3. UAL ET REGISTRES

3.2 Registres

Les registres sont une zone de stockage temporaire des données située dans le processeur.

Ils sont construits à l'aide de circuits logiques séquentiels afin de pouvoir mémoriser des bits.

Leur intérêt principal est de pouvoir travailler avec des données localisées directement dans le processeur et donc d'un accès beaucoup plus rapide que celles situées en mémoire principale.

3. UAL ET REGISTRES

3.2 Registres

Registres généraux (1)

Les registres généraux (regroupés dans un banc de registres) sont à la disposition du programmeur en assembleur (ou du compilateur si l'on code en langage évolué), qui les utilise à sa guise dans les instructions pour manipuler des données.

On a évidemment intérêt à travailler au maximum avec les registres généraux et à n'utiliser le stockage en mémoire qu'en dernier ressort car cela ralentit fortement l'exécution des instructions.

Suivant les processeurs, on dispose d'une dizaine à une centaine de registres généraux. Plus ils sont nombreux, moins le programme fait appel à la mémoire, mais plus les circuits prennent de la place sur la puce.

Ces registres sont caractérisés, comme l'UAL, par leur taille : combien de bits peuvent-ils stocker ?

3. UAL ET REGISTRES

3.2 Registres

Registres généraux (2)

Depuis plusieurs années, les processeurs sont capables de travailler avec des nombres de 32 bits et les registres pour les nombres entiers sont classiquement de cette largeur, comme l'UAL correspondante. Les processeurs récents possèdent maintenant des registres de 64 bits.

Cependant, deux problèmes se présentent. D'abord il n'y a aucun moyen de distinguer, une fois mis dans un registre, un nombre entier d'un nombre flottant ; or les calculs doivent se faire dans des UAL différentes.

Ensuite, il existe une norme de représentation des flottants sur 64 bits et il faut bien pouvoir les mémoriser dans le processeur. Ces deux raisons font qu'il existe deux bancs de registres dans un processeur : l'**un pour stocker les nombres entiers**, reliés à l'UAL entière, et l'**autre pour les nombres flottants**, reliés à l'UAL flottante.

3. UAL ET REGISTRES

3.2 Registres

Registres généraux (3)

Chaque banc de registres est situé au plus près de l'UAL correspondante ; les transferts s'en trouvent accélérés. Il faut alors pouvoir distinguer les registres entiers et flottants dans les instructions et les modes d'adressage.

C'est pourquoi on les nomme différemment, par exemple *r0* pour un registre de type entier et *f0* pour un flottant. Les instructions arithmétiques agissant sur chaque type portent également des noms différents de sorte qu'on ne les confonde pas (**ADD et FADD** par exemple).

Ces bancs de registres sont un peu plus complexes que les circuits séquentiels traditionnels car, comme chaque instruction peut avoir deux registres comme source (par exemple **ADD r1,r2,r3**), chaque banc autorise deux accès simultanés permettant de récupérer les deux valeurs souhaitées en une seule opération.

En plus des registres généraux, chaque processeur possède des registres spécialisés nécessaires au bon déroulement des instructions.

3. UAL ET REGISTRES

3.2 Registres

Registre d'instruction

Lorsqu'une instruction est récupérée en mémoire pour être exécutée dans le processeur, elle est mémorisée dans un registre spécial :

le registre d'instruction (**RI ou IR, *Instruction Register*, voir figure 3.3**).

Le séquenceur utilise ce lieu de stockage pour disposer de l'instruction et des bits la composant pendant tout le temps de son exécution.

Il gère entièrement l'utilisation de ce registre et le programmeur n'y a jamais accès.

3. UAL ET REGISTRES

3.2 Registres

Registre d'instruction

Compteur ordinal (1)

Avant son exécution, un programme est mis en mémoire, ses instructions étant placées les unes à la suite des autres.

Chacune est donc à une adresse précise, qu'il faut envoyer au boîtier mémoire lorsque le processeur veut récupérer ladite instruction pour l'exécuter.

Il doit donc à tout moment savoir quelle est la prochaine instruction à exécuter et surtout quelle est son adresse.

Un registre spécial, appelé « **compteur ordinal** » (ou. encore, suivant les modèles de processeurs, « **PC** » pour *Program Counter*, ou « **IP** » pour *Instruction Pointer*) contient l'adresse en question (**voir figure 3.3**).

3.2 Registres

Compteur ordinal (2)

Pour exécuter une instruction, le processeur commence par envoyer un ordre de lecture mémoire associé à la valeur contenue dans PC (d'où le lien entre PC et l'interface avec le bus à la **figure 3.3**) avant de récupérer l'instruction.

Le processeur incrémente alors PC de la taille de l'instruction pour que le registre pointe sur la suivante (rappelons que chaque adresse mémoire correspond à 1 octet et qu'une instruction est longue de plusieurs octets ;

PC doit donc être augmenté du nombre d'octets de l'instruction, et non de 1). **La figure 3.12** montre le début de l'exécution d'une instruction : sa récupération à l'adresse contenue dans PC, le stockage dans RI et l'incrémentation de PC.

Le programmeur n'a pas directement accès à ce registre mais peut le modifier *via* les instructions de branchement.

3.2 Registres

Compteur ordinal (3)

Celles-ci déroutent le processeur de son parcours normal, géré automatiquement par le séquenceur (exécution linéaire des instructions par incrémentation de PC), pour lui faire reprendre l'exécution à une autre adresse mémoire. Il suffit, pour ce faire, de remplacer la valeur de PC par l'adresse cible du saut, indiquée dans l'instruction de branchement.

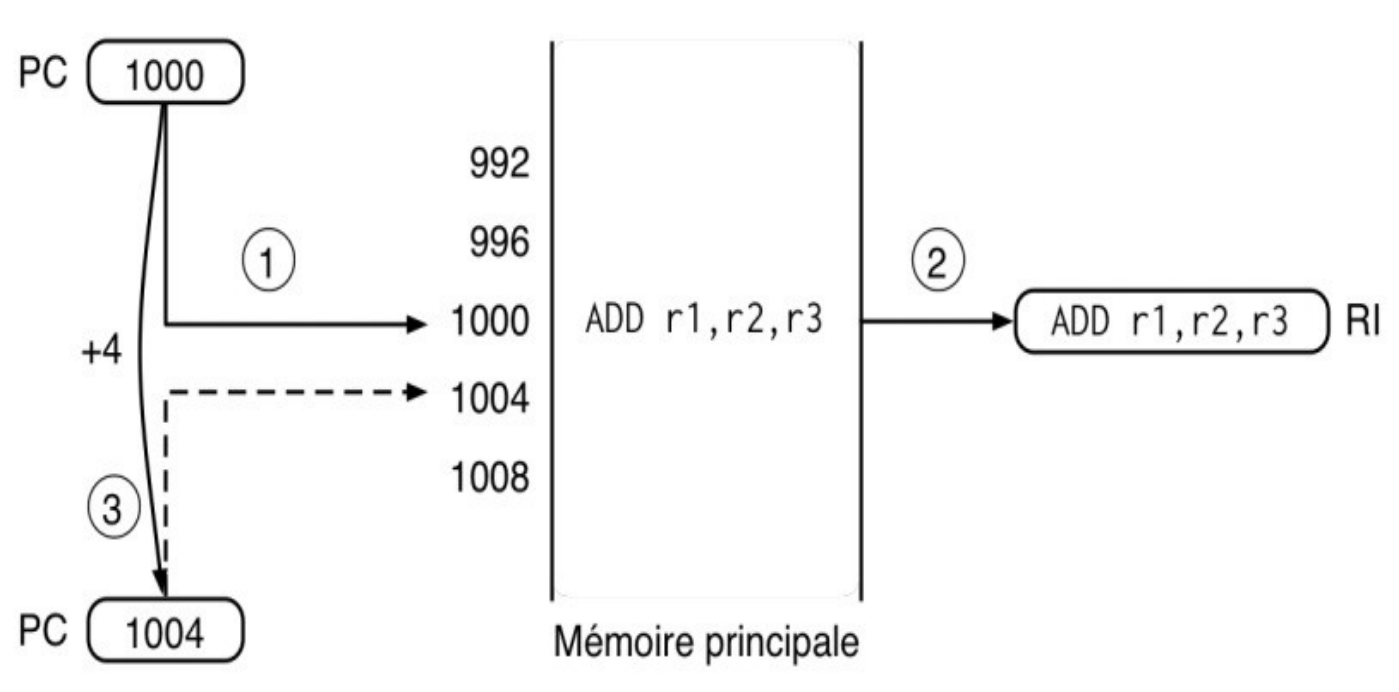


Figure 3.12 Incrémentation de PC.

3. UAL ET REGISTRES

3.2 Registres

Registre d'état (1)

Le registre d'état (***State Register*** **ou** ***Program Status Word***) est tel que ses bits ne forment pas de valeur numérique mais servent d'indicateurs (aussi appelés « **drapeaux** » ou ***flags***) sur l'état du processeur.

Certains bits peuvent être positionnés par le programmeur pour demander un comportement particulier (**par exemple fixer un niveau de masquage des interruptions, voir chapitre 4**), d'autres (appelés « **bits conditions** ») sont automatiquement mis à jour par le séquenceur à la fin de l'exécution de chaque instruction.

Ils sont utilisés pour les sauts conditionnels en liaison avec les instructions de test et de comparaison.

3. UAL ET REGISTRES

3.2 Registres

Registre d'état (2)

La « **condition** » liée à un branchement conditionnel consiste toujours en un test d'une valeur particulière d'un ou plusieurs bits conditions, positionnés par l'instruction précédente. Certains de ces bits, tels ceux qui suivent, sont suffisamment universels pour qu'on les retrouve sur tous les modèles de processeurs (mais leur nom peut varier) :

• Le bit **Z (*Zero*)**. Il est mis à 1 si le résultat de l'instruction est nul, sinon il est mis à 0.

• Le bit **C (*Carry*)**. Il est mis à 1 si l'instruction génère une retenue finale, sinon il est mis à 0.

• Le bit **N (*Negative*)**. Il est mis à 1 si le résultat de l'instruction est négatif, sinon il est mis à 0 ; c'est la recopie du bit de poids fort du résultat.

• Le bit **V (*oVerflow*)**. Il est mis à 1 si l'instruction génère un débordement arithmétique, sinon il est mis à 0.

3.2 Registres

Registre d'état (3)

L'utilisation de ces bits conditions se fait toujours de la même manière : d'abord, on effectue un calcul, un test ou une comparaison, positionnant les bits conditions suivant le résultat que l'on veut obtenir, puis l'on effectue un saut conditionnel sur ces bits.

Supposons par exemple que l'on veuille tester si le registre *r1* est nul. On peut écrire le code de la façon suivante :

ADD r0,r1,#0

JZS adresse_cible

3.2 Registres

Registre d'état (4)

La première instruction additionne *r1* et 0 et met le résultat dans *r0*. Ce résultat ne peut être nul que si *r1* luimême était nul au départ. Le bit Z n'est donc à 1 après cette instruction que si *r1* était nul.

Le branchement n'est effectif que si le bit Z est à 1 (**JZS signifie *Jump if bit Z Set*, soit « sauter si le bit Z est à 1 »**).

On a donc bien deux exécutions différentes suivant la valeur de *r1* : s'il est non nul, le processeur continue avec les instructions situées après le branchement ; s'il est nul, le programme se poursuit à l'adresse cible indiquée dans le branchement emprunté.

Notez que, dans certains processeurs, de plus en plus nombreux, les tests sont directement réalisés sur le contenu des registres, et non plus sur un registre d'état dont la mise à jour est délicate dans le cas d'une architecture incluant un pipeline, une exécution dans le désordre, etc.

3. UAL ET REGISTRES

3.2 Registres

Registre pointeur de pile (1)

Lors de l'exécution de programmes, le processeur doit parfois stocker de l'information en mémoire, non pas à la demande explicite du programmeur, mais afin d'assurer le bon fonctionnement des instructions.

Un exemple typique est l'appel de fonction : dans un langage évolué, le programme est dérouté pour que s'exécute ladite fonction puis reprend normalement là où il s'était arrêté (à l'inverse d'un saut qui le déroute définitivement à une autre adresse).

Le même mécanisme existe en assembleur ; il faut alors mémoriser la valeur de PC avant d'exécuter la fonction pour pouvoir reprendre le programme après l'appel.

3. UAL ET REGISTRES

3.2 Registres

Registre pointeur de pile (2)

À cette fin, une structure de pile est implémentée en mémoire et un registre spécial du processeur (appelé « **pointeur de pile** » ou **SP, *Stack Pointer***) pointe sur le haut de la pile, c'est-à-dire sur la première case mémoire libre à son sommet.

Là sont empilées ou dépilées des informations, soit automatiquement par le séquenceur, soit à la demande du programmeur.

3.2 Registres

Registre pointeur de pile (3)

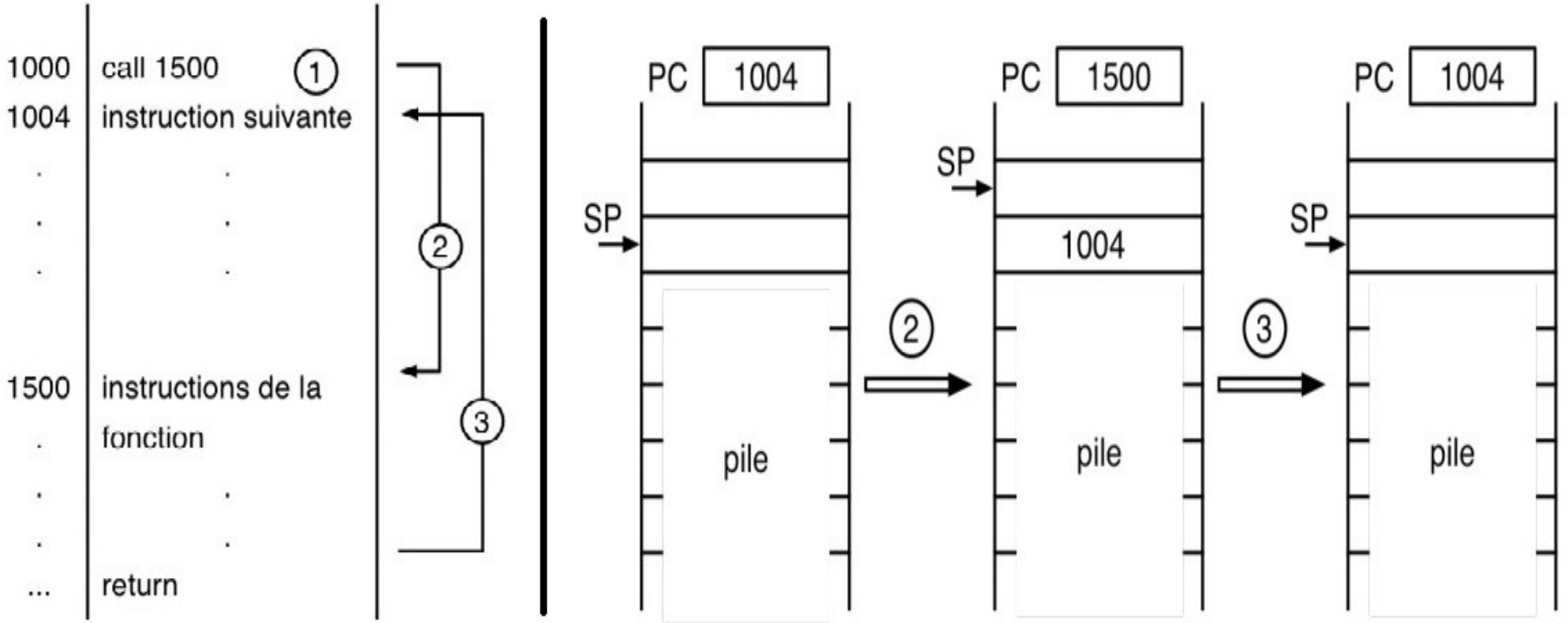
Des instructions spéciales (**PUSH** ou **POP**) sont à sa disposition, qui accèdent à la mémoire *via* le registre SP par un adressage indirect.

La figure 3.13 illustre l'usage de SP lors de l'appel d'une fonction pour mémoriser l'adresse de retour.

De façon générale, la pile sert également à passer des paramètres à une fonction et à stocker ses variables locales :

ces deux types d'informations n'ont pas d'existence en dehors de la fonction en question et, à l'entrée de celle-ci, la pile permet justement de les créer (par empilement), et de les détruire à la sortie (par dépilement).

3.2 Registres



① Appel de la fonction située à l'adresse 1500. L'adresse de retour est 1004.

② On empile l'adresse de retour qui se trouve dans PC.

③ On dépile dans PC ce qu'il y a au sommet de la pile. La prochaine instruction à exécuter est donc à l'adresse 1004.

Figure 3.13 Appel de fonction.

4. SÉQUENCEUR

Le séquenceur est directement lié à la puissance du processeur puisqu'il est responsable du bon déroulement des instructions.

Il doit prendre en charge tous les transferts de données et l'envoi de toutes les commandes aux autres composants internes.

4. SÉQUENCEUR

4.1 Cycle d'instruction (1)

L'exécution d'une instruction peut se décomposer en un certain nombre d'étapes composant le cycle d'instruction :

• récupération de l'instruction et mise à jour de PC (*Fetch 1*) ;

• décodage de l'instruction (*Decode*) ;

• récupération des données (*Fetch 2*) ;

• exécution de l'instruction (*Execute*) ;

• écriture du résultat et modification des bits conditions (*Write*).

Chaque étape est gérée par le séquenceur et transformée en ordres internes. **La figure 3.14** présente un exemple de l'exécution en interne de l'instruction ADD r1,r2,r3.

4. SÉQUENCEUR

4.1 Cycle d'instruction (2)

Chaque instruction demande ainsi de nombreux ordres et transferts internes, entièrement contrôlés par le séquenceur et rythmés par son horloge. Plus les instructions du processeur sont compliquées, plus elles nécessitent d'étapes internes et plus le séquenceur est complexe.

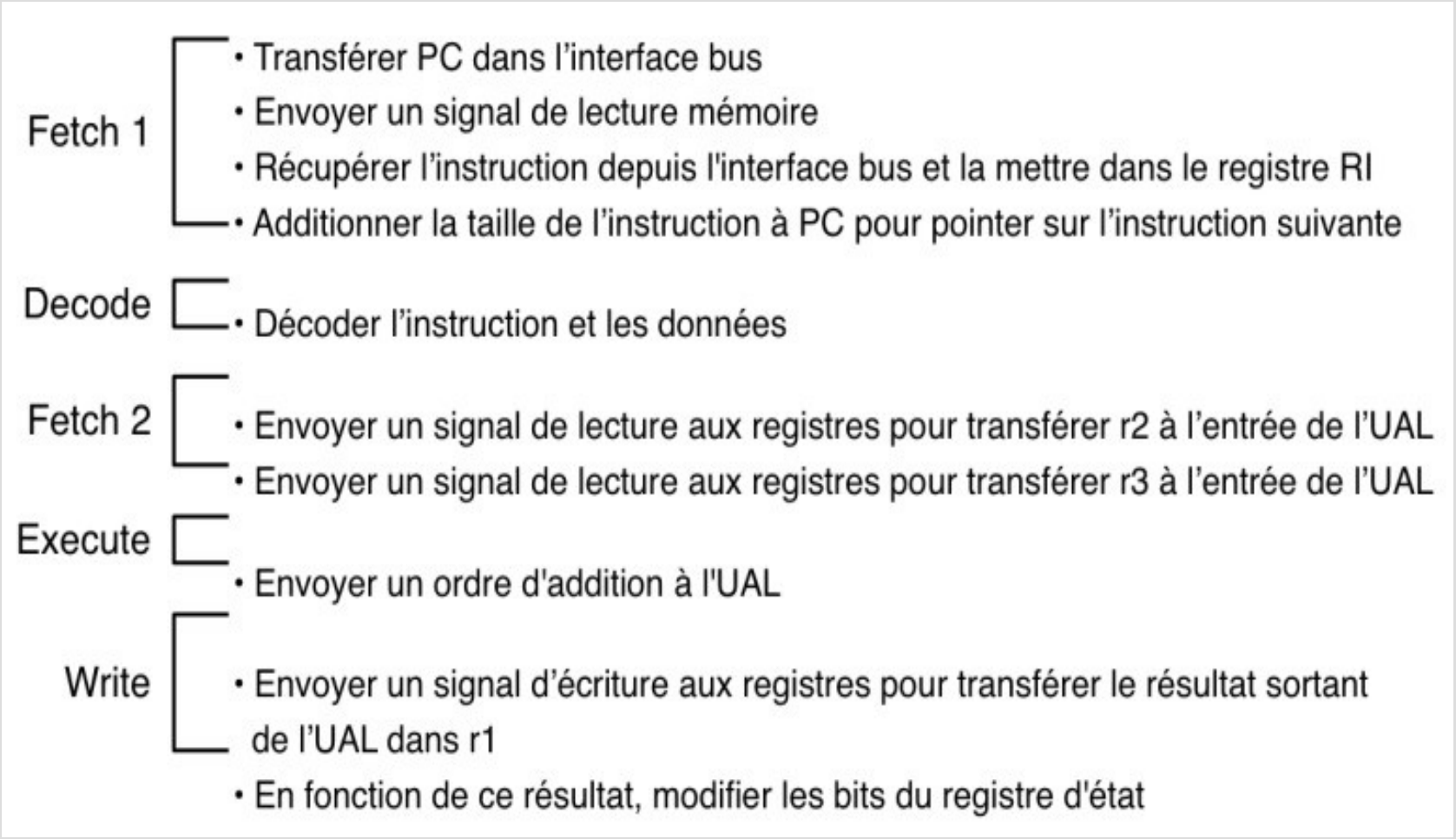


Figure 3.14 Exécution de ADD r1,r2,r3.

4. SÉQUENCEUR

4.2 Séquenceur câblé (1)

Les premiers microprocesseurs avaient des séquenceurs simples, qui calculaient les commandes internes à envoyer en fonction du code numérique de l'instruction à exécuter.

Ces séquenceurs étaient composés de simples circuits logiques combinatoires, qui généraient les ordres à l'aide de fonctions logiques à partir des bits composant le code numérique de l'instruction.

Ceux-ci n'étaient donc pas choisis au hasard mais devaient correspondre aux lignes de commande à activer (transfert de données, commande à l'UAL…) ; ils s'agissait de séquenceurs câblés.

4. SÉQUENCEUR

4.2 Séquenceur câblé (2)

Ce système de séquenceur est assez simple, ne nécessite pas de circuits compliqués, mais a une puissance limitée.

En effet, on ne peut pas obtenir une infinité de fonctions logiques différentes à partir des bits du code de l'instruction et cela restreint les possibilités pour les commandes internes.

Une instruction processeur complexe, demandant de nombreuses étapes internes pour son exécution, ne peut être implémentée avec un séquenceur câblé.

Pour ce faire, il faut mettre en place un séquenceur plus compliqué, capable d'exécuter des micro-instructions.

4. SÉQUENCEUR

4.3 Séquenceur microprogrammé (1)

Une instruction peut se décomposer en plusieurs micro-instructions, chacune correspondant à une action interne du processeur.

Pour augmenter les possibilités de ce dernier, on a transformé le séquenceur en processeur miniature, exécutant, pour chaque instruction, un programme formé des micro-instructions.

À l'intérieur du séquenceur se trouve une mémoire de microprogramme contenant la traduction en microinstructions de chaque instruction processeur, ainsi qu'un microcompteur ordinal chargé de gérer l'exécution de celles-ci.

Lorsqu'une instruction processeur est récupérée, son code numérique localise l'adresse de début du microprogramme d'exécution dans cette mémoire.

4. SÉQUENCEUR

4.3 Séquenceur microprogrammé (2)

Le microcompteur ordinal exécute alors les micro instructions dans l'ordre, chacune opérant un transfert interne de données ou envoyant une commande à un composant du processeur.

L'avantage de cette technique est de pouvoir proposer des instructions processeur plus complexes car il n'y a aucune limite au nombre de micro-instructions traduisant une instruction processeur.

En revanche, cela complique et ralentit fortement le séquenceur, qui occupe beaucoup plus de place sur la puce (au détriment des autres entités fonctionnelles) car il faut y mettre la mémoire de microprogramme ainsi que les circuits nécessaires à l'exécution des micro-instructions.

4. SÉQUENCEUR

4.3 Séquenceur microprogrammé (3)

La mémoire de microprogramme est bien sûr conçue une bonne fois pour toutes par le constructeur du processeur et n'est pas modifiable.

Le programmeur n'y a pas accès, ce mécanisme étant totalement invisible pour lui ; en d'autres termes, le niveau des instructions processeur est à sa portée, mais pas le niveau inférieur.

Les processeurs actuels essaient de combiner les deux techniques de contrôle en exécutant les instructions processeur simples par des circuits logiques rapides et en réservant le découpage en micro-instructions aux instructions processeur les plus complexes.

4. SÉQUENCEUR

4.4 RISC contre CISC (1)

Durant la décennie **1970-1980**, les processeurs ont progressé en complexité, profitant de l'intégration poussée des transistors pour offrir des jeux d'instructions plus complets et un séquenceur microprogrammé, malheureusement au détriment des autres composants internes, restreints à la portion congrue sur la puce.

Entre autres, les données des instructions pouvaient se trouver en registre bien sûr, mais également en mémoire, même pour les opérations arithmétiques (d'où les liens en pointillés à la **figure 3.3**, entre l'UAL et l'interface avec le bus).

4. SÉQUENCEUR

4.4 RISC contre CISC (2)

En **1981**, deux universitaires ont suggéré de réduire drastiquement le jeu d'instructions des processeurs, par exemple en interdisant les opérandes situés en mémoire, sauf pour les transferts, en limitant les modes d'adressage possibles, et en proscrivant les instructions trop compliquées.

Ils avaient constaté que seulement 20 % du jeu d'instructions était utilisé dans 80 % des instructions d'un programme standard.

Ils ont alors proposé de construire un processeur ayant un jeu d'instructions réduit (**RISC,** ***Reduced Instruction Set Computer***) par opposition aux processeurs existants (**CISC,** ***Complex Instruction Set Computer***).

4. SÉQUENCEUR

4.4 RISC contre CISC (3)

Cela devait permettre de simplifier fortement le séquenceur, de revenir à un séquenceur câblé, d'introduire un pipeline (**voir section suivante**),

d'optimiser le tout pour accroître la vitesse d'horloge, et d'utiliser la place libérée sur la puce pour augmenter le nombre de registres (et par là même limiter le nombre d'accès mémoire) et introduire la mémoire cache directement dans le processeur.

La réduction du jeu d'instructions entraînant un allongement des programmes (car certaines instructions disponibles sur les processeurs **CISC** doivent être simulées par plusieurs instructions successives sur les processeurs **RISC**),

les promoteurs de ces processeurs espéraient alors que le ralentissement induit serait plus que compensé par l'accélération des performances des processeurs.

4. SÉQUENCEUR

4.4 RISC contre CISC (4)

En fait, le passage d'un processeur **CISC** à un processeur **RISC** transférait la complexité du séquenceur au compilateur, qui devait optimiser le code pour utiliser le pipeline et profiter des nombreux registres.

Pendant longtemps, les partisans des deux types de processeurs se sont affrontés sur le terrain des performances pour aboutir à une synthèse dans les ordinateurs actuels :

-les techniques des processeurs **RISC** (séquenceur câblé, pipeline, nombreux registres, cache…) ont été intégrées à tous les processeurs;

-tandis que les instructions **CISC** sont toujours présentes *via* un séquenceur plus compliqué, réservé à leur usage.

5. ARCHITECTURES ÉVOLUÉES

5.1 Opérateurs et instructions spécialisés

Les processeurs ont profité des possibilités offertes par l'adjonction de transistors supplémentaires pour implémenter l'exécution d'instructions arithmétiques plus compliquées.

Il y a d'abord eu l'inclusion de l'UAL flottante directement dans le processeur, ce qui a permis d'effectuer rapidement des calculs sur les nombres réels, en commençant par les opérations simples puis en ajoutant progressivement des fonctions mathématiques complexes (racine carrée, fonctions trigonométriques et logarithmiques…).

Devant le développement des applications multimédias, tous les fabricants de processeurs ont ensuite inclus des instructions spécifiques à ces applications dans les jeux d'instructions de leurs processeurs.

Ces instructions donnent au développeur et au compilateur la possibilité d'effectuer certains calculs spécialisés en une seule opération. Ces extensions sont de deux types : des instructions mathématiques particulières et le calcul vectoriel.

5. ARCHITECTURES ÉVOLUÉES

5.2 Traitement pipeline (1)

Chaque instruction peut être décomposée en plusieurs étapes (la lecture de l'instruction, son décodage, son exécution, etc. (**voir section 4.1**), nécessitant chacune des circuits différents.

Ces derniers travaillent pendant une partie du temps d'exécution, mais restent inactifs lorsque l'instruction n'est pas à l'étape concernée.

Le principe du **traitement pipeline** consiste à ne pas attendre la fin de l'exécution d'une instruction pour lancer la suivante, en profitant de l'inactivité des circuits ayant déjà traité l'instruction en cours.

C'est l'idée du travail à la chaîne : un circuit effectue une étape d'exécution et enchaîne immédiatement la même étape avec l'instruction suivante pendant que la première avance dans la chaîne de traitement.

5.2 Traitement pipeline (2)

La **figure 3.15** illustre ce concept en décomposant une instruction en cinq étapes. Chaque chiffre correspond à une instruction avançant dans le pipeline et dont l'exécution se termine à sa sortie.

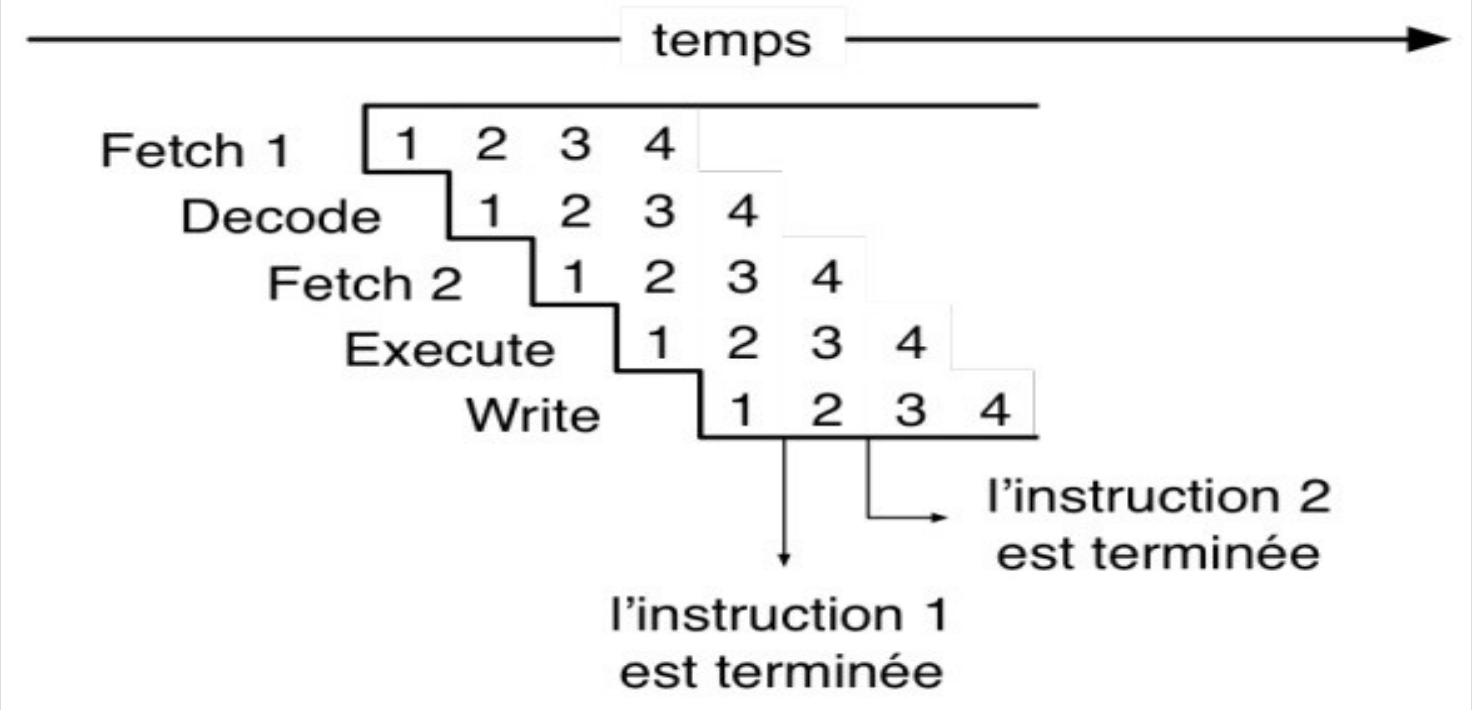


Figure 3.15 Traitement pipeline.

5. ARCHITECTURES ÉVOLUÉES

5.2 Traitement pipeline (3)

Le gain ne se fait pas directement sur la vitesse de traitement d'une instruction (elle doit toujours passer par les cinq étapes du **pipeline**) mais sur le nombre d'instructions par seconde qui sont exécutées. Dans le meilleur des cas, le processeur peut traiter cinq fois plus d'instructions qu'un système **non pipeliné**.

De plus, les circuits propres à chaque étape, plus simple, peuvent être optimisés pour fonctionner à une vitesse d'horloge plus rapide et accroître la vitesse du processeur. On a donc tout intérêt à fractionner le plus possible le pipeline pour augmenter le nombre d'instructions en cours d'exécution et optimiser au maximum les circuits correspondant à chaque étape.

Certains processeurs ont ainsi des **pipelines** pouvant contenir une vingtaine voire une trentaine d'étapes.Plusieurs écueils peuvent quand même perturber cette belle organisation et freiner l'exécution des instructions.

5. ARCHITECTURES ÉVOLUÉES

5.3 Processeur superscalaire (1)

Un **pipeline** permet de traiter plus d'instructions par cycle mais chacune d'elles se trouve à une étape de traitement différente. Pour pouvoir traiter plusieurs instructions en même temps (on parle alors d'**exécution parallèle**), les **processeurs superscalaires** ont été dotés de multiples unités d'exécution dans leur **pipeline**, voire de plusieurs **pipelines** complets (**voir figure 3.18**).

L'objectif de ces processeurs consiste alors à alimenter, à chaque cycle, chaque unité d'exécution. Dans ces conditions, le processeur délivre le maximum de sa puissance puisque aucune unité d'exécution ne reste inoccupée durant les cycles.

Avoir plusieurs circuits chargés de l'exécution permet de traiter une instruction compliquée (prenant beaucoup de temps), comme une addition flottante, sans bloquer le reste du **pipeline**, qui peut passer par les autres unités d'exécution.

5. ARCHITECTURES ÉVOLUÉES

5.3 Processeur superscalaire (2)

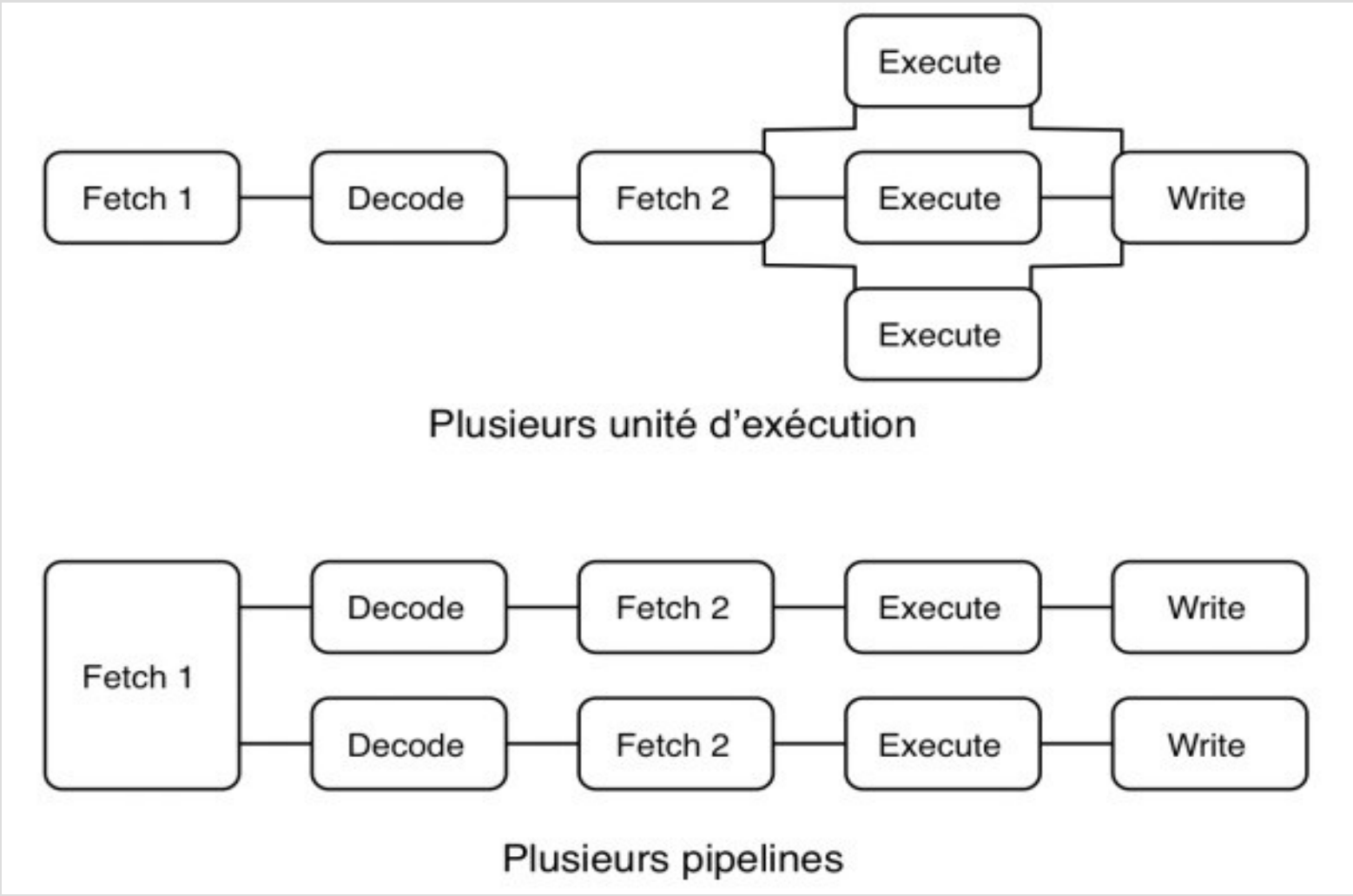


Figure 3.18 Processeur superscalaire.

5. ARCHITECTURES ÉVOLUÉES

5.3 Processeur superscalaire (3)

Il faut pour cela que les instructions successives n'aient pas de liens de dépendance et ne fassent pas appel aux mêmes circuits. On pourra ainsi probablement exécuter en parallèle une opération arithmétique entière et une opération flottante qui n'utilisent pas la même UAL et ne font pas référence aux mêmes registres.

De plus, la duplication des unités d'exécution permet de dédier ces dernières au traitement d'instructions du même type (par exemple, une unité d'exécution entière pour toutes les instructions manipulant des données entières, une unité d'exécution flottante pour la manipulation des données flottantes, une unité d'accès mémoire pour les lecture et écriture de données en mémoire…).

Cette spécialisation des unités d'exécution permet d'espérer un accroissement des performances et une meilleure utilisation globale du processeur.

5. ARCHITECTURES ÉVOLUÉES

5.4 Exécutions parallèles (1)

Un processeur superscalaire est en quelque sorte composé de deux blocs fonctionnels : l'unité de chargement et d'ordonnancement des instructions, et les unités d'exécution.

La première est chargée de récupérer les instructions en mémoire et de les réorganiser pour essayer, à chaque cycle, d'en envoyer le maximum aux unités d'exécution.

Elle essaie d'extraire le parallélisme du programme en fonction de la durée d'exécution de chacune des instructions et des dépendances entre elles.

Le second bloc est formé des différents pipelines d'exécution, chargés respectivement des calculs sur les entiers, les nombres flottants, ou encore du calcul des adresses lors des accès mémoire, etc.

5. ARCHITECTURES ÉVOLUÉES

5.4 Exécutions parallèles (2)

Les performances optimales sont bien sûr obtenues lorsque toutes les ressources du processeur sont utilisées au maximum. Malheureusement, cela est rarement le cas.

En raison du parallélisme limité des programmes, l'unité d'ordonnancement ne peut pas envoyer, à chaque cycle, le maximum d'instructions possible aux unités d'exécution (celles-ci étant de plus en plus nombreuses au sein des processeurs).

Elle émet en moyenne, pour des programmes classiques, deux instructions par cycle, alors que les unités d'exécution sont souvent plus nombreuses (il est parfois possible de traiter plus de deux instructions simultanément).

5.4 Exécutions parallèles (3)

De même, les unités d'exécution ne sont pas occupées à 100 % : comme elles sont spécifiques à certains types d'instructions, elles peuvent être vides si ces instructions ne sont pas présentes.

Les pipelines peuvent également avoir des retards et des trous si certaines instructions demeurent plus d'un cycle à une étape en raison d'une difficulté d'exécution.

Cela est résumé à la **figure 3.19**, où les cases vides symbolisent les ressources inexploitées lors d'un cycle.

5.4 Exécutions parallèles (4)

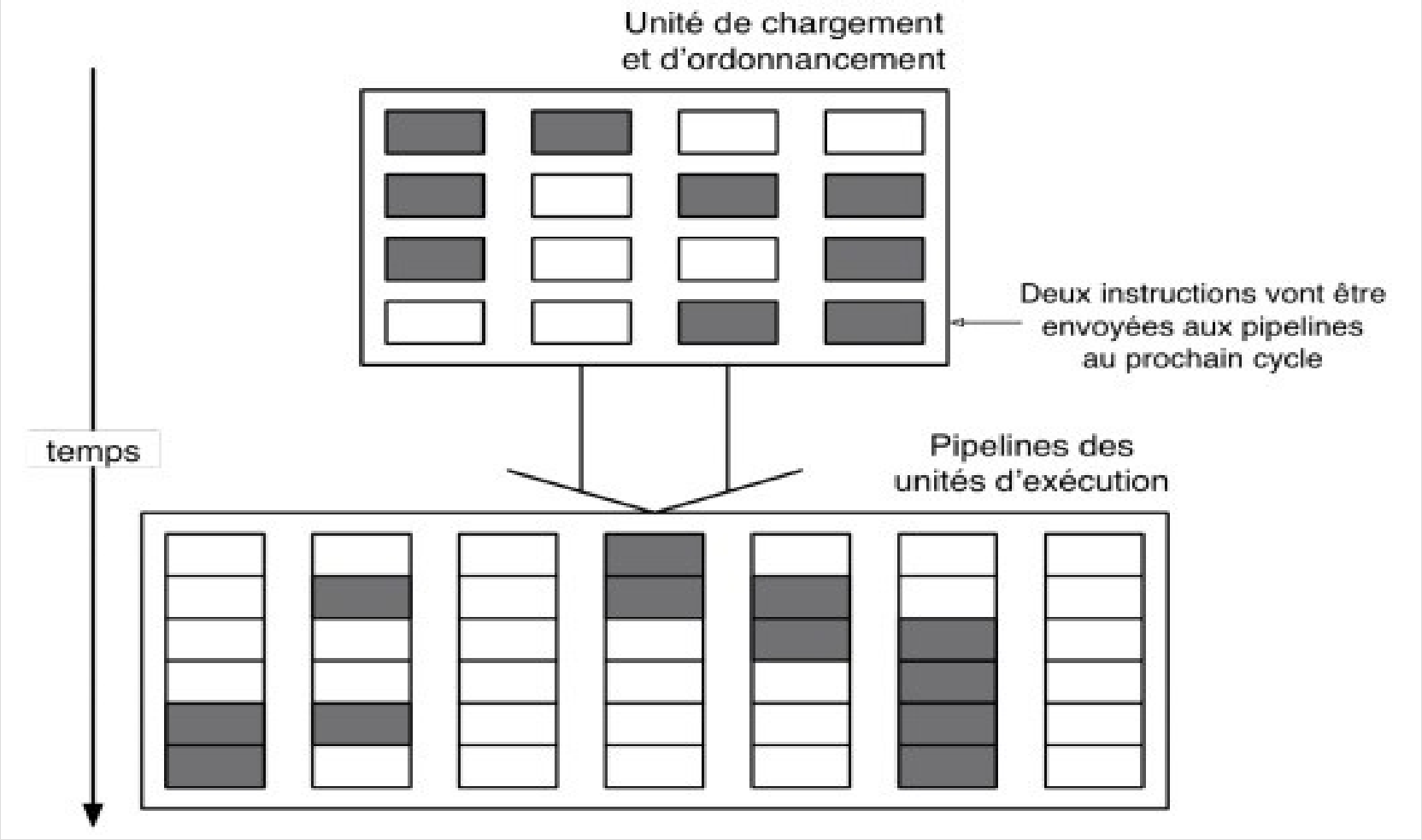


Figure 3.19 Chargement et exécution des instructions.

5.4 Exécutions parallèles (4)

Exécution multithread (1)

Un ordinateur a toujours plusieurs programmes en cours d'exécution, appelés « **processus** » **ou** « **threads** ».

Le système d'exploitation est chargé de les ordonner en découpant le temps en tranches (d'une centaine de millisecondes le plus souvent) et en attribuant ces tranches successivement à chacun des threads.

Le processeur exécute alors les instructions d'un thread pendant un temps relativement long, puis s'occupe des autres processus avant de revenir au premier. Pour augmenter l'occupation des ressources internes du processeur (et donc améliorer ses performances en exécutant plus d'instructions par cycle), on a essayé d'exécuter consécutivement plusieurs threads.

5.4 Exécutions parallèles (4)

Exécution multithread (2)

À chaque cycle, le processeur tente de dédier ses ressources (unité d'ordonnancement et unités d'exécution) aux instructions d'un seul thread.

Mais à chaque nouveau cycle (ou, suivant les implémentations, après quelques cycles), le processeur change de thread et, en exécutant les instructions de ce dernier, il est ainsi possible de compenser un retard dans un pipeline.

5.4 Exécutions parallèles (4)

Exécution multithread (3)

De son côté, l'unité d'ordonnancement envoie le maximum d'instructions du thread traité aux unités d'exécution (**voir figure 3.20**).

Les performances sont améliorées car certaines dépendances disparaissent naturellement à l'exécution (les instructions d'un même thread sont plus « **espacées** » dans le pipeline) et il en résulte une meilleure utilisation globale des unités d'exécution.

De plus, l'impact des retards dus aux accès mémoire (**voir chapitres 5 et 6**) est diminué (au lieu d'attendre, le processeur exécute quelques instructions d'un autre thread).

5.4 Exécutions parallèles (4)

Exécution multithread (4)

Cependant, il reste encore des trous car le processeur ne peut pas combiner deux threads dans un même cycle (chaque ligne horizontale contient des instructions d'un seul thread). Si les deux ont un faible parallélisme, peu d'instructions sont exécutées à chaque cycle.

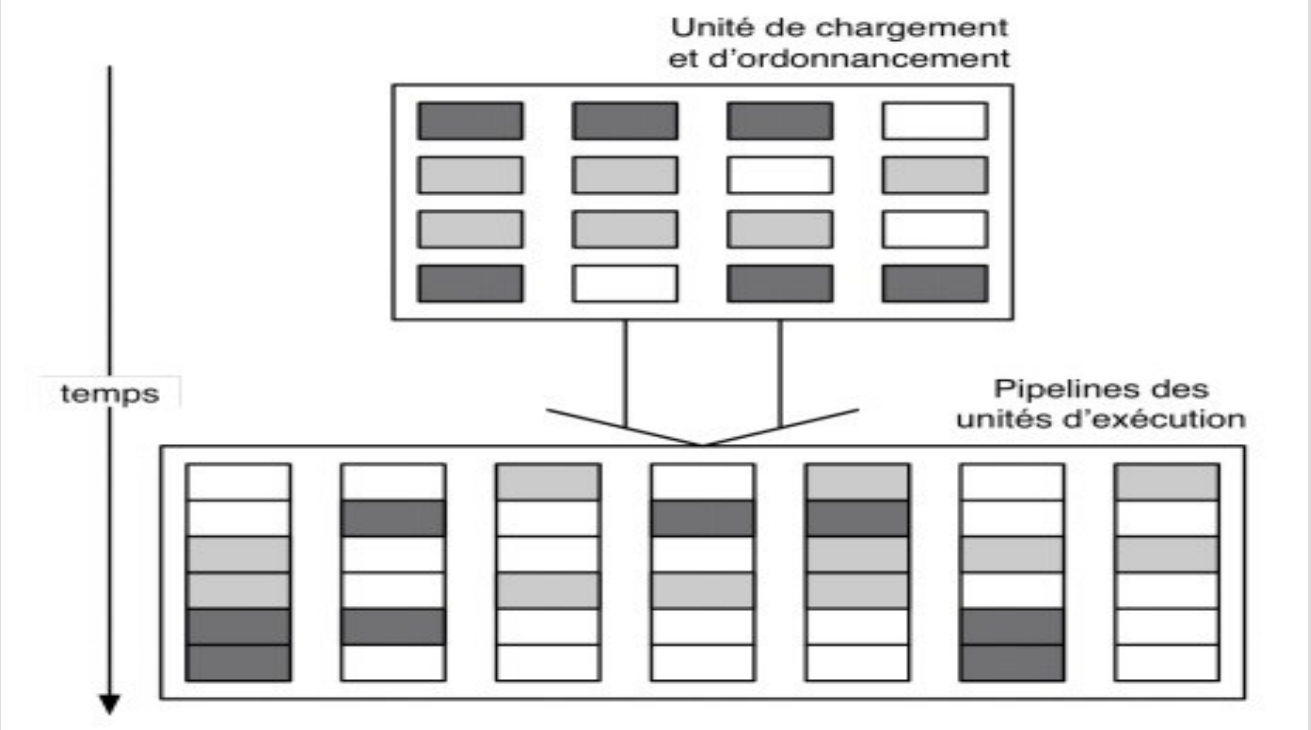


Figure 3.20 Exécution multithread.

5.4 Exécutions parallèles (4)

Exécution multithread simultanée (1)

En supprimant la contrainte d'un seul thread par cycle, on peut maximiser l'utilisation du processeur : c'est l'exécution SMT (***Simultaneous MultiThreading***) ou hyperthreading (nom commercial donné par le fabricant Intel).

À chaque cycle, le processeur essaie d'exécuter le maximum d'instructions possible, indépendamment de leur provenance (**voir figure 3.21**). Il y a encore moins de trous dans l'unité d'ordonnancement et les **pipelines** : le processeur exécute encore plus d'instructions par cycle.

5.4 Exécutions parallèles (4)

Exécution multithread simultanée (2)

L'accroissement en complexité du séquenceur (les instructions de threads différents ne doivent pas interférer entre elles, par exemple pour l'utilisation des registres) est largement compensé par l'augmentation des performances :

sur les processeurs d'Intel, l'implémentation de l'hyperthreading augmente de 5 % la surface de la puce pour un gain de vitesse d'environ 30 %.

Pour que le processeur puisse lancer deux threads simultanément par hyperthreading, il faut informer le système d'exploitation de cette possibilité, en lui présentant le processeur embarquant cette technologie comme deux (ou plus) processeurs logiques.

Le système tentera d'exécuter deux threads (ou plus, un sur chaque « processeur »), qui seront traités simultanément par un seul processeur physique.

5.4 Exécutions parallèles (4)

Exécution multithread simultanée (3)

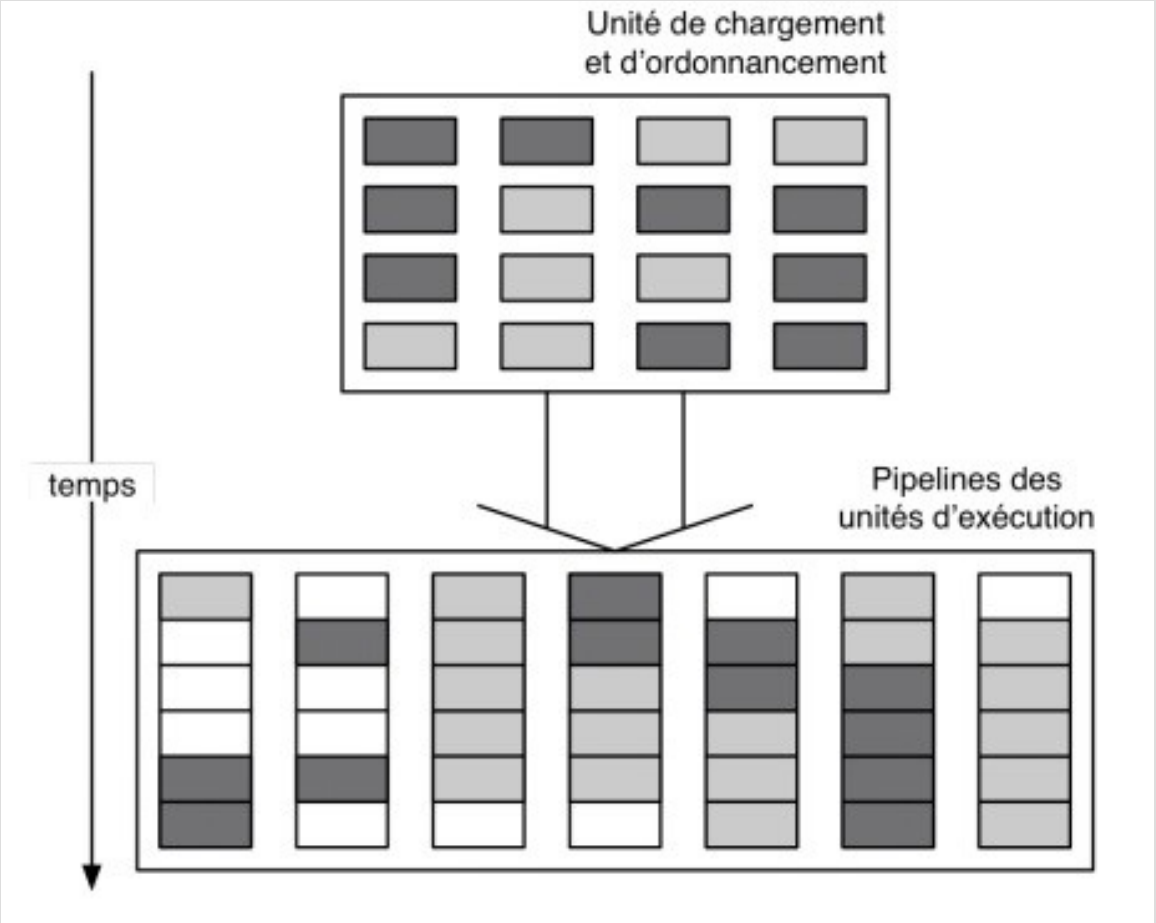


Figure 3.21 Exécution SMT.

5.4 Exécutions parallèles (4)

Exécution multithread simultanée (4)

Processeurs multicœurs

Les processeurs les plus récents poussent la logique encore plus loin en dupliquant leur cœur : unité d'ordonnancement, unités d'exécution et souvent le cache de niveau 1.

On les dit « **dualcore** », ou « **multicore** » s'il y en a plus que deux. On retrouve la possibilité d'exécuter deux threads simultanément, sans la complexité de l'hyperthreading, puisque les deux chemins d'exécution sont physiquement séparés.

6. CONCLUSION

Un ordinateur contient un processeur qui exécute en séquence des instructions de transfert ou de calcul se trouvant en mémoire principale.

Il autorise également les ruptures de séquence, ce qui permet d'avoir un comportement différent suivant les données d'un programme, en agissant directement sur le compteur ordinal.

Ces instructions sont traitées par un séquenceur, **chef d'orchestre interne**, qui envoie des commandes à l'unité arithmétique et logique et aux registres du processeur.

Ce séquenceur, plus ou moins compliqué, est le composant qui a connu le plus de perfectionnements ces dernières années (introduction d'un pipeline, processeur superscalaire, exécution multithread simultanée), permettant un accroissement sans précédent des performances et surtout de la rapidité d'exécution des instructions.