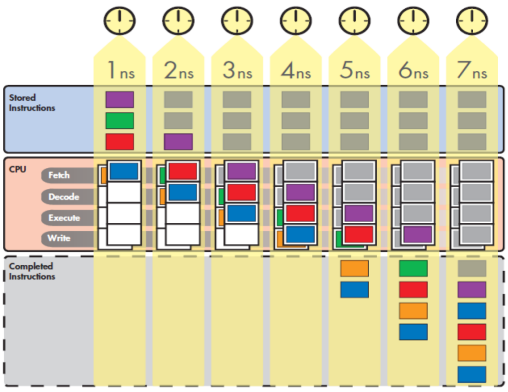


FIGURE 3 – Représentation du cycle de l'AR2

## 2.2 Architecture superscalaire

- Permet de dépasser le seuil d'une instruction par pulsation d'horloge fixée par un pipeline simple. C'est le cas du AR2 que l'on vient de traiter.
- + un CPU disposera d'ALU, + $\frac{instructions\ terminees}{pulsation\ d'horloge}$

### 2.2.1 Différentes opérations arithmétique et logique

- Opérations arithmétique : $+$, $-$, $\div$, $\times$
- Opérations logiques : AND, OR, NOT, XOR, shits, rotation de bits
- Opérations Scalaires et Vectorielles

Tout cela pour en venir au fait qu'on pourrait répartir spécifiquement les tâches ! ! !

### 2.2.2 Processeur Intel Pentium

Répartition des tâches de l'ALU :

- IU : Unité d'execution des entiers (Interger execution unit)
- FPU : Unité d'execution des virgules flottantes (Floating-point unit)
- LSU : Unité responsable des *laod,store* et du calcul des adresses (Load-Store unit)
- BEU : Unité responsable sauts conditionnels et non conditionnels (Branch Execution Unit)

Jusqu'ici les ALU étaient une unité concrète, dorénavant, l'ALU sera un terme générique de l'ensemble des unités d'exécution faisant des opérations arithmétiques ou logiques.

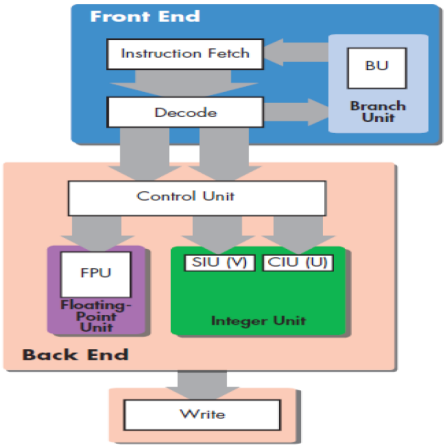


FIGURE 4 – Représentation de L'Intel Pentium

### 2.2.3 Problèmes de l'architecture superscalaire

- Certaines conditions peuvent empêcher la mise en parrallèle de plusieurs instructions arithmétiques
- Possibilitée de création de *"bulles"* (comme vu au Q1)
- Exemple :
  - 1ère instruction : *add A,B,C*
  - 2ème instruction : *add C,D,D*
  - **Impossible de les réaliser simultanément**
  - Conséquence en superscalaire : L'ALU éxecutant la 2ème instruction devra attendre que l'ALU executant la 1ère instruction finisse.
  - Conséquence en pipeline : La 2ème instruction *add* devra attendre attendre que la 1ère passe son étape *write* avant de pouvoir faire son *execute*
- Solution :
  - L'unité de Dispatch doit remarquer si 2 instructions ne peuvent pas être exécuté en même temps
  - Une technique appellé le *forwarding* consiste a prendre le résultat de la 1ère instruction a la sortie de l'ALU et de le renvoyer directement dans le 2ème ALU. On évite ainsi l'étape du *Write*. Pour éviter les conflits, une technique est de réattribuer les registres sur des registres physiques comme on peut voir sur la FIGURE5

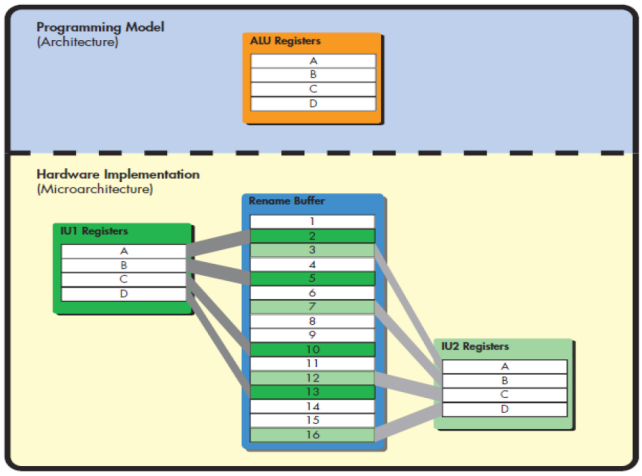


FIGURE 5 – Réattribution des registres

Exemple concret de son utilisation :
- 1ère instruction : *add A,B,C*
- 2ème instruction : *add D,B,A*
- Pas de dépendances dans ces 2 instructions et donc les 2 instructions peuvent s'exécuter simultanément.
- Problème sans réattribution des registres :
  La 2ème instruction réécrit sur le registre A qui lui est utilisé dans le calcul de la 1ère instruction.
- Avec réattribution des registres :
  Pas de soucis, la 2ème instruction fait son calcul, mets le résultat dans son registre personnel temporaire. Pareil pour la 1ère instruction. Et c'est seulement quand les 2 ont fini leur exécution que la valeur est écrite dans les registres du programming model.

### 2.2.4 Problèmes liés a la structure du CPU

- Il est nécessaire de faire des modifications pour que les 2 ALU puissent accéder simultanément au registre
- Solution :
  - Regrouper tous les registres en une unité spéciale → le fichier de registre. Il est accèssible par un bus de données et par 2 ports (Input/Output)
  - Chaque ALU est connecté à cette unité à l'aide de 2 ports de lecture permettant la lecture simple de 2 registres simultanément et d'un port d'écriture.

### 2.2.5 Problèmes liés aux instructions de saut

Avant, il fallait attendre pendant que l'instruction de saut était évaluée et donc ne pas continuer le pipeline... Ce qui créait des bulles

Maintenant, la solution est une technique appelé la *prédiction de branchement* qui contourne ce bloquage. Une fois que l'adresse de la prochaine instruction connue, l'attente du chargement de la prochaine instruction peut prendre du temps ! A nouveau une technique appelé *instruction caching* permet de réduire ce temps d'attente.

# 3 RISC-CISC

Pour des raisons évidentes, il fallu passer d'un mode où chaque programme était intimement lié au hardware a un mode d'utilisation où l'ISA (Instruction Set Architecture) joue le rôle d'intermédiaire entre le Hardware et le Software.

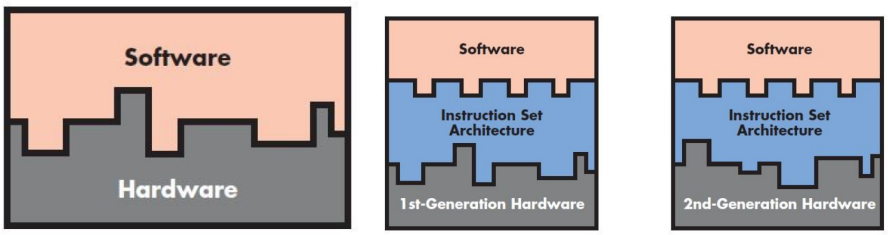


Figure 6 – Création de l'ISA

On a alors mis en place le *microcode engine* pour assurer cette abstraction. Il s'agissait de une mémoire ROM stockant des microcodes programs et une unité d'exécution qui exécute ces programmes.

Fonctionnement :
- L'unité microcode lit l'instruction a exécuter
- Elle trouve alors le code dans la mémoire ROM qui correspond a cette instruction
- L'exécution de cette instruction produit une séquence d'instruction dans la "langue" du processeur

Ce mode de fonctionnement s'appelle l'*emulation* !

Grâce a cela, a l'appartition d'une nouvelle génération de processeur, les fabricants n'avaient qu'a réécrire un microcode pour que l'ISA de l'utilisateur ne change pas.

Au fil du temps, de plus en plus d'instruction on été ajoutée, mais toutes n'étaient pas utilisées et plus d'instriction signifiait plus de ROM pour le microcode et des CPU plus volumineux et plus energivore.

## 3.1 RISC

On se débarrasse de tout ce qui est superflu :
- diminuer le nombre d'instruction
- diminuer la complexité des instructions

But :
- ISA plus léger
- Plus rapide
- Plus facile a implémenter dans le hardware (sans microcode engine)

L'énorme avantage de l'ISA n'as pas été oublié pour autant :
- Pour pouvoir se libérer su microcode engine, les machines RISC ont pu s'appuyer sur les progrès des compilateurs des langages de haut niveau. Migration du hardware vers le software

## 3.2 CISC

CISC est un terme créé à postériori pour distinguer les anciennes méthodes des « nouvelles » pensées RISC.

# 4 Pentium

## 4.1 Mémoire cache

Un transfert de données entre la mémoire centrale (RAM) et les registres se fait sur une quantité énorme de cycle horloge ⇒ apparition de la mémoire cache.

- Petite quantité de mémoire entre les registres et la mémoire centrale (RAM)
- Rapide
- Coûteuse

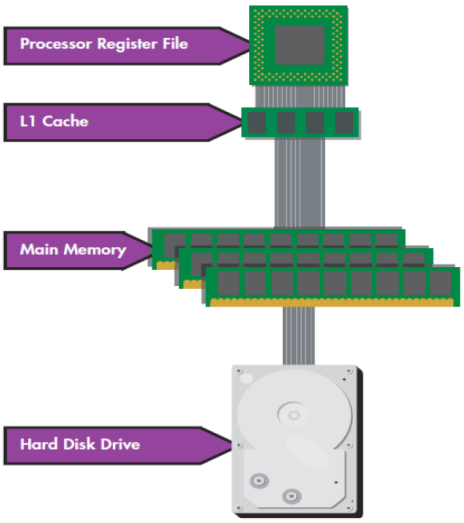


FIGURE 7 – Hiérarchie de la mémoire

- Contient des portions de code et de données fréquemment utilisées
- Proche du processeur
- Il peut exister plusieurs niveaux de cache mémoire :
  - Cache L1 : La plus petite, la plus chère, la plus proche du processeur
  - Cache L2 : Fréquemment utilisé, entre la L1 et la mémoire centrale
  - Cache L3 : Placé entre la L2 et la mémoire centrale. Moins fréquent
- Quand le processeur a besoin de données (ou de code), il vérifie le cache L1 et si ça réussit, il place les données dans le *fetch*. Si ça échoue, il va vérifier dans le cache L2, copier ce dont il a besoin en cache L1 puis être transmis au Fetch, pareil pour le cache L3 en passant par L2, puis L1, etc... Si le contenu désiré ne se trouve dans aucun cache, le processeur va le chercher dans la mémoire centrale et le place dans les cache.

Organisation du cache L1 :

- 2 parties stockées séparément :
  - 50% instruction cache (codes)
  - 50% data cache (données)
- Plus performant

Situation :

- Avant : sur le bus de mémoire reliant le processeur a la mémoire centrale
- Maintenant : les caches L1 et L2 font partie du CPU lui-même

## 4.2 Pipeline du Pentium

Toutes les étapes des pipelines sont toujours les mêmes, sauf pour l'étape *execute* qui est spécifique selon l'instruction.

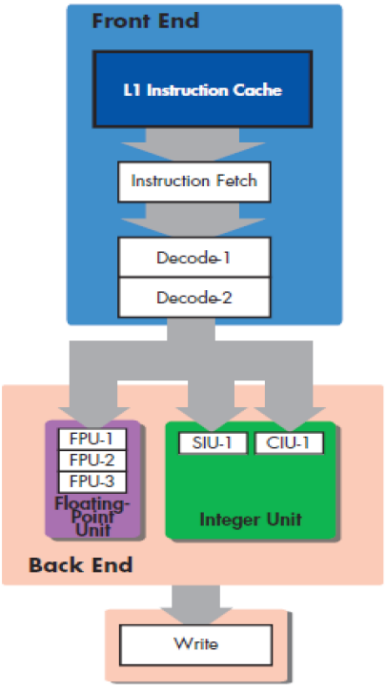


FIGURE 8 – Pipeline du Pentium

Comme la taille des instructions x86 étaient variables, nécessité de les charger dans un tampon où leurs limites sont détectées.

*Decode* est découpé en 2 parties :

- Decode-1 : Conversion de l'instruction en "language" du Pentium et détermination de si il s'agit d'une instruction de "branchement" (jump)
- Decode-2 : Comme l'ISA x86 permet des modes d'adressages multiples, besoin de plus de calcul d'adressage et c'est là qu'il les fait, et il décode aussi les instructions plus longues grâce a une microprogrammation ROM.

## 4.3 Branch Unit / Branch Prediction

Branch Unit (BU) : contient la Branch Execution Unit (BEU) et la Branch Prediction Unit (BPU). Quand une instruction de branchement (jump) conditionnelle est rencontrée, elle est envoyée à la BU pour être exécutée.
Une fois que BU détermine que le branchement (jmup) doit être fait, elle doit déterminer l'adresse de départ (appellé branch target) du prochain bloc d'instructions à exécuter.
Une technique utilisée pour ne plus avoir à attendre que la condition soit évaluée est le *speculative execution* où le processeur va estimer le résultat de la condition et va commencer l'exécution du notre autre bloc de données avant même que la condition soit évaluée, mais le résultat ne pourra être écrit dans le registre tant que la condition n'aura pas été évalué. Si BPU fait bien son travail, il n'y a pas de spéculation et le résultat peut être écrit comme n'importe quel instruction. Si la prédiction est mauvaise, il faut enlever toute l'instruction du pipeline et recommencer a la charger.

Il y a 2 types de prédictions :

- Prédiction statique : Se base sur l'hypothèse que les branchements se déroulent dans un cadre de boucle répétitive et l'instruction de branchement permet de vérifier si la boucle se répète. Très rapide dans un programme rempli de boucle
- Prédiction dynamique : Basée sur des algorithmes qui utilisent 2 tables à savoir une table de l'historique des branchement et une autre table de l'historique des cibles des branchements dans le buffer (registre).

Pentium utilises ces 2 techniques :

- Si une instruction n'as pas d'entrée dans la table historique des branchements, alors il y a prédiction statique.
- A l'inverse, si une instruction a une entrée dans la table historique des branchements, il y a prédiction dynamique

### 4.4 Pentium's Back End

- 2 ALU consacré aux entiers
  - Ne sont pas totalement indépendant l'un de l'autre et donc restrictions limitant les combinaisons d'instructions sur les entiers en parrallèle
  - Pipeline légèrement différent
- 1 ALU pour les opérations à virgule flottante
  - Calculs plus complexe d'où pipeline plus avec plus d'étages que pour les entiers
  - Performances limitées car :
    - Restrictions importantes concernant l'execution simultanées d'une instruction avec des entiers et d'une instruction avec des nombres décimaux
    - L'architecture de l'ISA x87 a montré certaines limites
- Le fichier de registres x87 contient 8 registres de 80 bits organisé en pile (organisé en LIFO - Last In First Out)
  - Pour modifier ou faire sortir un élément en bas de la pile, il faut vider tout ce qu'il y a au dessus
  - Le fichier de registre x87 permet alors de sélectionner un registre bien précis en utilisant ST(i)
    - ST = *Stack Top*, donc depuis le dessus de la pile
    - i = le n° du registre que l'on veut en partant du haut de la pile qui vaut l'étage 0.
  - Problème : pour chacune des instructions arithmétiques à virgule flottante, un des opérants doit absolument être le Stack Top
  - Résultat : le fichier de registre x87 est trop compliqué et est un obstacle aux performances

### 4.5 Pentium et x86

30% des transistors du Pentium étaient dédié à la compatibilitée avec l'ISA x86. Il était donc à la traîne face à ses concurents qui étaient construit autour du RISC. Avec le temps, le nombre de transistors sur un processeurs a augmenté et son prix diminué, ce qui a rendu ce problème moins contraignant au point de surpasser la concurence RISC.

## 5 Pentium Pro

- Performances beaucoup + grande que sur le Pentium Original
- Début de la microarchitecture P6 (grand succès)
  - Très extensible (conservé chez Intel pendant + de 5 ans)
- Démonstration de la domination du marché par Intel

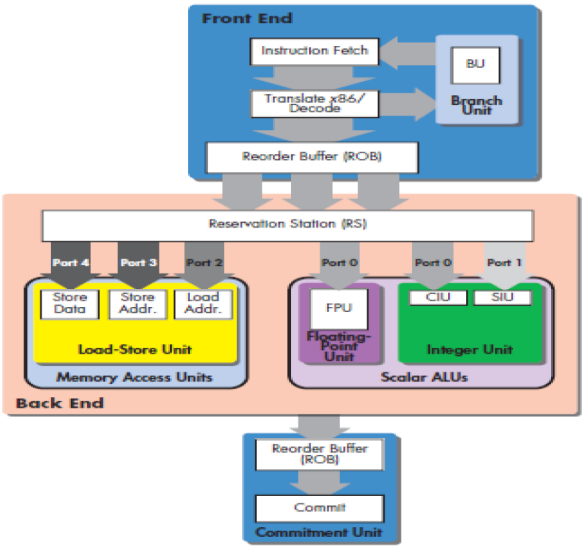


Figure 9 – Représentation Pentium Pro / Architecture P6

Raison de l'abandon du Pentium original :
— S'adapte difficillement aux flux d'instructions dynamiques
— Exploite très faiblement l'hardware superscalaire "large"
— Ne pouvant dispatch que 2 instructions par cycle d'horloge, les règles de dispatch ont été conçues dans cette optique. Or, si plus d'unité, d'execution venaient à être disponnibles, les règles de dispatch devraient prendre en compte cet unité d'execution et donc de prendre en compte toutes les combinaisons possibles.

Solutions aux problèmes du Pentium Original :
— Dispatch des instructions décodées vers un buffer (un tampon) entre l'unité decode et les unités execute
— Quand le buffer possède un certain nombre d'instructions, la planification dynamique du processeur peut-être mis en place et évalué
— Il ne reste plus qu'a envoyer les instructions aux unités execute au moment le plus opportun et dans l'ordre optimal
— Comme avec ce système les instructions peuvent être dans le désordre par rapport au programme original, un tampon est placé a la sortie de son execution pour pouvoir permettre de les remettre dans l'ordre.
— L'opération qui consiste a placer les instruction dans le buffer *issue* et de les envoyer dans les unitées d'execution une fois que la planification dynamique aie été faite s'appelle le *issuing*

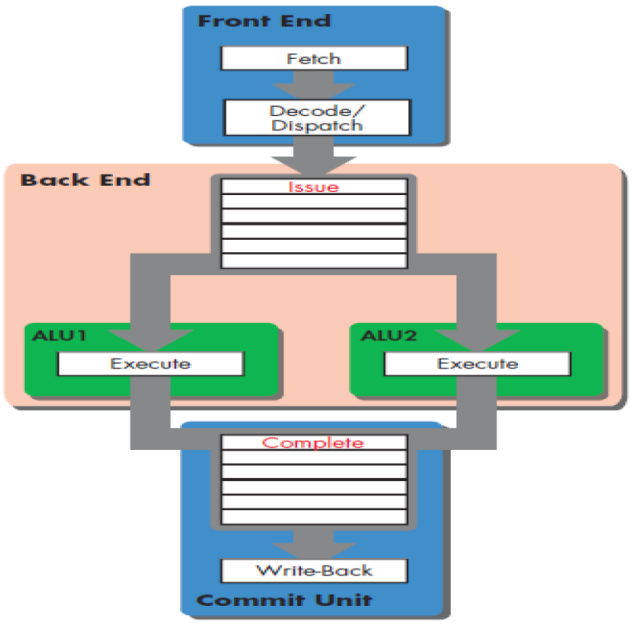


FIGURE 10 – Stratégie de planification dynamique

Pas toujours possible d'executer les instructions dans le désordre :
— Une instruction peut avoir comme entrée le résultat d'une instruction non execute
— Peut attendre des données venant de la mémoire
— Peut attendre que l'unité d'execution soit libre

Toute cette nouvelle stratégie s'appelle *out-of-order execution* ou *execution dynamique* :
— 2 nouvelles phases sont ajoutées
  — Issue phase
  — Completion phase

| | | |
|---|---|---|
| 1<br>2 | Fetch<br>Decode/dispatch | In order |
| 3 | Issue | Reorder |
| 4 | Execute | Out of order |
| 5 | Complete | Reorder |
| 6 | Write-back (commit) | In order |

FIGURE 11 – Cycle de vie d'une instruction avec planification dynamique

## 5.1 Issue Phase

- Mise en tampon
- Réorganisation (on ne suit plus l'odre du programme)
- Peut comporter plusieurs étages de pipeline et/ou plusieurs tampons dépendant des processeurs
- Permet de réduire l'effet des bulles

## 5.2 Completion Phase

- Mise en tampon après l'execution
- Permet de remettre dans l'ordre prévu par le programme
- Permet l'illusion d'une exécution séquentielle
- Quand les résultats d'une instruction sont écrit dans le registre, on dit qu'elle est *commit*
- Une instruction ne peut être commit tant que l'instruction la précédant n'est pas commit

## 5.3 L'architcture P6

### 5.3.1 Issue Phase

Le *reservation station (RS)* est un buffer permettant d'acceuillir 3 instructions simultanément qui attendront que la condition de son execution soit validé, après quoi l'instruction est envoyée a l'unité d'execution (jusque 5 instructions pouvant être envoyé simultanément parce que 5 ports (voir FIGURE9)). Comme dit plus haut, ici on ne suit plus l'odre du programme mais l'ordre d'exécution optimal.

### 5.3.2 Completion phase

Pour pouvoir remettre les instructions dans l'odre, il est nécessaire de savoir dans quel ordre sont arrivées les instructions dans le *reservation station*. C'est pour cela qu'on crée un *recorder buffer (ROB)* juste avant d'entrer dans le *RS* afin de pouvoir connaître le bon ordre de chaque instruction.

### 5.3.3 Pipeline

- Branch Target Buffer (BTB) et Instruction Fetch : 3 niveau et $\frac{1}{2}$
- Decode : 2 niveau et $\frac{1}{2}$ (décoder les instructions x86 vers le format du processeur de type RISC)
- Register name : Registre renommés et instruction enregistrées dans le *ROB* (1 cycle)
- Write to RS : Ecriture dans le reservation station (1 cycle)
- Read from RS : Issue phase en cours (peut rester dans le RS un nombre de cycle indéfini et la transition vers l'unité d'execution prends un cycle)
- Execute : Minimum 1 cycle
- Commit : 2 cycles (écrire dans le ROB puis remettre les résultats dans les registres)

Avantages de ce pipeline :

- ↑ fréquence d'horloge car étapes du pipeline plus courtes

— Pipeline avec un grand nombre d'instruction (problème si il faut vider le pipeline a cause d'une erreur de prédiction)

### 5.3.4 Branch Prediction du P6

— Précision des prédictions de 90%
— 512 entrées du BHT + BTB et 4 bit pour les infos des prédictions passées

### 5.3.5 RISC, CISC

— La phase *decode* est longue a cause de l'Instruction set Traduction
— Pour utiliser les techniques d'ordonnancement dynamique RISC, il fallai limiter la complexitée des instructions en traduisant les opérations x86 en opérations plus courtes et plus à l'image du RISC.
— Mise en place d'une micro architecture de decodage qui utilise le microcode ROM pour les anciennes instructions CISC qui sont traduites en micro-opérations.

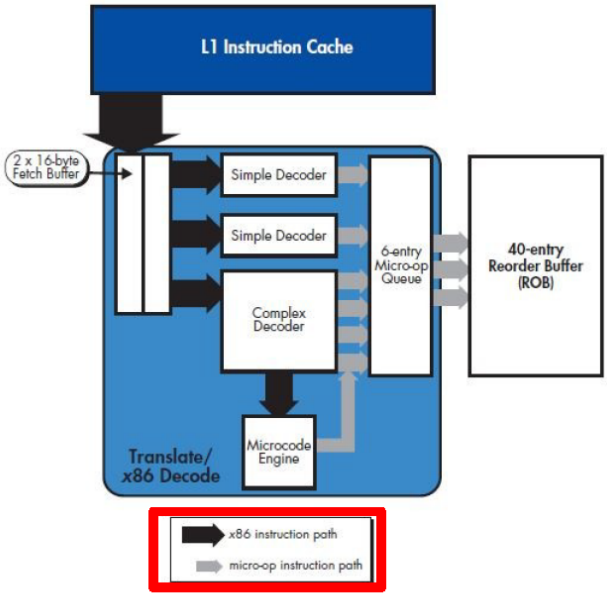


FIGURE 12 – microarchitecture de decodage

Tout ce système d'heritage à un coût :

— 40% de transistors pour ces opérations de decodage

# 6 Processeurs 64 bits

Quand on parles d'un processeur étant de 16, 32, 64 bits ; cela correspond au flux de données du processeur (nombre de bit que peut contenu chaque des registre du processeur).
— La largeur du flux de données a pu doubler
— La taille des bus et la taille des registres ont aussi doublé
Avantages :
— Dynamic Range (DR)
  — On peut représenter beaucoup plus de nombre qu'en 32 bits
  — Certaines applications nécessites des entiers à 64 bits
  — Quand la valeur d'entier dépasse le nombre de bit max du processeur, on entre en situation d'overflow
    — Quand ça arrive, le nombre donné est faux

  - Un bit permet de signaler si la DR a été dépassé pour prévenir les erreurs
- Possibilitée d'utiliser dess adresses plus grandes
  - La fourchette d'adresse s'appelle *l'espace adressable*
  - Tous les composants du PC avec lesquels on communique ont ainsi une adresse
- En 32 bits, le proceseur et l'OS vont faire croire à un programme qu'il a un espace adressable de 4 Go ($2^{32}$). En 64 bits, la théorie serait que cette espace soit de 18 millions de To ($2^{64}$) mais dans les faits c'est pas moins de 282To de mémoire virtuelle ($2^{48}$)

# Table des matières