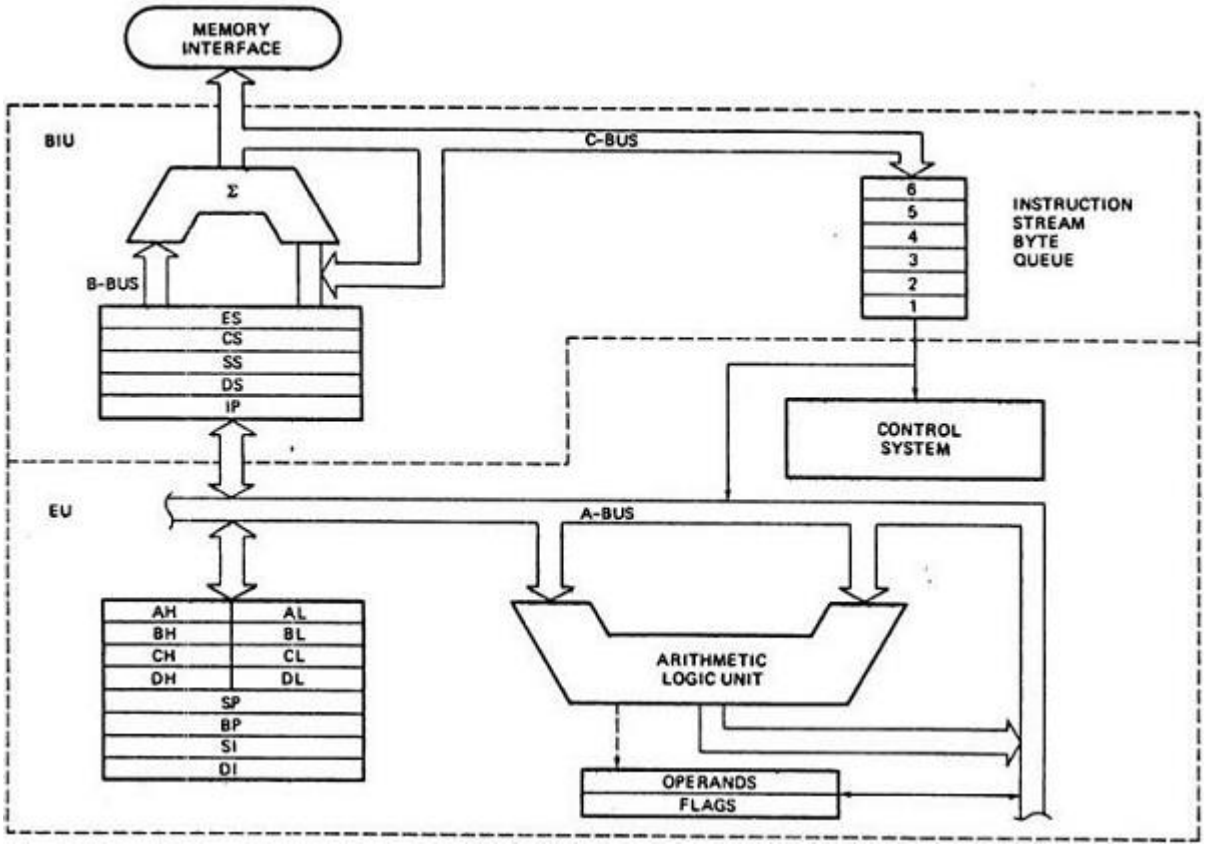
Les composants du CPU



Objectifs du cours

Ce chapitre se concentre sur les **composants internes du CPU** (Central Processing Unit) et leur rôle dans le fonctionnement de l'ordinateur. Le CPU, souvent considéré comme le « cerveau » de l'ordinateur, se compose de plusieurs unités spécifiques, chacune dédiée à une fonction essentielle. Ce chapitre présente en détail ces unités, comme l'**unité de contrôle** (qui dirige l'exécution des instructions), l'**unité arithmétique et logique** (ALU, responsable des opérations mathématiques), ainsi que les **registres**, qui stockent temporairement des données.

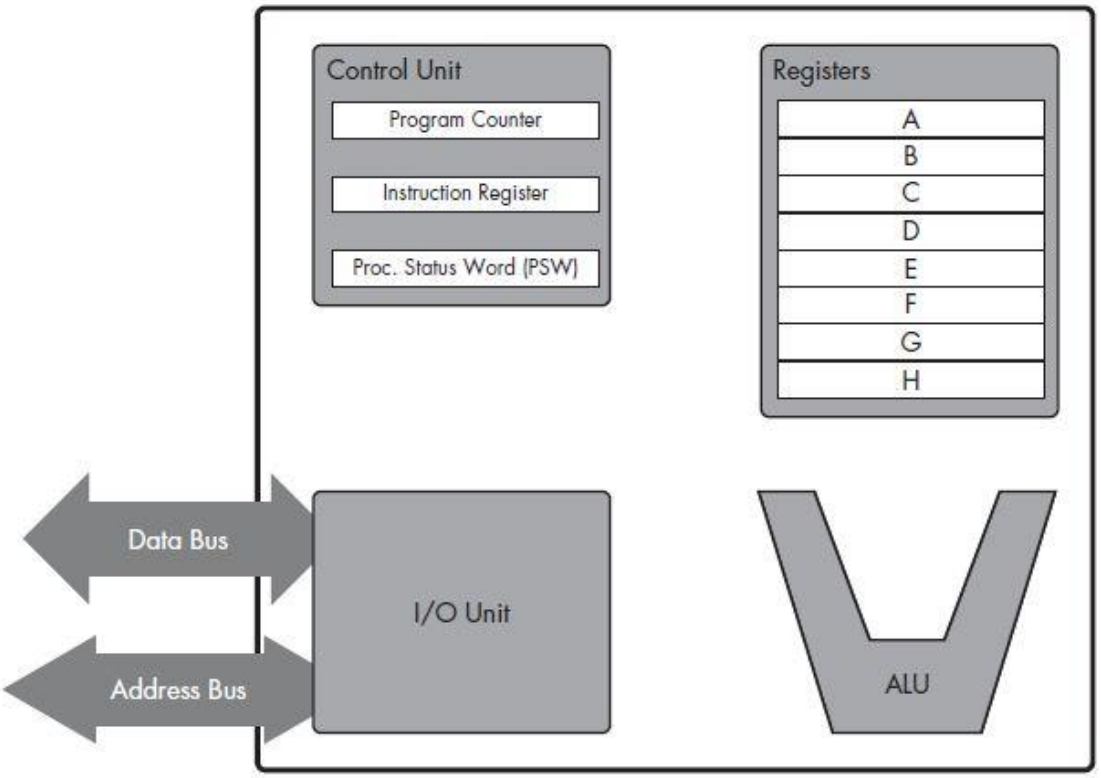
En plus de décrire ces composants, nous explorerons les techniques avancées qui ont permis d'augmenter les performances des CPU modernes, notamment le **pipeline**. Ce procédé, qui divise l'exécution des instructions en plusieurs étapes, permet un traitement plus rapide et efficace. Nous aborderons également les avantages et limites du pipeline, en soulignant les défis associés, tels que les **dépendances de données** et les **brouillages** (hazards).

Une section spécifique sera consacrée aux méthodes de **mesure des performances** des processeurs, essentielles pour évaluer et comparer les CPU. Nous examinerons des indicateurs courants comme le **nombre d'instructions par seconde (IPS)** et le **temps d'exécution**.

Enfin, pour illustrer le fonctionnement du CPU en pratique, ce chapitre inclura l'utilisation d'un **simulateur de CPU**. Ce simulateur permettra de suivre, étape par étape, l'exécution d'un programme et d'observer les interactions entre les différents composants. Cette approche interactive facilitera la compréhension des concepts théoriques et renforcera l'apprentissage des mécanismes internes du CPU.

Les composants du CPU et leurs fonctions

Le **Central Processing Unit** (CPU) est structuré autour de divers composants, chacun chargé d'accomplir des tâches spécifiques pour exécuter les programmes informatiques. Chacun de ces composants contribue à une fonction particulière nécessaire au traitement des instructions et au bon déroulement des opérations au sein de l'ordinateur.



Adressage des instructions :

L'une des premières tâches du CPU est de **localiser l'instruction suivante** à exécuter. Cette tâche est réalisée par le **compteur ordinal** (Program Counter ou PC), qui pointe vers l'adresse mémoire de l'instruction en cours. Il s'incrémente automatiquement après chaque cycle d'instruction pour accéder à l'instruction suivante, permettant ainsi une exécution séquentielle du programme.

Décodage des instructions :

Une fois l'instruction récupérée, le CPU doit en comprendre la nature. Cette étape est prise en charge par le **registre d'instruction** (Instruction Register ou IR), qui décode l'instruction en la traduisant dans un format que les autres unités du CPU peuvent comprendre. L'IR identifie le type d'opération à réaliser

et, si nécessaire, envoie des informations aux autres unités pour qu'elles se préparent à l'exécution.

Transfert de données (Load and Store Unit) :

Le CPU doit régulièrement échanger des données entre ses registres internes et la mémoire centrale ou les périphériques d'entrée/sortie. La **Load and Store Unit** gère ces transferts, permettant au CPU de charger des données en mémoire dans des registres pour traitement, ou d'écrire des résultats en mémoire après traitement. Cette unité joue un rôle essentiel pour maintenir le flux de données entre le CPU et les autres composants.

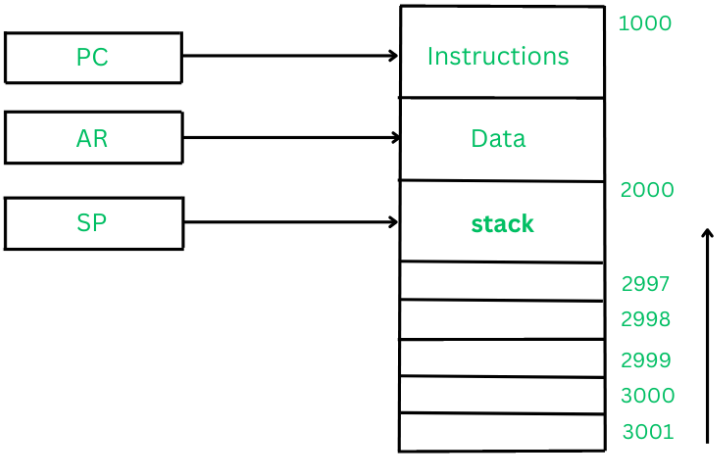
Gestion des sauts et conditions (Branch Unit) :

Pour exécuter des programmes efficacement, le CPU doit être capable de gérer les instructions de **saut**, conditionnelles ou non. Ces instructions permettent au programme de bifurquer vers une autre adresse mémoire au lieu de suivre une séquence linéaire. La **Branch Unit** est responsable de gérer ces sauts en fonction des conditions prédéfinies dans le programme, comme les comparaisons entre valeurs ou la vérification d'états spécifiques.

Surveillance des instructions exécutées (Process Status Word - PSW) :

Le **Process Status Word** (PSW) est un ensemble de **bits d'état** qui fournissent des informations sur l'état actuel des opérations du CPU. Par exemple, le PSW peut contenir des indicateurs de dépassement de capacité, de zéro ou de signe, ce qui aide le CPU à prendre des décisions lors de l'exécution de calculs ou de comparaisons. Ces indicateurs sont essentiels pour ajuster l'exécution en fonction des résultats intermédiaires.

Gestion de la pile (Stack Pointer) :



Memory Stack Organization

La **pile** est une structure de données qui permet de sauvegarder et de restaurer des informations essentielles durant l'exécution d'un programme, comme les adresses de retour des appels de fonction. Le **Stack Pointer** est un registre spécial qui garde la trace de l'emplacement actuel de la pile, facilitant l'ajout (push) et le retrait (pop) d'éléments de cette structure. Cela permet de conserver l'intégrité des données et de gérer les appels de fonction de manière structurée.

Unités arithmétique et logique (ALU) :

Les opérations de calcul sont confiées à l'**unité arithmétique et logique** (ALU), un composant clé du CPU. L'ALU est responsable des opérations mathématiques de base (addition, soustraction, multiplication, division) ainsi que des opérations logiques (ET, OU, NON) utilisées dans les comparaisons et les décisions. En fonction des instructions reçues, l'ALU peut traiter des calculs rapides sur des données numériques.

Connectivité avec les bus (Unité d'entrées/sorties) :

Pour interagir avec les autres composants de l'ordinateur, le CPU doit se connecter aux différents **bus** (données, adresses et contrôle). Cette connexion est assurée par l'**unité d'entrées/sorties** (I/O Unit), qui coordonne les échanges entre le CPU et les périphériques via les bus. Cette unité est cruciale pour permettre au CPU d'envoyer et de recevoir des informations vers et depuis la mémoire, les périphériques, ou d'autres parties du système.

L’unité d’adressage des instructions et le compteur de programme (PC)

L’**unité d’adressage des instructions** joue un rôle central dans le fonctionnement du CPU. C'est elle qui, via le **compteur de programme** (PC), permet au processeur de suivre le flux d’instructions à exécuter en indiquant l’adresse de chaque instruction en mémoire. Le PC est un registre interne du CPU qui stocke l’adresse de la prochaine instruction à lire et à exécuter. Ce processus commence dès l'initialisation du système et se poursuit tout au long de l'exécution d'un programme.

Le rôle du PC lors du démarrage du CPU

Lors du démarrage de l'ordinateur, après l'alimentation de la carte mère et l’envoi du signal de **RESET** au CPU, le PC est configuré pour pointer vers une adresse particulière en **ROM** (Read-Only Memory). Cette adresse contient le début du **BIOS** (Basic Input/Output System), un ensemble d’instructions stockées de manière permanente en ROM. Le BIOS est le programme de démarrage qui initialise le matériel de l’ordinateur et prépare le chargement du système d’exploitation.

Dès que le PC contient cette adresse de départ, le CPU peut lire la première instruction du BIOS. Le processus se déroule alors de manière séquentielle : après chaque instruction, le PC est mis à jour pour pointer vers l’instruction suivante en mémoire, assurant ainsi une exécution continue du BIOS jusqu'à l'initialisation complète de l'ordinateur. Le BIOS, à son tour, configure les paramètres initiaux du système, effectue un test des composants critiques (POST - Power-On Self Test), et localise le système d'exploitation pour le charger en mémoire vive (RAM).

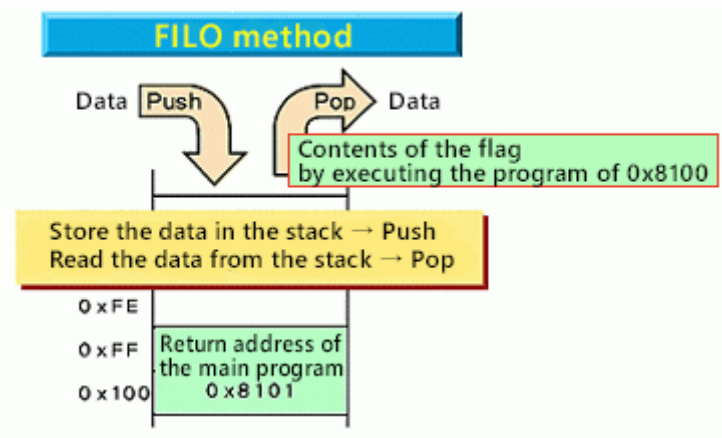
Le rôle du PC lors de l'exécution normale du programme

Après le démarrage initial, le PC continue de jouer un rôle central dans le suivi de l'exécution des instructions. Lorsqu'un programme utilisateur ou le système d'exploitation est en cours d'exécution, le PC est mis à jour pour pointer vers la prochaine instruction à chaque cycle d'horloge. En règle générale, il s'incrémente de manière constante pour passer à l’instruction suivante.

Cependant, la taille des instructions peut varier selon le type de processeur et la complexité de l'instruction. Par exemple, une instruction de base occupant 1 octet peut être incrémentée de 1, tandis qu’une instruction plus complexe,

contenant des valeurs immédiates ou des adresses spécifiques, peut occuper 2, 4, voire 8 octets. Dans ce cas, le PC est incrémenté en fonction de la longueur réelle de l'instruction, afin que le CPU passe directement à l'adresse de la prochaine instruction sans sauter ni revenir en arrière.

La gestion de la pile (STACK) et du Stack Pointer

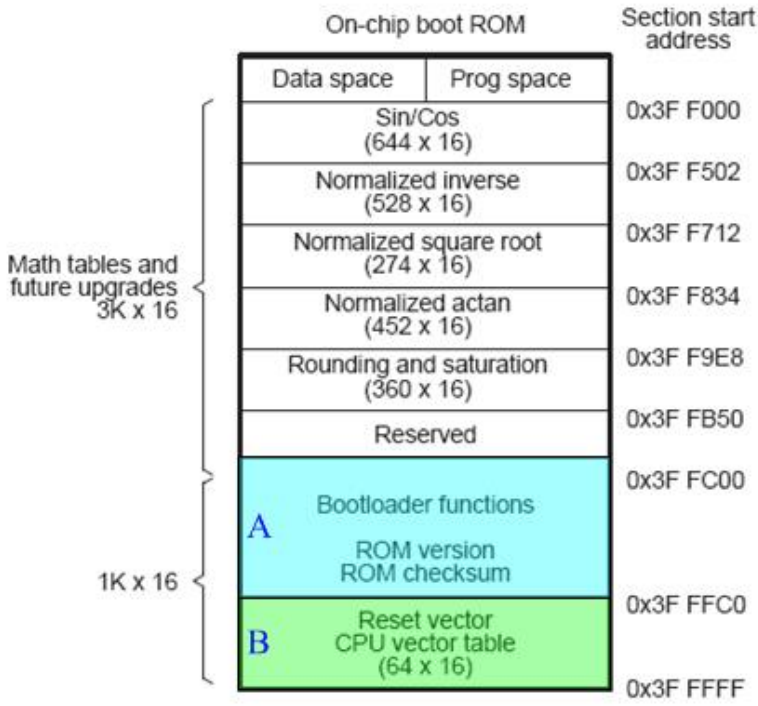


La **pile** (STACK) est une zone de la mémoire dédiée au stockage temporaire d'informations essentielles pendant l'exécution d'un programme. Elle fonctionne selon le principe **LIFO** (Last In, First Out), ce qui signifie que la dernière donnée ajoutée à la pile sera la première à en être retirée. La pile est particulièrement utile pour les appels de fonction, car elle permet de sauvegarder les adresses de retour, les paramètres de fonction et les variables locales, facilitant ainsi la reprise de l'exécution après une sous-routine ou une interruption.

Le **Stack Pointer** (SP) est un registre qui contient l'adresse de l'élément situé en haut de la pile, indiquant où les données doivent être ajoutées ou retirées. Au démarrage du CPU, le BIOS initialise le Stack Pointer pour pointer vers une adresse spécifique en mémoire, réservée à la pile. Cela garantit que la pile est prête à recevoir les données essentielles dès que le système commence à exécuter des instructions.

Lorsqu'une fonction est appelée, le CPU utilise le Stack Pointer pour sauvegarder l'adresse de retour (celle de l'instruction suivante), permettant ainsi au programme de revenir à son état initial une fois la fonction terminée. Si une interruption matérielle survient, le CPU utilise également la pile pour sauvegarder son état, garantissant une exécution fluide lors de la reprise du programme principal.

Les vecteurs d'interruption et leur rôle



Les **vecteurs d'interruption** sont des emplacements mémoire contenant les adresses des routines de gestion des interruptions, qui permettent au CPU de répondre rapidement à des événements externes inattendus. Lorsqu'une interruption matérielle survient, le CPU consulte ces vecteurs d'interruption pour déterminer quelle routine exécuter. Chaque type d'interruption a un vecteur spécifique, avec une adresse qui pointe vers le code à exécuter.

Au démarrage de l'ordinateur, le BIOS initialise également ces vecteurs d'interruption pour que les périphériques puissent signaler leurs besoins au CPU. Par exemple, un clavier, une souris ou une carte réseau peut émettre une interruption pour demander l'attention du processeur.

Les interruptions matérielles : définition et déclenchement

Une **interruption matérielle** est un signal envoyé par un périphérique externe au CPU pour indiquer qu'une action requiert son intervention. Ce signal interrompt temporairement le flux normal des instructions pour que le CPU puisse traiter cette demande urgente. Les interruptions matérielles sont souvent déclenchées par des événements externes, comme une entrée de clavier, un clic de souris, l'arrivée d'une donnée sur un port réseau, ou encore une alarme système.

Lorsqu'une interruption survient, le CPU suspend son exécution en cours, sauvegarde son état actuel (y compris le contenu du PC et des registres critiques) dans la pile, et consulte le vecteur d'interruption correspondant pour connaître l'adresse de la routine à exécuter. Une fois la routine d'interruption traitée, le CPU récupère son état depuis la pile et reprend l'exécution du programme principal exactement là où il s'était arrêté.

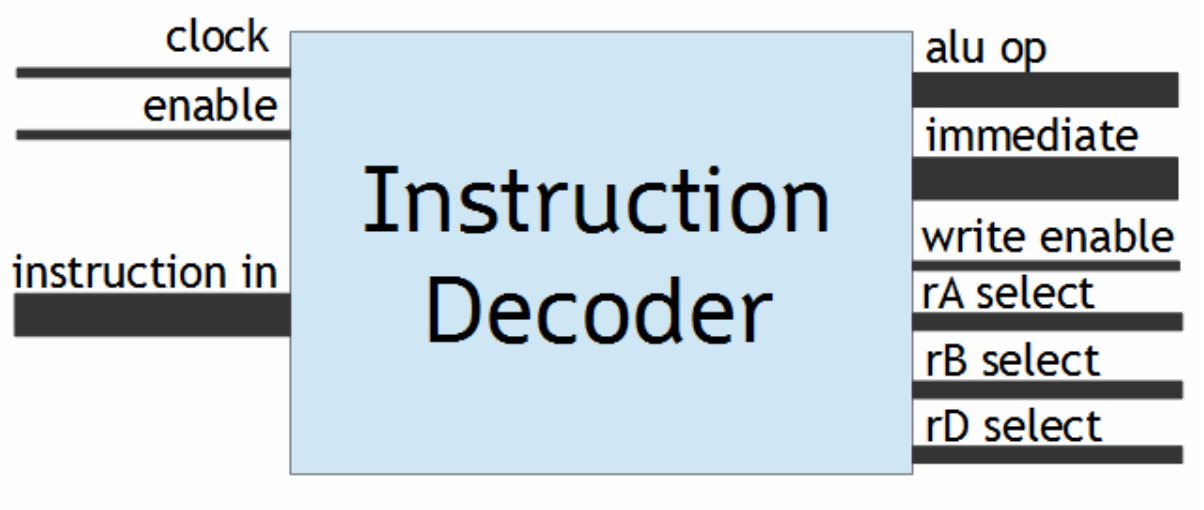
Cette gestion des interruptions permet une grande réactivité du CPU aux événements externes, améliorant ainsi l'interaction de l'ordinateur avec ses périphériques et offrant une expérience utilisateur fluide.

En résumé

L'unité d'adressage des instructions et le PC jouent un rôle crucial dans la gestion de l'exécution des instructions, en assurant un suivi précis de l'adresse de chaque instruction à lire et à exécuter. Au démarrage, le PC prend l'adresse de départ du BIOS en ROM et initialise le système, avant de poursuivre avec le programme principal. Le Stack Pointer et la pile facilitent la gestion des appels de fonctions et des interruptions, tandis que les vecteurs d'interruption permettent une réponse rapide aux événements matériels.

Ce processus coordonné entre le PC, le Stack Pointer, la pile et les vecteurs d'interruption illustre la complexité et la précision avec lesquelles le CPU gère le flux d'instructions et les interruptions, assurant ainsi le bon fonctionnement du système informatique.

L'Instruction Register (IR) : Le décodeur d'instructions du CPU



L'**Instruction Register** (IR), ou **registre d'instruction**, est l'un des composants essentiels du **CPU**. Son rôle principal est de **décoder les instructions**

récupérées en mémoire, de manière à fournir au processeur toutes les informations nécessaires pour exécuter correctement chaque instruction. En d'autres termes, l'IR interprète les commandes pour indiquer au CPU comment procéder, en identifiant quels composants doivent être sollicités et dans quel ordre. Il constitue ainsi une étape indispensable du **cycle d'instruction**.

Fonctionnement de l'Instruction Register : Décodage et validation

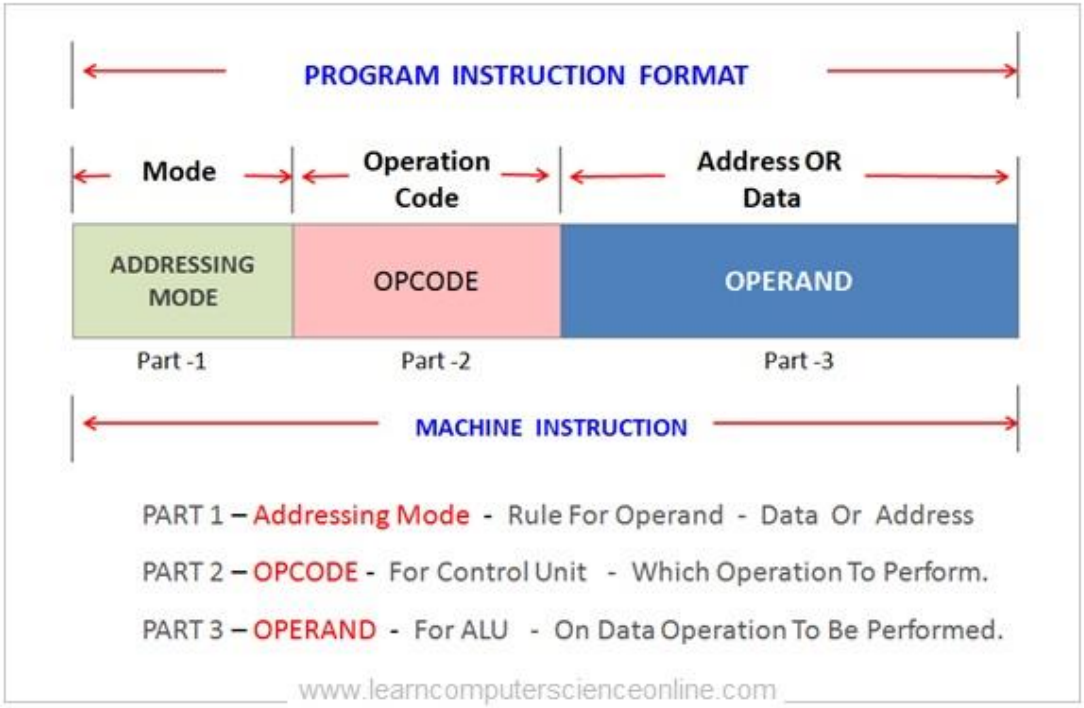
Lorsque le **compteur ordinal** (Program Counter ou PC) pointe vers une nouvelle instruction en mémoire, celle-ci est chargée dans l'IR pour être interprétée. Le rôle de l'IR est donc de **décoder l'instruction** en examinant ses différents composants et en déterminant les étapes nécessaires à son exécution. Il identifie notamment si l'instruction est complète en elle-même ou si elle nécessite des informations supplémentaires, comme des **données immédiates** ou des adresses externes.

Le processus de **décodage** implique que l'IR analyse la structure de l'instruction pour extraire son **opcode**, qui représente le code opérationnel de l'instruction. Cet opcode informe le contrôleur du CPU du type d'opération à effectuer (arithmétique, logique, saut conditionnel, etc.). De plus, l'IR est conçu pour comprendre le **format** de chaque instruction, car les instructions sont codées d'une manière spécifique, divisée en sections qui renseignent sur les différentes étapes de leur exécution.

Par exemple, l'IR peut signaler au CPU si l'instruction est prête à être exécutée immédiatement ou si d'autres données doivent être lues en mémoire. Cela inclut le mode d'adressage, qui spécifie la manière dont les données seront accessibles, ainsi que les paramètres d'entrée et de sortie de l'instruction. En définissant la **complétude de l'instruction**, l'IR informe le CPU sur la suite des étapes, et dans certains cas, déclenche une demande de données additionnelles si nécessaire.

Structure d'une instruction : Organisation et codage

Les instructions sont organisées selon un format standardisé qui facilite le décodage par l'IR. Elles sont généralement constituées de plusieurs parties clés, dont les premiers bits fournissent des informations essentielles pour le traitement de l'instruction.



1. **Bits d'adressage** :
   Les premiers bits de l'instruction indiquent le **mode d'adressage**. Le mode d'adressage détermine si les données sont directement accessibles au sein du CPU, par exemple via des **registres internes** (comme les registres à usage général ou GPRS), ou si elles nécessitent un accès à la mémoire externe. Ce système permet au CPU de s'adapter aux différentes structures de données et aux multiples configurations d'exécution.
2. **Opcode** :
   L'**opcode** est la partie de l'instruction qui spécifie le type d'opération à réaliser. Il est extrait par l'IR et interprété pour diriger le contrôleur du CPU vers le composant adéquat pour l'exécution de la tâche. Par exemple, un opcode correspondant à une opération arithmétique entraînera la sollicitation de l'**ALU** (Arithmetic Logic Unit), tandis qu'un opcode de saut conditionnel activera la **Branch Unit**.
3. **Adresses et paramètres** :
   Pour les instructions plus complexes, comme les opérations arithmétiques, l'instruction contient également des **adresses de registre** qui indiquent les

sources de données (opérandes) et la destination des résultats. Dans le cas d'une opération d'addition, par exemple, l'instruction peut spécifier deux adresses d'entrée pour les registres contenant les valeurs à additionner, ainsi qu'une adresse de sortie pour stocker le résultat.

Modes d'adressage et extraction de l'opcode

Le mode d'adressage est un élément crucial dans l'exécution des instructions. Il indique à l'IR et au CPU où se trouvent les données nécessaires et comment les manipuler. Les modes d'adressage peuvent inclure :

- **Adressage immédiat** : Les données sont incluses directement dans l'instruction, ce qui permet une exécution rapide sans accès additionnel à la mémoire.
- **Adressage direct** : L'instruction contient l'adresse mémoire exacte où se trouve la donnée, que le CPU doit alors aller chercher en mémoire.
- **Adressage indirect** : L'instruction fournit une adresse qui, à son tour, contient l'adresse de la donnée réelle.
- **Adressage par registre** : La donnée se trouve directement dans un registre interne, permettant une manipulation rapide sans recours à la mémoire centrale.

L'IR doit extraire l'opcode pour déterminer précisément quel composant doit être sollicité dans le CPU. Cette extraction permet de diriger les étapes de traitement vers l'unité appropriée, comme la **Load and Store Unit** pour les transferts de données entre le CPU et la mémoire, ou l'ALU pour les calculs mathématiques et logiques. Grâce à l'opcode, l'IR transforme l'instruction brute en une série d'actions concrètes exécutables par le CPU.

Exécution des instructions arithmétiques : Un exemple

Prenons l'exemple d'une **instruction arithmétique**, comme une addition. L'IR décode l'instruction pour identifier le type d'opération (addition) grâce à l'opcode. Ensuite, il lit les adresses des registres impliqués : deux adresses d'entrée pour les deux valeurs à additionner, et une adresse de sortie pour enregistrer le résultat. Si l'adressage est interne (par exemple, si les valeurs se trouvent dans des registres internes), l'opération se déroule directement dans le CPU, ce qui réduit le temps d'exécution. L'ALU est ensuite sollicitée pour réaliser l'addition, et le résultat est stocké dans le registre de sortie spécifié.

Rôle de l'Instruction Register dans le contrôle des opérations

L'IR joue également un rôle de **contrôle** en surveillant la complétude de l'instruction. Il identifie si une instruction est immédiatement exécutable ou si elle nécessite des étapes additionnelles, comme la lecture de données supplémentaires. Par exemple, une instruction de chargement (Load) pourrait demander d'aller chercher une donnée en mémoire avant de la traiter, tandis qu'une instruction de saut conditionnel doit vérifier une condition (ex. une comparaison entre deux valeurs) avant d'exécuter le saut.

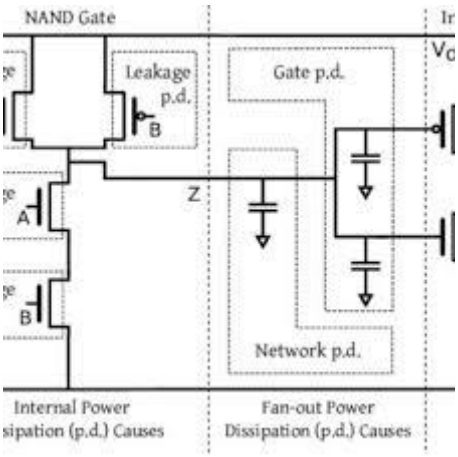
Dans le cas de **jumps conditionnels** (sauts conditionnels), l'IR vérifie les bits d'état (stockés dans le Process Status Word ou PSW) avant d'indiquer au CPU de changer de flux d'exécution. Cette gestion est cruciale pour permettre l'exécution de programmes complexes avec des branches et des boucles.

En résumer

L'Instruction Register (IR) est fondamental dans la gestion des instructions. En décomposant les instructions, en extrayant l'opcode, en vérifiant les modes d'adressage et en identifiant les étapes nécessaires, l'IR assure que chaque instruction est correctement exécutée. Ce composant du CPU joue un rôle de décodeur intelligent, transformant des codes bruts en actions spécifiques, synchronisant l'usage des autres unités du CPU et assurant un contrôle optimal sur le déroulement des opérations.

En somme, l'IR assure que le CPU interprète et exécute chaque instruction avec précision, en dirigeant le flux d'exécution selon le programme en cours. Sa capacité à décoder les instructions et à guider le CPU en fonction des opcodes fait de l'IR une pièce maîtresse dans la performance et la flexibilité des systèmes informatiques.

La Load and Store Unit (LSU)



La **Load and Store Unit** (LSU) est un composant central dans l'architecture du CPU, responsable des opérations de **chargement (Load)** et de **stockage (Store)** des données entre les registres internes du CPU et la mémoire centrale. Cette unité permet au processeur de récupérer les données nécessaires depuis la mémoire pour les traiter, puis de stocker les résultats une fois les calculs terminés.

Dans un programme, chaque opération de lecture ou d'écriture en mémoire implique une série de cycles d'horloge qui permettent d'établir la communication entre le CPU et la mémoire. Ce processus est plus coûteux en temps que les opérations effectuées directement dans les registres internes du CPU, en raison du besoin de coordonner les différents bus de données, d'adresses et de contrôle.

Les étapes d'une opération Load ou Store

Le processus d'une opération de chargement ou de stockage suit plusieurs étapes clés, nécessitant chacune un **cycle d'horloge** :

1. **Établissement de l'adresse sur le bus** : L'opération commence par la mise en place de l'**adresse mémoire** sur le bus d'adresses. Cette adresse indique où, dans la mémoire centrale, se trouvent les données à lire ou l'emplacement où écrire les données. Une fois l'adresse établie, le CPU doit attendre la stabilisation du signal sur le bus d'adresses.
2. **Validation de l'opération via le bus de contrôle** : Après avoir spécifié l'adresse, le CPU utilise le **bus de contrôle** pour valider l'opération de lecture ou d'écriture. Le bus de contrôle envoie un signal pour indiquer si l'opération doit être une lecture (Load) ou une écriture (Store). Ce signal permet à la mémoire de se préparer à envoyer ou recevoir les données.

3. **Lecture ou écriture des données sur le bus de données** : Une fois que l'opération est validée, les données sont transférées via le **bus de données**. Dans le cas d'une opération de lecture, la mémoire place les données demandées sur le bus de données pour que le CPU puisse les récupérer et les charger dans un registre interne. En écriture, le CPU place les données sur le bus de données pour qu'elles soient enregistrées à l'adresse mémoire spécifiée.
4. **Retour à l'adresse de l'instruction suivante** : Une fois l'opération de lecture ou d'écriture terminée, le CPU met à jour l'adresse dans le compteur ordinal (Program Counter) pour pointer vers l'instruction suivante, permettant ainsi au programme de continuer son exécution.

Chaque étape nécessite au moins un cycle d'horloge, et certaines opérations en mémoire peuvent prendre plusieurs cycles en fonction de la taille des données à lire ou écrire. Par exemple, une opération sur un mot de 32 bits peut nécessiter un cycle supplémentaire par rapport à une opération sur un octet de 8 bits, car la mémoire peut être structurée en segments de différentes tailles.

La pénalité d'accès mémoire et l'avantage des registres internes

Une opération de lecture ou d'écriture en mémoire est coûteuse en cycles d'horloge par rapport à une opération effectuée directement dans un registre interne du CPU. Lorsque le CPU doit accéder à des données dans les registres internes, les données sont disponibles instantanément, sans besoin de coordonner l'accès par les bus de données, d'adresses ou de contrôle. En revanche, un accès mémoire implique de multiples cycles d'horloge pour transférer l'adresse, valider l'opération, et récupérer ou envoyer les données.

Cette différence de vitesse entre l'accès aux registres internes et la mémoire centrale est souvent appelée **pénalité d'accès mémoire**. En effet, les opérations en mémoire prennent plus de temps et peuvent ralentir le déroulement global du programme, en particulier dans les programmes qui nécessitent un grand nombre de lectures ou d'écritures en mémoire.

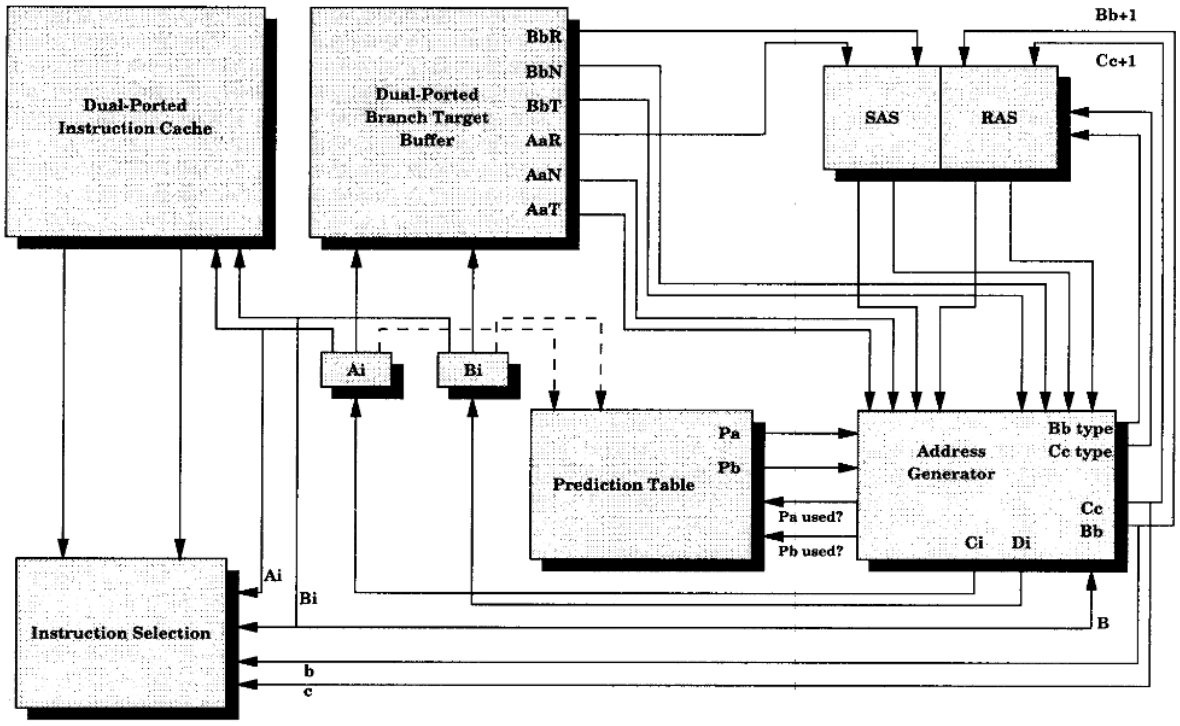
Pour atténuer cette pénalité, les processeurs modernes utilisent des **caches** – de petites mémoires rapides situées près du CPU – qui stockent temporairement les données les plus fréquemment utilisées, réduisant ainsi le nombre de fois où le CPU doit accéder à la mémoire centrale. Cependant, même avec des caches, certaines données doivent encore être chargées ou stockées directement en mémoire centrale, et la LSU reste cruciale pour gérer ces opérations.

Importance de la Load and Store Unit dans l'optimisation des performances

La **Load and Store Unit** joue un rôle clé dans l'optimisation des performances du CPU. En gérant efficacement les transferts de données entre les registres et la mémoire, la LSU permet au CPU de se concentrer sur les calculs et les traitements sans être entravé par les délais d'accès mémoire.

Les architectes des processeurs travaillent constamment à améliorer la vitesse des opérations de chargement et de stockage, en minimisant le nombre de cycles d'horloge nécessaires pour chaque accès mémoire. Par exemple, certaines unités LSU modernes incluent des mécanismes de prédiction d'accès pour anticiper les besoins de données du CPU, ce qui permet de réduire les délais.

Le Branch Unit (BU) : Gestion des sauts et optimisation du flux d'instructions



(cela peut vite devenir compliqué ...)

La **Branch Unit** (BU) est un composant essentiel du CPU, responsable de la gestion des **sauts** au sein des programmes. Les sauts sont des instructions qui modifient l'ordre séquentiel de l'exécution d'un programme, permettant au CPU de se déplacer vers une autre section de code. Ces sauts peuvent être **conditionnels** (basés sur une condition spécifique) ou **inconditionnels** (sans condition).

Les sauts sont essentiels pour le fonctionnement des **boucles**, des **instructions conditionnelles** (comme les "si... alors"), et des **appels de fonctions**. Grâce à la Branch Unit, le CPU peut naviguer efficacement dans le programme et exécuter des instructions dans un ordre qui dépend des résultats des calculs précédents.

Les sauts conditionnels et inconditionnels

Un saut **inconditionnel** est une instruction simple qui déplace immédiatement le point d'exécution vers une adresse spécifique, sans vérifier de condition. Un exemple courant est une boucle infinie, où l'instruction de saut renvoie continuellement le CPU à la même adresse mémoire.

Les **sauts conditionnels**, en revanche, sont effectués uniquement si une certaine condition est remplie, comme le résultat d'une comparaison (ex. « si x est égal à 0, alors sauter à l'adresse Y »). La BU vérifie les **bits d'état** (présents dans le Process Status Word ou PSW) pour déterminer si la condition est vérifiée. Si c'est le cas, le saut est exécuté ; sinon, le programme continue son exécution séquentielle.

Les instructions CALL et RET : Gestion des appels de fonction

En plus des sauts conditionnels et inconditionnels, la Branch Unit gère les instructions **CALL** et **RET**. Ces instructions sont utilisées pour les **appels de fonction**. L'instruction **CALL** permet au programme de sauter vers l'adresse d'une fonction, tout en sauvegardant l'adresse de l'instruction suivante dans la pile, de sorte que le CPU puisse y revenir après l'exécution de la fonction.

Lorsque la fonction est terminée, l'instruction **RET** (retour) est exécutée, permettant au CPU de récupérer l'adresse sauvegardée dans la pile et de reprendre l'exécution du programme principal là où il l'avait laissé. La combinaison des instructions CALL et RET permet au CPU d'exécuter des sous-routines ou fonctions et d'y revenir automatiquement, ce qui est essentiel pour la structure des programmes modernes.

La prédiction de saut : Amélioration des performances

Dans les **processeurs modernes**, la Branch Unit est dotée de mécanismes de **prédiction de saut** pour améliorer les performances du CPU. Lorsqu'une instruction de saut conditionnel est rencontrée, il peut y avoir un délai si le CPU doit attendre le résultat de la condition avant de savoir quelle instruction exécuter ensuite. La prédiction de saut aide à minimiser ce délai.

Les prédicteurs de saut fonctionnent en analysant le comportement antérieur des sauts pour **prédire** l'adresse de destination. Par exemple, dans une boucle, si le saut a été pris plusieurs fois de suite, la BU peut prédire que le saut sera à nouveau pris, anticipant ainsi l'adresse de la prochaine instruction. Si la prédiction est correcte, le CPU peut exécuter les instructions sans interruption, ce qui accélère le traitement.

Bien sûr, la prédiction de saut n'est pas infaillible. Si la BU fait une prédiction incorrecte, le CPU doit annuler les instructions préchargées et revenir à l'adresse correcte, ce qui entraîne une perte de temps appelée **mauvaise prédiction**. Malgré ces erreurs occasionnelles, la prédiction de saut améliore généralement les performances des CPU modernes en optimisant le flux d'instructions.

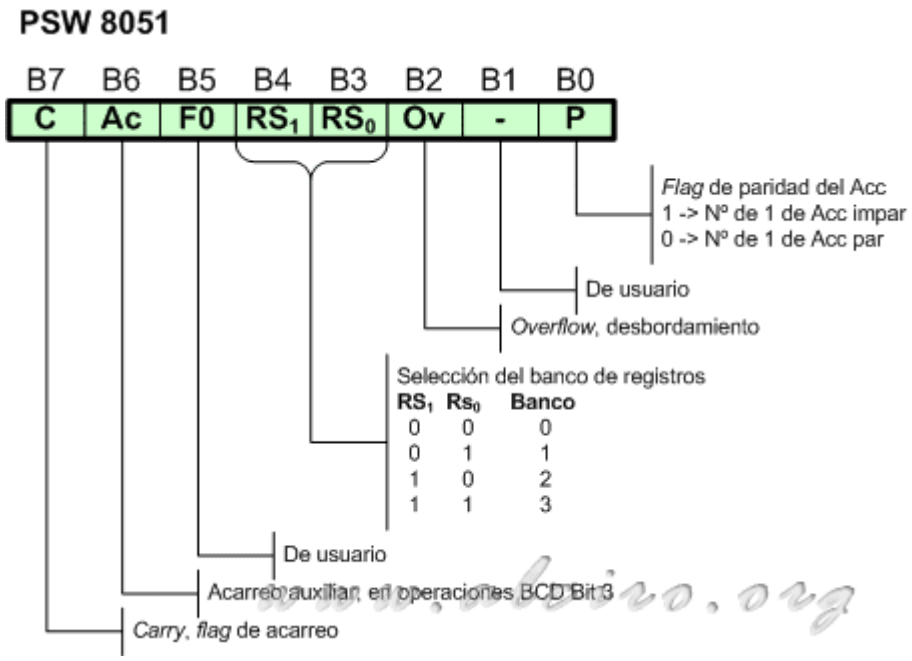
Transmission du résultat au Program Counter (PC)

Une fois la décision de saut prise, que ce soit pour un saut conditionnel, un appel de fonction (CALL) ou un retour (RET), la **Branch Unit** transmet l'adresse de destination au **Program Counter** (PC). Le PC est alors mis à jour avec cette nouvelle adresse, indiquant au CPU où aller pour lire la prochaine instruction.

En résumé, la Branch Unit est indispensable pour la gestion des sauts et des flux de contrôle dans les programmes, en ajustant le PC pour que le CPU puisse suivre le flux d'instructions correct. Grâce à ses mécanismes de prédiction de saut, la BU optimise également les performances en anticipant les adresses de destination et en minimisant les retards.

Le Process Status Word (PSW) : Indicateurs d'état et leur rôle dans l'exécution des programmes

Le **Process Status Word** (PSW), souvent appelé **registre d'état**, est un composant essentiel du CPU. Il contient une série de **flags** ou **indicateurs d'état** qui reflètent des informations critiques sur les opérations effectuées par le processeur. Ces indicateurs servent à suivre les résultats des calculs et des comparaisons, et influencent la prise de décision au sein du CPU, en particulier pour les instructions de **sauts conditionnels**.



Structure et composition du PSW

Le PSW est composé de plusieurs flags, qui sont chacun représentés par un **bit**. Chaque flag est indépendant des autres, ce qui signifie que les informations qu'ils contiennent ne sont pas interconnectées. Les valeurs des flags sont déterminées par les opérations en cours, principalement celles réalisées par l'**unité arithmétique et logique** (ALU). En fonction des résultats produits par l'ALU, ces flags sont mis à jour pour refléter l'état actuel du système ou le résultat des calculs.

Les flags présents dans le PSW peuvent varier selon l'architecture du processeur, mais on retrouve souvent les mêmes indicateurs fondamentaux dans la plupart des CPU. Les flags les plus courants incluent le **Carry**, **Overflow**, **Zero**, et **Negative**. Ces indicateurs fournissent des informations précises sur les

résultats des opérations arithmétiques et logiques, et sont essentiels pour le bon fonctionnement des programmes.

Les principaux flags du PSW

Voici une description des principaux flags présents dans le PSW :

1. **Carry Flag (C)** : Le Carry Flag est utilisé pour indiquer si une opération arithmétique a généré un **report** (carry) ou un **emprunt**. Par exemple, dans une opération d'addition, si la somme des bits dépasse la capacité d'un registre (par exemple, au-delà de 255 pour un registre de 8 bits), un bit de report est généré, et le Carry Flag est mis à 1 pour signaler ce dépassement. Ce flag est particulièrement utile pour les opérations en arithmétique binaire multi-octets.
2. **Overflow Flag (O)** : L'Overflow Flag indique si une opération arithmétique a dépassé la capacité du type de données signé, c'est-à-dire si le résultat est trop grand pour être représenté avec le nombre de bits disponibles. Par exemple, dans une addition de nombres signés, un débordement peut se produire si la somme dépasse la limite positive ou négative de la représentation. Ce flag est crucial pour éviter des erreurs lorsque le programme utilise des entiers signés.
3. **Zero Flag (Z)** : Le Zero Flag est utilisé pour indiquer si le résultat d'une opération est égal à zéro. Après chaque opération arithmétique ou logique, l'ALU vérifie si le résultat est nul ; si c'est le cas, le Zero Flag est mis à 1. Ce flag est souvent utilisé dans les **sauts conditionnels**, car il permet au CPU de réagir différemment en fonction de l'obtention d'un résultat nul.
4. **Negative Flag (N)** : Le Negative Flag est utilisé pour indiquer si le résultat d'une opération est négatif. Ce flag est mis à 1 si le bit de signe (le bit le plus significatif) du résultat est égal à 1, ce qui signifie que le résultat est négatif dans les nombres signés. Ce flag est particulièrement utile pour les instructions de comparaison et les sauts conditionnels basés sur des valeurs positives ou négatives.

Source des informations dans le PSW

Les informations présentes dans le PSW proviennent principalement des opérations effectuées par l'**ALU**. Chaque fois qu'une opération arithmétique ou logique est exécutée, l'ALU met à jour les flags correspondants dans le PSW en fonction du résultat obtenu. Par exemple, après une addition, l'ALU vérifie si un report a été généré et, si c'est le cas, elle met le Carry Flag à 1. Si l'addition

produit un résultat nul, le Zero Flag est mis à 1, tandis que le Negative Flag peut être mis à 1 si le résultat est négatif.

Ces informations sont ensuite stockées dans le PSW, où elles sont accessibles pour les autres composants du CPU, en particulier la **Branch Unit** (BU). La Branch Unit utilise ces flags pour décider de la suite des opérations, notamment lors des sauts conditionnels.

Utilisation des flags pour les sauts conditionnels

Les **sauts conditionnels** permettent au CPU de prendre des décisions basées sur les résultats de calculs précédents. En examinant les flags du PSW, le CPU peut décider de sauter à une nouvelle adresse d'instruction ou de continuer son exécution de manière séquentielle.

Par exemple :

- Si une instruction de saut est conditionnée sur le fait que le résultat soit nul, le CPU vérifie le **Zero Flag**. Si le flag est à 1 (ce qui indique que le résultat de la dernière opération était zéro), le saut est exécuté.
- Pour une comparaison de grandeur, comme un saut si le résultat est négatif, le **Negative Flag** sera utilisé. Si ce flag est activé, cela signifie que le résultat de la dernière opération était un nombre négatif, et le saut sera effectué en conséquence.

Les sauts conditionnels permettent de créer des structures de contrôle dans les programmes, comme les boucles et les conditions, qui adaptent l'exécution en fonction des calculs réalisés.

Importance du PSW dans le contrôle de flux du CPU

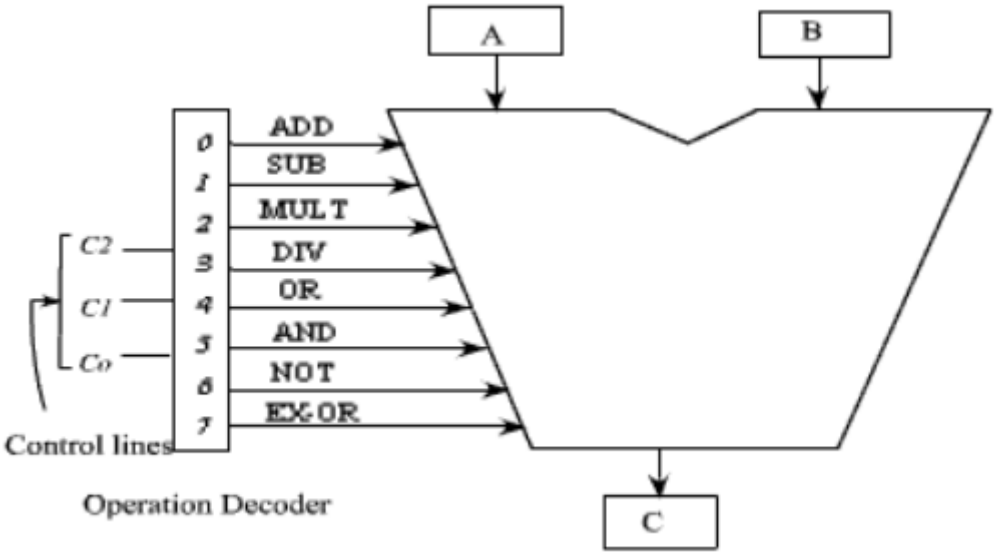
Le PSW est crucial pour le contrôle de flux dans le CPU. Les informations qu'il contient influencent directement les décisions d'exécution prises par le processeur. En utilisant les flags, le CPU peut réagir de manière dynamique aux résultats des opérations, ajustant ainsi son exécution pour répondre aux besoins du programme.

Ce registre est fondamental pour les architectures modernes de CPU, car il facilite le traitement conditionnel et optimise la prise de décision dans les programmes. Le PSW joue un rôle de lien entre l'ALU et la Branch Unit, garantissant que les résultats de chaque opération influencent correctement le déroulement des instructions.

Les Unités Arithmétiques et Logiques (ALU) : Fonctionnement et Rôle dans le CPU

Les **unités arithmétiques et logiques** (ALU) sont des composants centraux du CPU, responsables de réaliser toutes les opérations mathématiques et logiques nécessaires à l'exécution des programmes. Ces unités sont spécifiquement sollicitées pour les calculs arithmétiques, comme les additions ou les multiplications, ainsi que pour les opérations logiques, comme les "AND" et "OR". Le rôle de l'ALU est crucial, car elle permet au processeur de traiter des données et de prendre des décisions basées sur les résultats des calculs.

Multiplicité des ALU dans certains processeurs



Dans certains processeurs, il peut y avoir **plusieurs ALU** pour exécuter différents types de calculs. Par exemple, on trouve souvent une ALU dédiée aux calculs entiers, tandis qu'une autre ALU est spécialisée dans les calculs en **virgule flottante**. Les opérations en virgule flottante, nécessaires pour les calculs scientifiques ou graphiques, requièrent une gestion spécifique des nombres à virgule, ce qui peut rendre le traitement plus complexe. En intégrant plusieurs ALU, les processeurs modernes peuvent effectuer plusieurs opérations simultanément, améliorant ainsi les performances globales.

Chaque ALU est connectée, en entrée et en sortie, aux **registres** du CPU. Les registres sont de petites mémoires très rapides qui stockent temporairement les données en cours de traitement. Les seules données accessibles aux ALU proviennent donc des registres internes ou, dans certains cas, de l'**Instruction**

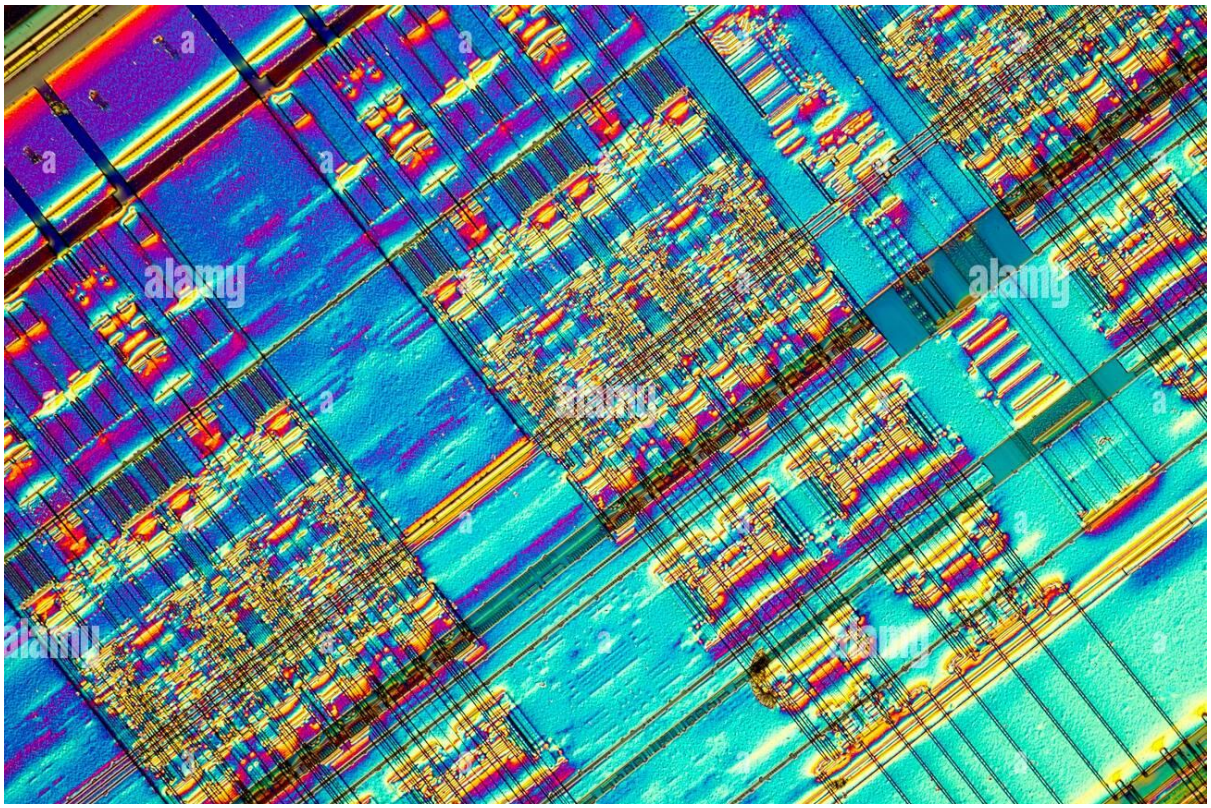
Register (IR) lorsqu'une instruction fournit directement une valeur immédiate (donnée incluse dans l'instruction elle-même).

Fonctionnement des ALU : Entrée et Sortie des Données

Lorsqu'une opération nécessite l'intervention de l'ALU, celle-ci reçoit ses **données en entrée** depuis les registres. Ces données peuvent être de plusieurs types : soit des valeurs stockées en mémoire et chargées au préalable dans les registres, soit des **données immédiates** fournies directement par l'instruction. Dans ce dernier cas, c'est l'Instruction Register qui transmet ces données immédiates à l'ALU lors du décodage de l'instruction.

Une fois l'opération effectuée, le **résultat produit par l'ALU** est ensuite transmis à un registre de sortie, où il est conservé pour d'autres traitements ou pour être transféré en mémoire. Les registres jouent donc un rôle clé dans la rapidité de traitement, car ils permettent à l'ALU d'accéder aux données sans nécessiter de chargement ou de stockage dans la mémoire principale, ce qui prendrait plus de temps.

Types d'Opérations et Variabilité des Instructions



Les ALU sont conçues pour réaliser des **opérations arithmétiques et logiques**. Parmi les opérations arithmétiques, on retrouve l'addition, la soustraction, la

multiplication et, dans certains cas, la division (bien que cette dernière soit parfois réalisée par une unité spécialisée). Pour les opérations logiques, l'ALU peut effectuer des opérations telles que "AND", "OR", "NOT" et "XOR" (pour eXclusive OR).

Certaines opérations, comme l'opération "NOT", ne nécessitent qu'une seule **opérande en entrée**. Le "NOT" inverse tous les bits d'un mot, transformant les 1 en 0 et les 0 en 1. Ce type d'opération est couramment utilisé dans la manipulation de données binaires et les traitements logiques.

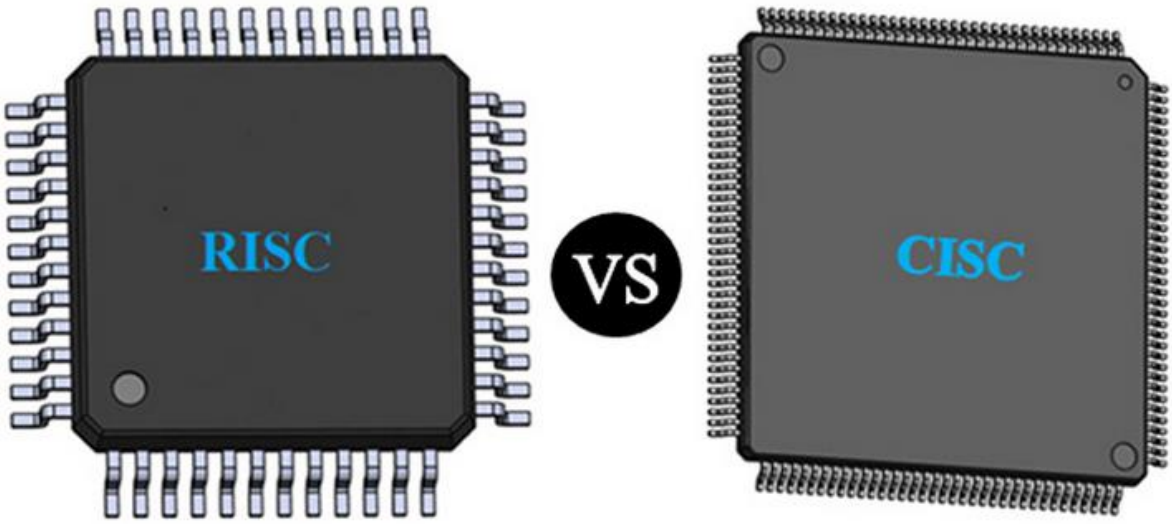
Les ALU peuvent traiter des entiers de différentes tailles, selon l'instruction et l'architecture du processeur. Par exemple, elles peuvent manipuler des valeurs de 8 bits, 16 bits, 32 bits ou 64 bits. Les processeurs modernes sont généralement conçus pour fonctionner avec des mots de 32 ou 64 bits, ce qui permet un traitement rapide de grandes quantités de données. Cependant, pour des besoins spécifiques, certaines instructions peuvent limiter la taille des données traitées, ce qui donne de la flexibilité au CPU en fonction des opérations à exécuter.

Rôle de l'Instruction Register (IR) dans le Contrôle des ALU

Le **choix de l'opération exécutée par l'ALU** est contrôlé par l'**Instruction Register** (IR), qui extrait l'**opcode** de chaque instruction lors du décodage. L'opcode spécifie le type d'opération que l'ALU doit effectuer (par exemple, addition, soustraction, OU logique). En fonction de cet opcode, l'IR oriente le CPU vers l'ALU appropriée (dans le cas de multiples ALU) et active l'opération demandée.

Ce processus de décodage et de sélection permet au CPU de traiter efficacement les instructions, en orientant chaque opération vers l'unité qui peut la gérer le plus rapidement possible. Par exemple, une instruction arithmétique simple peut être envoyée vers l'ALU dédiée aux entiers, tandis qu'une instruction de calcul en virgule flottante sera traitée par l'ALU dédiée aux opérations en virgule flottante.

Variabilité des Instructions : CISC vs RISC



Le **nombre d'instructions** qu'un CPU peut traiter dépend en grande partie de son **architecture**. Il existe principalement deux types d'architectures : **CISC** (Complex Instruction Set Computing) et **RISC** (Reduced Instruction Set Computing).

- Dans une architecture **CISC**, comme celles utilisées dans les processeurs Intel x86, le CPU est conçu pour gérer un grand nombre d'instructions complexes, souvent capables d'effectuer plusieurs opérations en une seule instruction. Les CISC permettent de réduire le nombre total d'instructions nécessaires pour accomplir une tâche, mais chaque instruction peut être plus longue à exécuter.
- Dans une architecture **RISC**, au contraire, le CPU dispose d'un ensemble d'instructions plus limité, mais chaque instruction est conçue pour être exécutée en un seul cycle d'horloge, si possible. Les processeurs RISC, comme ceux basés sur ARM, sont souvent plus rapides pour exécuter des instructions simples, car chaque opération est divisée en étapes atomiques. Cela favorise la vitesse, mais peut nécessiter plus d'instructions pour accomplir des tâches complexes.

Impact des Cycles d'Horloge sur les Performances du CPU

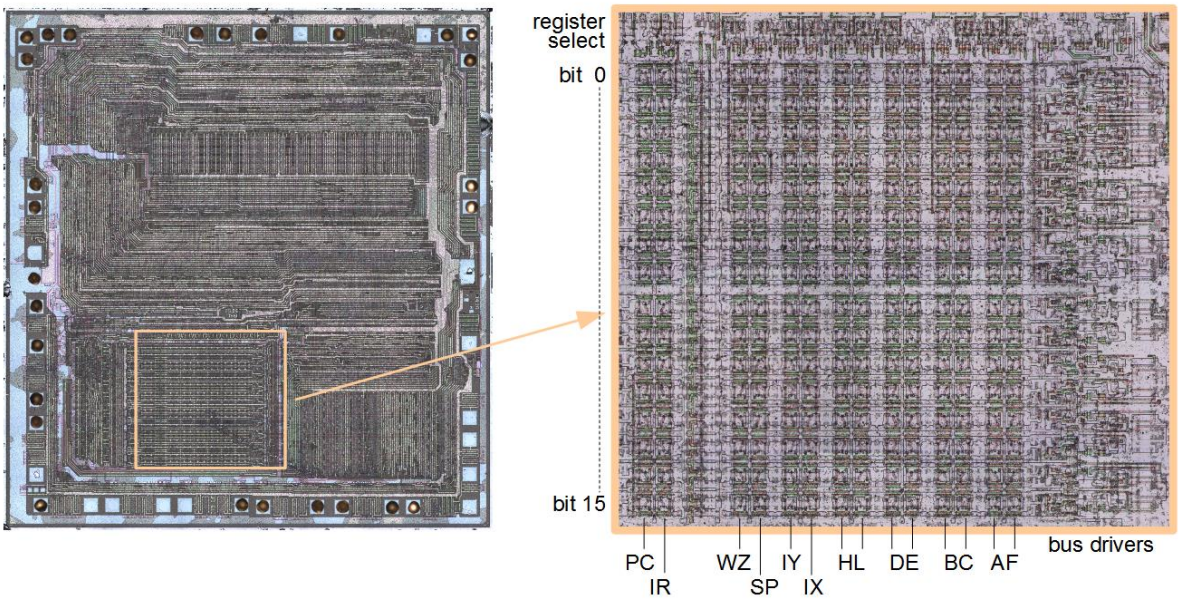
L'exécution des instructions par l'ALU prend un certain nombre de **cycles d'horloge**, selon la complexité de l'opération et la taille des données traitées. Par exemple, une addition simple peut être réalisée en un cycle, tandis qu'une multiplication ou une division peut nécessiter plusieurs cycles. Les opérations

en virgule flottante, qui sont plus complexes, prennent généralement plus de temps que les opérations sur des entiers.

L'**impact de la durée des cycles** devient particulièrement significatif dans les processeurs CISC, où certaines instructions peuvent prendre beaucoup de cycles pour être exécutées, contrairement aux processeurs RISC qui privilégient la rapidité d'exécution par cycle unique.

Cette variabilité des temps d'exécution a un effet direct sur les performances du CPU. Les instructions nécessitant plusieurs cycles retardent l'exécution des instructions suivantes, surtout si elles sont essentielles au programme en cours. Pour atténuer ces ralentissements, les architectures modernes de CPU utilisent des techniques comme le **pipeline**, où plusieurs instructions sont traitées simultanément dans des phases différentes (par exemple, une instruction en décodage pendant qu'une autre est en exécution dans l'ALU).

Les GPRs (General Purpose Registers) : Les registres à usage général



Les **General Purpose Registers** (GPRs), ou registres à usage général, jouent un rôle fondamental dans le CPU, en servant de zones de stockage temporaire pour les données manipulées au cours des traitements. Utilisés pour diverses opérations (chargement, calculs, transferts), les GPRs permettent au CPU d'accéder rapidement aux données sans recourir constamment à la mémoire principale, ce qui accélère le traitement.

Les **données contenues dans les GPRs** sont directement accessibles aux unités de calcul du CPU, notamment les ALU, qui s'appuient sur ces registres pour effectuer des opérations arithmétiques et logiques. La quantité de GPRs

varie en fonction de l'architecture et du modèle du CPU. Par exemple, les processeurs modernes à architecture **x86** disposent généralement de huit registres à usage général, tandis que les architectures **ARM** peuvent en avoir jusqu'à 16 ou 32, selon le modèle.

Chaque GPR peut être **scindé** pour traiter des données de tailles différentes, selon les besoins du programme. Par exemple, un registre de 64 bits peut être divisé en deux segments de 32 bits ou en quatre segments de 16 bits, permettant ainsi une flexibilité accrue dans le traitement des données. Cette modularité est essentielle pour les applications qui manipulent des données de formats variés.

La **taille des GPRs** détermine également le **Data Range** disponible dans les applications, c'est-à-dire l'étendue des valeurs que le CPU peut traiter directement. Par exemple, un GPR de 32 bits permet de manipuler des valeurs allant de 0 à 4 294 967 295 (en mode non signé), tandis qu'un registre de 64 bits étend cette capacité de calcul. En somme, les GPRs optimisent la réactivité du CPU en minimisant les accès à la mémoire principale.

Interfaces d'I/O sur les bus : Connectivité et gestion des signaux

Les **interfaces d'entrée/sortie (I/O)** sur les bus jouent un rôle crucial dans la connectivité du CPU avec les autres composants de l'ordinateur. Leur fonction principale est d'assurer une **liaison électrique fiable** entre les éléments internes du CPU et les composants externes, permettant le transfert fluide de données et de commandes. Ces interfaces sont essentielles au fonctionnement coordonné du système, car elles facilitent les communications entre le CPU, la mémoire, et les périphériques via les différents types de bus.

En fonction du type de bus – **bus d'adresse, bus de données, ou bus de contrôle** – les interfaces d'I/O peuvent gérer des signaux unidirectionnels ou bidirectionnels :

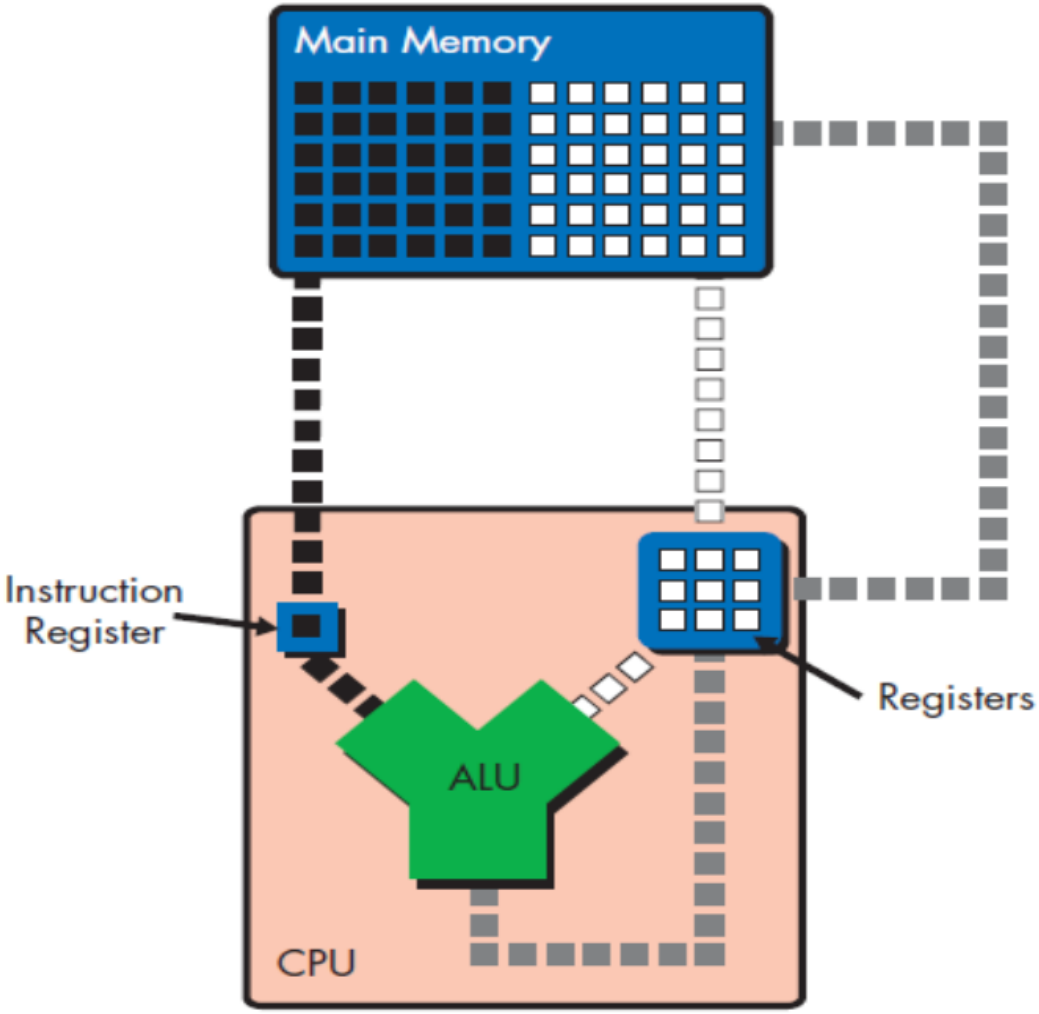
- **Bus d'adresse** : Ce bus est **unidirectionnel**. Le CPU envoie les adresses vers la mémoire ou les périphériques pour indiquer l'emplacement où lire ou écrire des données. Les signaux ne circulent que du CPU vers les autres composants.
- **Bus de données** : Ce bus est généralement **bidirectionnel**. Il permet aux données de circuler entre le CPU, la mémoire et les périphériques, dans les deux sens, selon qu'il s'agisse d'une opération de lecture ou d'écriture.
- **Bus de contrôle** : Ce bus gère divers signaux de commande pour synchroniser les opérations. Les signaux peuvent être unidirectionnels ou

bidirectionnels, en fonction de l'opération. Par exemple, le CPU peut envoyer un signal de lecture ou d’écriture, et des périphériques peuvent renvoyer un signal d’interruption.

En organisant les échanges selon la direction et le type de signal, les interfaces d'I/O garantissent que les communications se déroulent sans conflit, améliorant ainsi l'efficacité du système et la rapidité des opérations du CPU.

Fonctionnement du CPU

Dans ce chapitre, nous décrirons en détail le **déroulement de l'exécution d'un programme** dans un CPU, en expliquant les **flux de données** et les **flux d'instructions** entre le processeur et la mémoire centrale. Pour faciliter la compréhension de ces échanges, nous utiliserons une illustration représentant la mémoire centrale et les flux de données vers et depuis le CPU. Cette figure met en lumière l'architecture **Von Neumann**, où les instructions et les données partagent les mêmes connexions pour leur transit vers le processeur.



Représentation de la mémoire centrale : Instructions et données

Dans notre illustration, la mémoire centrale est représentée par un grand rectangle divisé en plusieurs espaces mémoire. Ces espaces sont divisés en deux sections : l'une pour les **instructions** et l'autre pour les **données**. Les instructions sont symbolisées par des **carrés noirs** tandis que les données sont représentées par des **carrés blancs**.

Bien que notre illustration sépare visuellement les instructions et les données pour des raisons pédagogiques, il est important de noter que, dans une architecture de type Von Neumann, ces éléments **ne sont pas physiquement séparés** dans la mémoire. Les instructions et les données partagent un même espace mémoire, ce qui signifie que le CPU peut accéder aux deux types d'informations via les mêmes connexions (ou "fils") vers la mémoire centrale.

Les trois flux de données

À partir de la mémoire centrale, trois flux d'informations sont représentés :

1. **Flux des instructions** : Les instructions (carrés noirs) sont envoyées depuis la mémoire centrale vers le CPU, où elles seront décodées et exécutées. Ce flux symbolise le parcours des instructions stockées en mémoire pour être traitées par le processeur.
2. **Flux des données entrantes** : Ce flux, représenté par des **carrés blancs**, transporte les données nécessaires pour les calculs et les traitements dans le CPU. Ces données peuvent provenir de différentes parties de la mémoire centrale et sont dirigées vers le CPU pour un traitement immédiat ou pour être chargées dans les registres du CPU.
3. **Flux des données sortantes** : Ce flux est illustré par des **carrés gris** et représente les données que le CPU renvoie vers la mémoire centrale. Ces données peuvent être le résultat d'un calcul, une valeur mise à jour ou toute autre information que le CPU souhaite sauvegarder en mémoire.

Ces trois flux convergent vers un rectangle couleur **saumon** qui représente le CPU dans notre illustration.

Structure interne du CPU : Instruction Register, GPRs et ALU

À l'intérieur du CPU (carré saumon), plusieurs composants sont visibles, chacun jouant un rôle spécifique dans le traitement des instructions et des données :

- **Instruction Register (IR)** : Le premier composant représenté dans le CPU est l'**Instruction Register** (IR), figuré par un rectangle **bleu** avec un carré **noir** à l'intérieur. Le rôle de l'IR est de **décoder** les instructions entrantes (carrés noirs) pour en extraire l'**opcode** (le code opérationnel de l'instruction) et les informations nécessaires à son exécution. Une fois l'opcode extrait, il est transmis à l'ALU, symbolisé par un flux continu de carrés noirs en direction de l'ALU.
- **Base des registres GPRs** : Un autre composant, également représenté par un rectangle **bleu**, contient des **carrés blancs** à l'intérieur pour symboliser

les **General Purpose Registers (GPRs)**. Ces registres sont des zones de stockage temporaire pour les données que le CPU utilise au cours de l'exécution des instructions. Les registres offrent un accès rapide aux données, permettant aux unités de traitement, comme l'ALU, d'exécuter leurs opérations sans avoir besoin d'accéder en permanence à la mémoire centrale.

- **ALU (Arithmetic Logic Unit)** : La **Arithmetic Logic Unit** (ALU) est représentée par une forme en **Y** vert. Contrairement aux apparences, les branches du Y ne symbolisent pas les deux entrées classiques de l'ALU, mais plutôt les flux de données qui traversent l'ALU. Le premier flux, représenté par les carrés noirs, est le flux d'opcodes provenant de l'Instruction Register, qui indique à l'ALU le type d'opération à effectuer (addition, soustraction, comparaison, etc.). Les deux autres flux, avec des carrés blancs pour les données entrantes et des carrés gris pour les données sortantes, représentent les informations de données que l'ALU reçoit et renvoie.

La configuration de l'ALU dans ce schéma montre que les **données de l'ALU** ne sont pas directement connectées à la mémoire centrale, mais uniquement à la base des registres. Cela signifie que les résultats produits par l'ALU sont d'abord stockés dans les GPRs avant d'être éventuellement écrits dans la mémoire centrale.

L'architecture Von Neumann : un flux partagé pour instructions et données

Le modèle représenté ici correspond à l'**architecture Von Neumann**, où instructions et données partagent les mêmes canaux de communication (ou "fils") entre le CPU et la mémoire centrale. Dans cette architecture, les instructions et les données transitent par les mêmes chemins, ce qui simplifie la structure matérielle mais peut engendrer des **conflits de flux** lorsqu'instructions et données doivent être accédées simultanément. Ce phénomène, connu sous le nom de **goulot d'étranglement de Von Neumann**, limite les performances car le CPU doit parfois attendre que le bus soit libre pour accéder aux instructions ou aux données.

En pratique, cela signifie que lorsque le CPU accède à une instruction en mémoire, il peut être temporairement empêché de charger ou d'écrire des données jusqu'à ce que le bus soit de nouveau disponible. Cette limitation peut ralentir l'exécution des programmes, notamment dans les applications qui nécessitent des transferts fréquents de données et d'instructions.

Découpage de l’instruction : Les étapes de l’exécution

Pour comprendre le déroulement de l'exécution d'une instruction dans le CPU, nous la découpons en quatre étapes distinctes : **fetch**, **decode**, **execute** et **write**. Ces étapes permettent de gérer les instructions de manière ordonnée et efficace. Pour illustrer ce processus, nous prenons l’exemple d’une opération d’addition.

Étape 1 : Fetch (Récupération de l’instruction)

Le **fetch** est la première étape du cycle d’instruction. Ici, le CPU **va chercher** en mémoire centrale (RAM) l’instruction suivante à exécuter dans le programme. Le compteur ordinal (PC) pointe vers l’adresse de cette instruction en mémoire, permettant au CPU de la récupérer.

L'opération de fetch peut être complexe si l'instruction nécessite plusieurs octets pour être complète. Par exemple, certaines instructions contiennent des **données immédiates** ou des informations supplémentaires nécessitant un octet ou plusieurs pour être interprétées correctement. Dans notre cas simplifié, le **fetch** inclut ces données si elles sont nécessaires pour compléter l’instruction avant de passer à l’étape suivante.

Étape 2 : Decode (Décodage de l’instruction)

Le **decode** est l’étape où le CPU décode l'instruction pour en comprendre le contenu et la nature. Cette opération se fait dans l’**Instruction Register** (IR), qui lit l’instruction et extrait son **opcode** pour déterminer l’action à réaliser.

Dans notre exemple d’addition, l’opcode spécifie qu’il s’agit d’une addition et indique à l’IR de préparer le CPU à cette opération. Le registre identifie également les **opérandes** nécessaires (les valeurs à additionner) et vérifie les registres sources où ces valeurs sont stockées. Le décodage est donc essentiel pour orienter le CPU vers l’unité de calcul appropriée et pour préparer les registres à fournir les données.

Étape 3 : Execute (Exécution de l’instruction)

L’étape d’**exécution** est celle où l’instruction est effectivement réalisée par le CPU. Pour notre exemple d’addition, nous divisons cette étape en deux sous-étapes : **read** et **add**.

1. **Read** : Durant cette sous-étape, les **données sont lues dans les registres** et transférées vers l’entrée de l’**ALU** (Arithmetic Logic Unit). Cette action place

les deux opérandes nécessaires (les valeurs à additionner) dans l'ALU, en préparation de l'addition proprement dite.

2. **Add** : La sous-étape add correspond à l'**opération d'addition** elle-même. L'ALU additionne les valeurs présentes dans ses entrées et génère le résultat, prêt à être utilisé ou stocké. Cette opération utilise les opérandes fournis par le registre et l'opcode transmis par l'IR, qui a spécifié qu'une addition devait être réalisée.

L'étape d'exécution peut prendre plus de cycles d'horloge si l'opération est complexe. Dans ce cas, une addition est simple, mais des calculs plus compliqués, comme une multiplication ou une division, pourraient nécessiter plusieurs cycles.

Étape 4 : Write (Écriture du résultat)

La dernière étape, **write**, consiste à **sauvegarder le résultat** de l'opération dans le registre spécifié par l'instruction. Après que l'ALU a produit le résultat, ce dernier est transféré dans un registre où il sera accessible pour des opérations futures ou pour être éventuellement écrit en mémoire centrale.

En stockant le résultat dans le registre, le CPU est prêt pour la prochaine instruction, qui sera récupérée via le compteur ordinal. Cette écriture est essentielle pour la continuité des calculs et des traitements, en permettant au CPU de réutiliser le résultat si besoin.

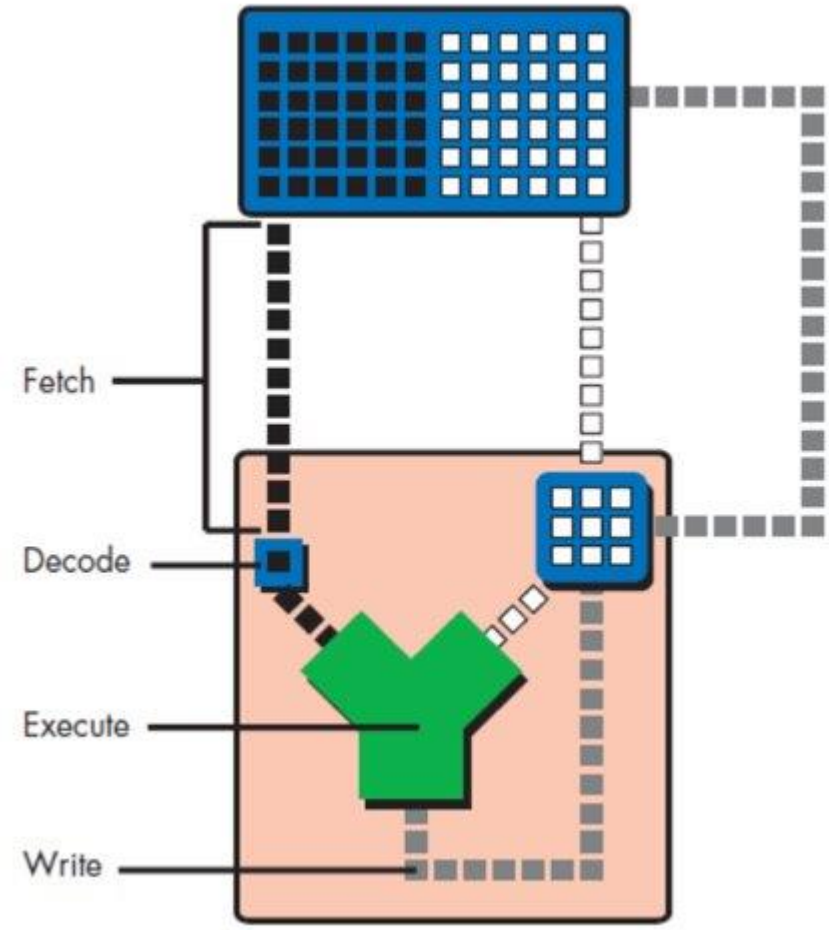
Résumé visuel des étapes d'exécution

Pour mieux comprendre le déroulement de la vie d'une instruction, notre schéma de flux illustre ces étapes clés : le fetch et le decode, puis l'exécution divisée en read et add, et enfin le write. Cette séquence de traitement permet au CPU de gérer chaque instruction avec efficacité en suivant un ordre précis, facilitant la continuité des calculs et des transferts de données dans le programme.

Ces étapes sont les étapes fondamentales de l'exécution d'une instruction, mais nous allons aller plus loin dans le découpage de l'instruction et fusionner les étapes fetch et decode, puis, subdiviser l'étape exécute.

Justification :

- le fetch dépend de l'instruction.
- pour exécuter une opération (add), on doit d'abord lire les données.


Fetch
Decode
Execute
Write

Calcul de la performance du CPU

Première façon de compter

La performance d'un CPU se mesure principalement en évaluant le **temps d'exécution** d'un programme, c'est-à-dire le temps nécessaire pour exécuter toutes les instructions du programme. En effet, chaque instruction s'exécute en série après la précédente, et le **temps d'exécution total** est directement lié au nombre d'instructions exécutées par unité de temps, souvent exprimé en **nanosecondes (ns)** dans nos exemples.

Plusieurs facteurs influencent ce temps d'exécution :

1. **Fréquence d'horloge** : Plus la fréquence d'horloge du CPU est élevée, plus celui-ci peut traiter d'instructions par seconde. Par exemple, un CPU fonctionnant à 3 GHz effectue potentiellement 3 milliards de cycles par seconde.
2. **Complexité des instructions** : Les instructions peuvent varier en termes de cycles nécessaires. Les opérations simples (comme une addition) prennent souvent un seul cycle, tandis que des instructions complexes (comme une multiplication) peuvent en nécessiter plusieurs.
3. **Taille des instructions** : Le CPU doit charger les instructions en mémoire. Les instructions de plus grande taille peuvent ralentir le processus si elles nécessitent plusieurs cycles pour être chargées et exécutées.
4. **Qualité du compilateur** : L'efficacité du programme dépend aussi de la manière dont le **compilateur** convertit le code en instructions machine. Un compilateur performant peut optimiser le programme pour minimiser le nombre d'instructions nécessaires, améliorant ainsi le temps d'exécution.

Ainsi, la performance d'un CPU est le résultat de la combinaison de ces facteurs, chacun contribuant à la rapidité globale du traitement des programmes.

Deuxième façon de compter

Une autre méthode pour évaluer la performance d'un CPU consiste à mesurer le **nombre d'instructions qui "sortent leurs effets" par unité de temps**. Cette approche permet de considérer non seulement le nombre d'instructions exécutées, mais également leur impact réel sur l'état du programme.

La notion de **"sortir ses effets"** se réfère au moment où une instruction a complètement terminé son exécution et où ses résultats sont visibles ou utilisables par les instructions suivantes. Par exemple, dans une addition, "sortir

ses effets" signifie que le résultat de l'opération a été calculé par l'ALU et stocké dans un registre, prêt pour toute autre instruction dépendant de ce résultat.

Ce calcul prend en compte plusieurs aspects :

1. **Instructions dépendantes** : Certaines instructions ne peuvent "sortir leurs effets" qu'une fois qu'une instruction antérieure a produit un résultat. Cela introduit des dépendances qui peuvent ralentir le flux d'exécution.
2. **Pipeline et parallélisme** : Les CPU modernes utilisent des techniques comme le **pipeline** et l'**exécution en parallèle** pour traiter plusieurs instructions simultanément. Cependant, même dans ces configurations, seules les instructions ayant effectivement "sorti leurs effets" sont prises en compte dans cette mesure.

Cette méthode de mesure offre une vision plus réaliste des performances, car elle reflète le nombre d'instructions utiles produisant un impact tangible sur le programme par unité de temps, en tenant compte des dépendances et des techniques de traitement simultané.

Le parallélisme ne sera pas abordé ici, mais bien le pipeline.

Le pipeline dans les CPU : Principe et profondeur

Le **pipeline** est une technique d'optimisation utilisée dans les CPU pour augmenter le débit d'exécution des instructions. Au lieu de traiter une instruction à la fois du début à la fin, le pipeline divise le processus d'exécution en plusieurs **étapes**. Chaque étape traite une partie de l'instruction, de sorte que plusieurs instructions peuvent être en cours de traitement simultanément, chacune à une étape différente du pipeline.

Ce concept peut être comparé à une chaîne de montage en usine : chaque poste effectue une tâche spécifique sur un produit, et dès qu'il a terminé, il passe au poste suivant. De cette manière, une chaîne bien organisée peut traiter plusieurs produits en parallèle, chacun passant par les étapes du processus, ce qui accélère le traitement global.

La profondeur du pipeline

La **profondeur du pipeline** correspond au nombre d'étapes que le pipeline comporte. Par exemple, un pipeline à **profondeur 4** comprend quatre étapes, comme suit :

1. **Fetch** (récupération de l'instruction en mémoire),

2. **Decode** (décodage de l'instruction pour identifier l'opération),
3. **Execute** (exécution de l'opération, par exemple une addition dans l'ALU),
4. **Write** (écriture du résultat dans le registre ou la mémoire).

Chaque étape du pipeline dure 1 nanoseconde (ns), ce qui signifie que chaque instruction prend 4 ns pour être complètement exécutée dans notre exemple.

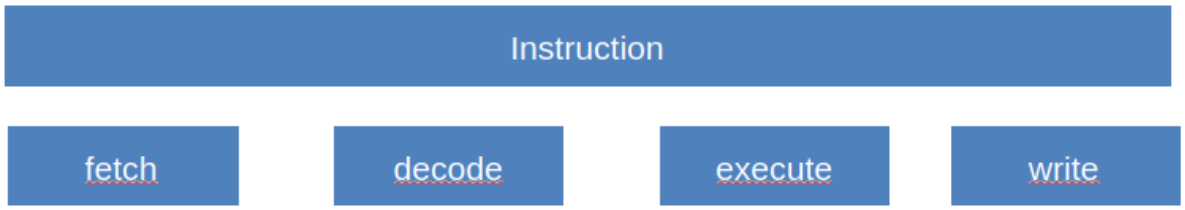
Comparaison entre exécution avec et sans pipeline

Pour comprendre l'impact du pipeline, imaginons un programme dans lequel cinq instructions doivent être exécutées, chaque instruction nécessitant 4 étapes.

1. **Exécution sans pipeline** : Dans un CPU sans pipeline, chaque instruction doit être terminée avant que la suivante puisse commencer. Ainsi, chaque instruction prendra 4 ns pour être entièrement exécutée. Pour les cinq instructions, le temps total d'exécution est calculé comme suit :

   $$\text{Temps total sans pipeline} = 4\ \text{ns} \times 4\ \text{instructions} = 16\ \text{ns}$$

2. **Exécution avec pipeline** : Avec un pipeline de profondeur 4, les instructions peuvent être décalées d'une étape, ce qui permet de les exécuter simultanément à différents stades. Par conséquent, chaque instruction commence à 1 ns d'intervalle, comme si elle était décalée dans le pipeline, jusqu'à ce que toutes les étapes soient occupées.
   a. Instruction 1 : commence au début
   b. Instruction 2 : commence après 1 ns
   c. Instruction 3 : commence après 2 ns
   d. Instruction 4 : commence après 3 ns
   e. Instruction 5 : commence après 4 ns.



Après chaque étape, les données passent à l'étape suivante. L'instruction suivant commence donc directement après la première étape, et le flux continue comme cela au fur et à mesure.

À partie de la quatrième étape, l'instruction 1 sort ses effets. À l'étape suivante (1ns), c'est l'instruction 2 qui sort ses effets et ainsi de suite.

Après l'achèvement de la 4ème étape de l'instruction 1, le pipeline est "rempli" et une instruction est terminée toutes les 1 ns. Ainsi, le temps total pour exécuter les cinq instructions est :

Temps total avec pipeline=4 ns pour 4 instructions

En utilisant le pipeline, le CPU exécute toutes les instructions en seulement 4 ns, contre 16 ns sans pipeline. On a donc un gain de 4 (on va 4 fois plus vite) avec un pipeline de profondeur 4. Il s'agit ici d'une valeur théorique qui peut être impactée par certains effets.

Comparatif des deux modèles

Le pipeline améliore considérablement l'efficacité en permettant d'exécuter plusieurs instructions en parallèle, chacune étant à un stade différent du processus. Cela conduit à plusieurs avantages :

1. **Réduction du temps d'exécution total** : Dans notre exemple, le pipeline a réduit le temps d'exécution total des cinq instructions de 16 ns (sans pipeline) à 8 ns (avec pipeline). Cette réduction est possible car le pipeline permet au CPU d'initier l'exécution de chaque instruction sans attendre la fin complète de la précédente.
2. **Débit accru** : Grâce au pipeline, le débit d'instruction augmente. Dans notre exemple, une instruction est complétée chaque nanoseconde une fois le pipeline rempli. En d'autres termes, le CPU peut traiter plus d'instructions par unité de temps avec le pipeline qu'en mode séquentiel.
3. **Utilisation efficace des ressources** : Avec le pipeline, chaque étape du CPU est utilisée de manière continue, minimisant ainsi les périodes d'inactivité. Par exemple, pendant que la première instruction est dans l'étape 4 (write), la deuxième instruction est dans l'étape 3 (execute), la troisième instruction est en phase de decode, et ainsi de suite. Cela maximise l'utilisation des unités de traitement.

Cependant, le pipeline a aussi ses limites :

- **Complexité de gestion** : Plus le pipeline est profond, plus il est difficile de gérer les dépendances entre instructions, car certaines instructions peuvent nécessiter des résultats de précédentes. Ce problème est appelé **brouillage de pipeline** (pipeline hazard).
- **Latence d'amorçage** : Bien que nous n'ayons pas considéré l'amorçage du pipeline dans cet exemple, le pipeline doit être "rempli" avant d'atteindre son

débit maximal. Cette phase d'amorçage crée un délai initial pour atteindre le rendement optimal.

En somme, le pipeline est une technique essentielle pour améliorer les performances du CPU en augmentant le débit d'exécution des instructions. Elle est particulièrement efficace pour des tâches qui se prêtent bien à une division en étapes régulières et indépendantes. Néanmoins, cette technique exige une gestion rigoureuse pour minimiser les retards liés aux dépendances, tout en assurant que chaque étape du pipeline est utilisée de manière optimale.

Impact de l'approfondissement du pipeline et ses limites

L'approfondissement du **pipeline** est une technique utilisée pour diviser chaque étape d'instruction en sous-étapes plus courtes. En fragmentant ainsi le processus, on augmente le **débit d'instructions** (completion rate) car chaque sous-étape prend moins de temps à exécuter, permettant un flux plus rapide d'instructions dans le pipeline. Cependant, cette stratégie présente des limites, et diviser indéfiniment chaque étape n'entraîne pas toujours une amélioration de la performance.

Exemple de division des étapes d'instruction

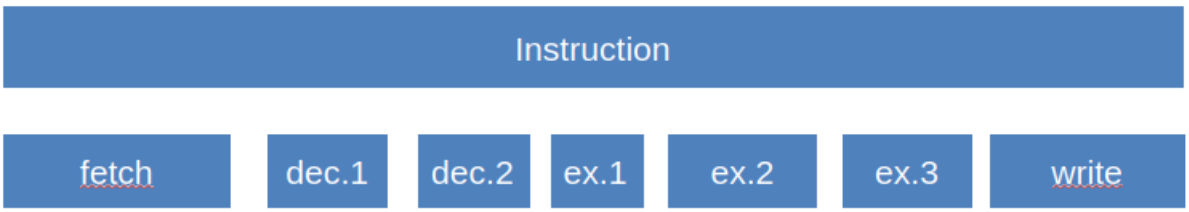
Imaginons une instruction initialement divisée en quatre étapes, chacune prenant 1 nanoseconde (ns) pour un temps total de 4 ns. Ces étapes sont les suivantes :

- **E1** : 1 ns
- **E2** : 1 ns
- **E3** : 1 ns
- **E4** : 1 ns

Pour approfondir le pipeline, on pourrait subdiviser chaque étape en plusieurs sous-étapes. Cela donnerait la configuration suivante :

- **E1** : divisée en trois sous-étapes
  - E1.1 = 0,2 ns
  - E1.2 = 0,4 ns
  - E1.3 = 0,4 ns
- **E2** : divisée en trois sous-étapes
  - E2.1 = 0,3 ns
  - E2.2 = 0,3 ns
  - E2.3 = 0,4 ns

- **E3** : divisée en quatre sous-étapes
  - E3.1 = 0,2 ns
  - E3.2 = 0,2 ns
  - E3.3 = 0,2 ns
  - E3.4 = 0,3 ns
- **E4** : divisée en trois sous-étapes
  - E4.1 = 2 ns
  - E4.2 = 0,6 ns
  - E4.3 = 2 ns



(photo non contractuelle)

En subdivisant ainsi les étapes, on obtient un pipeline de profondeur 14, ce qui augmente le débit théorique du CPU, car chaque instruction traverse le pipeline en sous-étapes plus courtes.

Limites de la subdivision des étapes

La division des étapes en sous-étapes n'apporte des bénéfices que si la durée de la plus grande sous-étape est inférieure à la durée des étapes initiales. Dans notre exemple, l'étape la plus courte de la configuration d'origine était de 1 ns. En subdivisant chaque étape, nous avons introduit des sous-étapes dont les durées vont de 0,2 ns à 2 ns. Cependant, certaines sous-étapes (comme E4.1 et E4.3, de 2 ns chacune) dépassent la durée initiale de 1 ns.

Cette limite implique que l'approfondissement du pipeline n'a de sens que si la **plus grande sous-étape est inférieure à la plus petite des durées d'étape initiales**. Autrement, le bénéfice potentiel est annulé, car une sous-étape plus longue ralentit l'ensemble du pipeline, empêchant les autres étapes de se compléter à pleine capacité.

Amélioration du completion rate

Un des avantages de l'approfondissement du pipeline est l'augmentation du **completion rate** (débit d'instructions terminées par unité de temps). Dans notre exemple, avant la subdivision, le CPU traitait une instruction en 4 ns, ce qui donnait un completion rate de :

$$\frac{1 \text{ instruction}}{4 \text{ ns}} = 0{,}25 \text{ instruction/ns}\}$$

Après la subdivision en 14 sous-étapes, le pipeline complet se remplit à un rythme plus rapide, avec une nouvelle instruction initiée toutes les 0,6 ns. Ainsi, le nouveau completion rate est :

$$\frac{1 \text{ instruction}}{0{,}6 \text{ ns}} \approx 1{,}67 \text{ instruction/ns}\}$$

Ce completion rate accru indique que le pipeline est capable de traiter un flux d'instructions plus élevé, ce qui se traduit par une performance accrue dans les programmes qui nécessitent un grand nombre d'instructions simples.

Impact sur le temps d'exécution total d'une instruction

Bien que le completion rate augmente avec l'approfondissement du pipeline, le **temps d'exécution total d'une instruction** peut paradoxalement augmenter. Dans l'exemple initial, chaque instruction prenait 4 ns. Avec les sous-étapes supplémentaires, chaque instruction prend maintenant :

$$0{,}6 \text{ ns} \times 14 = 8{,}4 \text{ ns}$$

Le temps d'exécution d'une instruction est donc passé de 4 ns à 8,4 ns, bien que le completion rate ait considérablement augmenté. Cette augmentation du temps d'exécution par instruction est due aux sous-étapes plus longues (comme E4.1 et E4.3 de 2 ns), qui ralentissent l'avancement de chaque instruction individuelle à travers le pipeline. En d'autres termes, le pipeline traite davantage d'instructions en parallèle, mais chaque instruction individuelle met plus de temps à atteindre la fin.

Comparatif des deux modèles

Modèle	Temps par instruction	Completion rate	Temps total d'une instruction
Sans subdivision	4 ns	0,25 instruction/ns	4 ns
Avec subdivision	8,4 ns	1,67 instruction/ns	8,4 ns

Ce comparatif montre les avantages et inconvénients de l'approfondissement du pipeline. D'un côté, le completion rate augmente considérablement, permettant au CPU de traiter davantage d'instructions en parallèle. De l'autre, le temps d'exécution d'une instruction individuelle s'allonge, ce qui peut poser problème pour des tâches nécessitant des résultats immédiats.

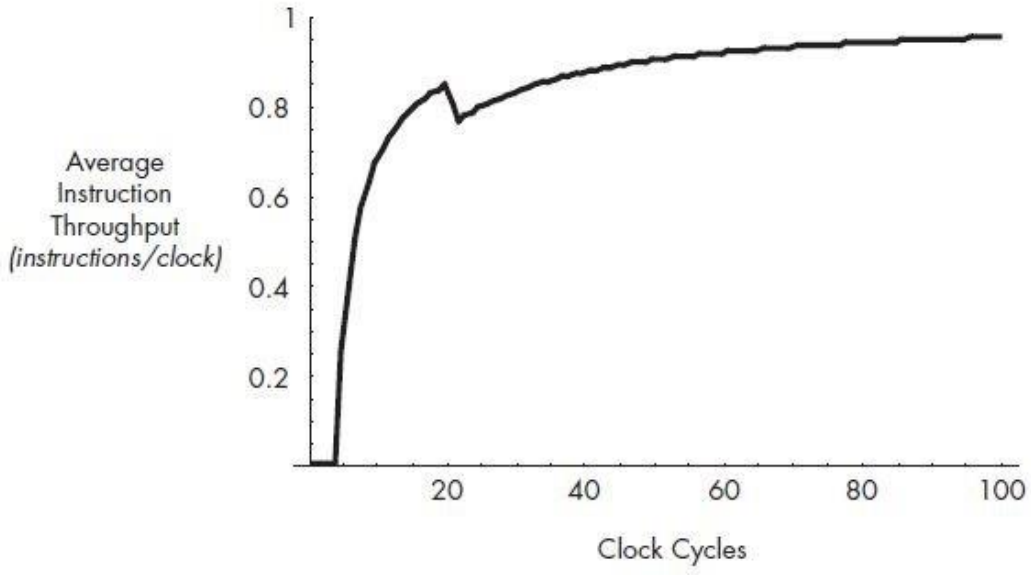
Conclusion

L'approfondissement du pipeline est une méthode efficace pour augmenter le débit d'instructions dans un CPU. Cependant, cette technique a ses limites, car subdiviser chaque étape n'est utile que si la plus grande des sous-étapes reste inférieure à la durée des étapes initiales. Un pipeline trop profond peut entraîner une augmentation du temps d'exécution par instruction, même si le completion rate augmente. Pour cette raison, l'approfondissement du pipeline doit être planifié avec soin, en équilibrant la profondeur du pipeline et la durée des sous-étapes pour optimiser à la fois le completion rate et la performance globale du CPU.

Le décrochage du pipeline

Dans le cadre de l'explication du **décrochage du pipeline** illustré dans les graphiques, le concept de **"bulle" de pipeline** ou de décrochage correspond aux moments où le pipeline ne peut pas avancer au rythme prévu. Ces bulles se forment lorsqu'une étape dans le pipeline prend plus de temps que prévu, ce qui crée un vide temporaire dans la séquence de traitement des instructions.

Premier graphique : Décrochage pour un cycle au vingtième cycle d'horloge

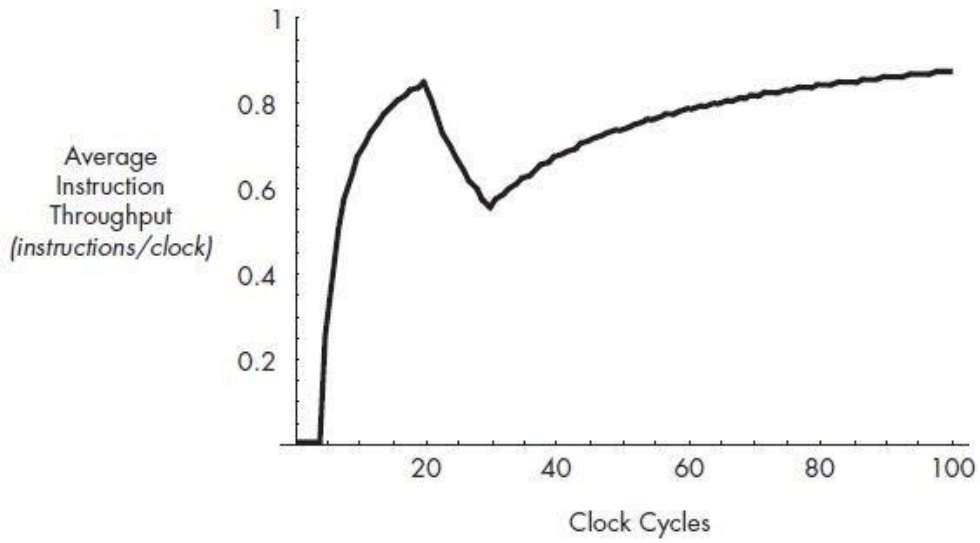


Dans le premier graphique, nous observons un décrochage d'un seul cycle d'horloge au vingtième cycle. Cela signifie qu'à ce moment-là, l'une des étapes du pipeline a rencontré un délai, provoquant une **pause temporaire** dans la progression du pipeline. Cette pause peut être due à plusieurs raisons, comme des instructions nécessitant plus de cycles pour être exécutées ou des

dépendances de données. Le décrochage pour un cycle entraîne une réduction momentanée du débit d'instructions. Cependant, une fois la bulle passée, le pipeline reprend son rythme normal, avec une interruption mineure de la performance globale.

Deuxième graphique : Décrochage de dix cycles au vingtième cycle d'horloge

Le second graphique présente un décrochage beaucoup plus important, avec une pause de dix cycles au vingtième cycle d'horloge. Ici, l'impact est beaucoup plus significatif, car dix cycles d'horloge sont nécessaires avant que le pipeline puisse reprendre son rythme normal. Ce type de décrochage massif peut être causé par des instructions complexes, telles que des opérations de multiplication ou division sur des données en virgule flottante, ou par des dépendances critiques, comme des sauts conditionnels nécessitant une décision avant de continuer.



Impact sur la performance

Ces décrochages réduisent le **completion rate** du pipeline, c'est-à-dire le nombre d'instructions terminées par unité de temps. Avec un décrochage de dix cycles, le pipeline est inactif pendant une période prolongée, ce qui diminue le débit d'instructions et augmente le temps total d'exécution des programmes. Ces bulles de pipeline mettent en lumière les limites de cette architecture, où les instructions ne peuvent pas toujours être exécutées de manière parfaitement séquentielle et continue.