Architecture des ordinateurs 1

Romain Deleuze

26 décembre 2019

Table des matières

Chapitre 1

Concepts de base

Lecture 1: Instructions, registre et ALU

Mardi 17 September 2019 8 :3

1.1 Instructions

1. Une série d'instructions arrivent au cœur de l'ordinateur. Traitées par le cpu pour avoir un résultat. Exemple :
   - 1. Instruction : addition
   - 2. Données : 1 et 2
   - 3. Résultat : 3

   Notion de flux continu
2. Explications exemple :
   - 1. Lire les nombres
   - 2. Modifier les nombres
   - 3. Écrire les nombres

   Conséquence sur le design des composants -> Accélérer les actions.

3. En détails :
   - 1. Lecture/Écriture -> nécessite une zone modifiable pour le stockage des données
   - 2. Modifier -> Nécessite une ALU pour modifier les données. Celle-ci faisant partie du CPU. Partie du CPU qui execute le calcul.
   - 3. Manipuler les données -> nécessite un chemin, les bus.
     - 1. Data Bus -> ensemble des fils qui traitent les données.
     - 2. Instruction bus -> ensemble des fils qui transportent les instructions. L'ensemble des fils -> bus.
   - 4. Résultat -> les données résultantes vont être renvoyées dans la zone mémoire. (écrire)

1.2 Registres

1. Exemple de réflexion architecturale : Les registres proches de l'ALU ont de meilleurs performances dans le traitement de données. -> Registres internes au CPU. Registre -> petit espace de stockage, rapide incorporé au CPU.

2. Exemple de l'addition : A+B=C
   - 1. Obtenir depuis les registres sources (A et B)
   - 2. Additionner les nombres.
   - 3. Placer le résultat dans le registre de destination (C) -> Écrase ce qui se trouvait dans le registre(0 et 1) -> les registres ne sont jamais vides.
3. **La mémoire RAM**
   - 1. Les registres sont petits -> contiennent peu de données.
   - 2. Données utiles -> déplacées vers les registres(accumulateur).
   - 3. Le reste -> dans **main memory** composée de mémoire de type **RAM(Random access memory) -> Accès RW(Lecture/Écriture)**, supprimée si plus de courant, besoin de **rafraichir la mémoire** pour garder les infos.).
   - **RAM statique** -> conserve la info sans rafraichissement.

   Schema : Main memory : Instructions + données(lire) CPU = ALU + registres (modifier) in memory bus Écrire : Résultat -> données in main memory
4. Exemple Addition : A+B=C
   - 1. Charger les deux opérandes (A et B) depuis la mémoire centrale (main memory) vers deux registres sources.
   - 2. Additionner :
     - a. Lire le contenu des registres A et B
     - b. Additionner les contenus des registres A et B
     - c. Écrire le résultat dans le registre C
5. **Le programme : code stream** Le code -> série d'instructions de commandes qui dépend du type de CPU(X86, ARM...) -> précise ce que la machine doit faire. -> exemple : Entrée reset.

   Exemple :
   - 1. Instruction **load** pour charger les nombres depuis la mémoire vers les registres
   - 2. Instruction **add** pour que le ALU réalise l'addition.
   - 3. Instruction **store** pour que l'ordinateur place le résultat en mémoire.
6. **Catégorie d'instructions**
   - 1. Arithmétique : add, sub, mul, div
   - 2. Accès mémoire : load, store

— 3. Branchement (jump)

— 4. Logique (conditions) : and, or, not

7. **Architecture basique et format d'instructions** Basé sur une machine fictive : **AR1** composé de :

— 1. 1 ALU

— 2. 4 registres : A, B, C, D

— 3. 256 cellules mémoires addressable de 0 à 255

(a) **Format des instructions :**

```
instruction source1, source2, destination
```

Exemple :

```
add A, B, C
#Passage dans le compilateur transforme les nbs binaires.
```

(b) **Format des instructions d'accès mémoire**

```
Instruction source, destination
#exemple:
load \#12, A
#exemple d'un programme:
load \#12, A
load \#13, B
add A, B, C
store C, \#14
#Les donnees en source ne changent pas.
```

(c) Limites de l'exemple : Comment connaitre le contenu des cellules #12 et #13.

8. **Valeurs immédiates et jeux sur les addresses**

(a) Utilisation d'une valeur à la place d'une adresse.

```
add, A, 2, A
# Ajouter la valeur 2 au contenu du registre A et ecrire
    le result dans A (en ecrasant ce qui s'y trouvait)
#Utilisation du symbole # pour aller chercher une adresse.
```

(b) 2ème programme (utilisation de D sans savoir sa valeur) :

```
load #D, A ;Contenu de #D = 12, valeur de la cellule 12
    dans A
load #13, B ;Contenu de la cellule 13 dans B
add A, B, C ;Add A+B -> result dans C
store C, #14 ;Stocker C dans cellule 14
```

Le # pointe vers la case mémoire contenue dans le registre mentionné

(c) 3ème programme :

```
load #11, D ;Contenu de la cellule 11 dans D(pointeur)
load #D, A ;Contenu de la cellule D dans A (cellule 11)
load #13, B ;Contenu de la cellule 13 dans B
add A, B, C ;Add A+B -> result dans C
store C, #14 ;Stocker C dans cellule 14
```

9. Résumé :

- 1. Mettre une valeur dans un registre
- 2. Récupérer cette valeur en tant qu'adresse de cellule mémoire.
- 3. Lire le contenu de cette cellule.
- 4. Charger ce contenu dans un registre.
- 5. La mémoire est divisée en segments(bits consécutifs) :
  - 1. Certains stockent des données.
  - 2. D'autres stockent du code.

10. **Adressage relatif = adresse de base + offset** Exemples :

```
load #(D+108), A
store B, #(D+108)
```

11. **Les adresses mémoires et les nombres entiers** : sont stockés dans les mêmes registres appelés : **general purpose registers(GPRs)**
    AR1 : A,B,C,D = GPRs.
12. **Autres registres :** Registre de **statut & registre de contrôle**.
    Schema + bus de contrôle, tout est addressable.

Chapitre 2

Les mécanismes d'exécution d'un programme

Lecture 2: Mécanismes d'exécution Mardi 23 Septembre 13 :10

2.1 Opcodes et languages machines

1. Dans un ordi
   - Instruction = suite de 0 et 1
   - Programme = suite d'instructions
   - programme = longue suite de 0 et 1 (transformée en mnémonics pour la lecture)
   - Programme stocké en MR centrale
2. Language humain = **Mnémonics**
3. Programmer en assembleur = mapping = Mnémonics -> Opcodes

2.2 Mapping mnémoniques et registres (AR1)

1. Mnemonic et Opcode
   - add = 000
   - sub = 001
   - load = 010
   - store = 011
2. Registre et binary code
   - A = 00
   - B = 01
   - C = 10
   - D = 11

2.3 Instruction add avec registres

- 0 = mode
- 1,2,3 = opcode
- 4,5 = source 1
- 6,7 = source 2
- 8,9 = destination
- 10->15 = 00000000
- Si le premier bit est 0 -> manipulations registres
- Si 1 -> Valeurs immédiates sont utilisées
- Les 6 derniers bits complètent l'écriture sur 2 octets, remplir l'espace disponible(padding)
- Exemple : add C, D, A = 00001011 00000000
- 0 premier bit de mode -> Manipulations registres

2.4 Instruction add avec valeurs immédiates

- 0 = mode
- 1,2,3 = opcode
- 4,5 = source
- 6,7 = destination
- 8->15 = 8 bit immediate value
- = instruction set
- Exemple : add C, 8, A = 10001000 00001000
- 1 premier bit de mode -> Valeurs immédiates
- 8 -> sur 8 bits = 00001000

3. Remarques importantes
   - Augmenter le nombre de bits utilisés pour coder une opération ou un register permet d'augmenter le nombre d'opérations et de registres disponibles.
   - 2 bits -> 4 possibilités
   - 3 bits -> 8 possibilités
   - 4 bits -> 16 possibilités
   - 2 exposant nombre de bits - les bits utilisés pour les opérations des registres (destination, mode ect...)
4. Les opérations ainsi disponibles forment **l'instruction set**

2.5 Instruction load avec type immédiat

1. Type immédiat (immediate-type load)
2. Sur 16 bits :
   - 0 = mode -> va être à 1 du à opération immédiate
   - 1,2,3 = opcode
   - 4,5 = source -> 00 -> vu qu'opération immédiate pas de registre padding "00"
   - 6,7 = destination
   - 8->15 = 8 bit immediate value
3. Exemple : load #12, A = 10100000 00001100 -> Charger le contenu de A avec la valeur pointé par l'adresse 12
4. 010 = load # un autre opcode existe pour load sans #

2.6 Instruction load avec registre

1. Type avec registre sur 16 bits :
   - 0 = mode -> 0 car opération avec registre
   - 1,2,3 = opcode
   - 4,5 = source
   - 6,7 = 00
   - 8,9 = destination
   - 10->15 = 8 bit immediate value

2.7 Instruction load avec base + offset

1. Type avec base + offset sur 16 bits :
   - 0 = mode
   - 1,2,3 = opcode
   - 4,5 = base
   - 6,7 = destination
   - 8->15 = 8 bit immediate value
2. Exemple : load #12, A = 10100000 00001100
3. store C, #14 = 10111000 00001110

2.8 L'assembleur

1. Avant :
   - Programme réalisé en logique cablée

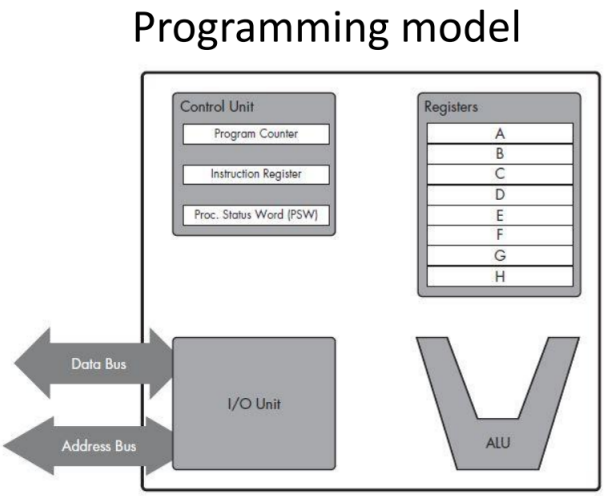
- Charger/modifier/adapter un programme -> modification physique des circuits de l'ordinateur
- Le résultat apparait sous forme binaire également
- un clavier ou un écran aurait demandé un ordinateur à eux-seuls pour fonctionner
- Pas d'intégration de fonctions logiques
- Pas de RAM -> relais = bascules à mémoire

2. Maintenant :
   - Développements technologiques
   - Intégration tech -> plus de puissance
   - Création de mémoires contenant des programmes
   - Traitement de texte en programmes (compilateur)
   - exemples : Bandes perforées

2.9 Compilateur

3. Fonctionnement du compilateur, il va lister les instructions de cette manière :
   - load
   - store
   - add
   - ...
   - il simplifie la tâche
   - Besoin de connaître les ressources dispo :
     - Nombre de registres
     - instructions supportées
     - Adressage mémoire (où)
     - Comment ces registres vont fonctionner

4. Ce modèle se nomme **programming model**



l'ALU et l'I/O unit vont travailler, le bus d'adressage n'est utilisé que par le microprocesseur pour donner les adresses.

2.10 program counter

1. le CPU va lire les instructions à suivre dans le programme. Pas modifiable à priori sauf dans certains cas.
2. Pointe vers la prochaine adresse à lire ou à écrire.
3. Met l'info sur le bus d'adresse.
4. Incrémentage automatique par le microprocesseur. JMP instructions peuvent modifier celui-ci.

— opcode
— source
— destination
— mode

2.11 instruction register

1. Lorsque l'instruction est lue, elle sera mise dans l'instruction register pour être décoder et exécutée.
2. Flux de données et flux d'instructions.
3. Permet au microprocesseur de savoir quelle instruction exécuter. Les registres servent comme destination pour les résultats des instructions, ils peuvent aussi servir comme sources.

2.12 Proc Status Word (PSW)

1. Sert pour les instructions JMP, garde le statut d'une condition (JPMZ, JMPN, JMPE)
2. Sert à définir l'overflow, le négatif...

2.13 Stack Register

1. Si saut -> utilisation du PC en y stockant la valeur du PC depuis lequel on jump dans la stack Jump vers l'élément = call
2. Puis retour dans le cours du programme grâce à la valeur dans la stack au dessus de la pile et l'enlever de cette pile vers le PC = return

2.14 Fetch-Decode-Execute

1. **Fetch** est une action du CPU qui **lit une donnée en MR**
2. A l'adresse **PC** (program counter)
3. L'envoie à **l'IR (instruction register)**
4. **Decode opcode/source/destination/mode**
5. **Execute l'instruction quand il a besoin dans un cas complexe**
6. Le PC est incrémenté (ou modifié par l'instruction exécutée)
7. Quand l'exécution est terminée, le cycle reprend

2.14.1 exemple

1. PC = 500 -> IR = load #12, A ; Envoi du contenu de la cellule 12 dans le PC et le bus d'adresse
2. PC = 502 -> IR = load #13, B
3. PC = 504 -> IR = add A,B,C ; Plus rien dans le bus d'adresse (pas de valeur immédiate) recherche seulement dans le microprocesseur
4. PC = 506 -> IR = store C, #14 ; Envoi de la cellule mémoire 14 dans le bus d'adresse (mémoire RAM ou I/O car besoin d'écriture) pour stocker la valeur contenue dans C

2.15 recap dans AR1

1. Les cellules mémoires ont une taille de 1 octet (8 bits)
2. Les instructions ont un format sur 2 octets
3. Pour les instructions, il faut charger 2 octets pour avoir une instructions
4. Il faut incrémenter le PC de 2 à chaque instruction

Chapitre 3

CPU

Lecture 3

Lundi 30 Septembre 2019 13 :

3.1 CPU

3.1.1 L'horloge

1. Étapes vues jusqu'ici sont réalisées au rythme de 1 pulsation d'horloge
2. L'horloge est in signal cadencé à vitesse (fréquence) fixe par un dispositif dédié
3. Idéalement, un cycle fetch-code-execute doit être réalisé en une impulsion d'horloge
4. A un certain moment les données ne sont pas bonnes, la fréquence d'horloge sert à synchroniser toutes les données pour s'exécuter proprement

3.2 Les instructions de saut inconditionnel

1. Unconditional branch = jump #target
2. Permet de continuer l'exécution d'un programme à un autre endroit dans la mémoire
3. La cible du saut (target) est une valeur immédiate (#500) ou une adresse stockée dans un registre (#D ou #(D+103))
4. Charge une valeur dans le PC (program counter)
5. Attention : registre et bus d'adresse doivent être de même longueur (en bits) sinon utilisation de plusieurs registres si plus long -> création de banque de données (bits qui ne changent pas)

3.3 Les instructions de saut conditionnel

1. Conditional Branch = jumpx #target

2. Le PC est chargé si la condition est remplie
3. La condition est souvent un bit du PSW
4. Le PSW a plusieurs bits de "résultats d'opérations"
5. 1(true) ou 0(false) sera écrit dans le PSW selon une condition bien précise
6. Pour vérifier que la condition est remplie il suffira donc de vérifier le bit approprié dans PSW(flags)

3.3.1 exemple

```
sub A, B, C ; soustraire le nombre contenu dans le registre 1 du
    nombre contenu dans le registre B et stocker le r sultat dans
    C
jumpz #106  ; v rifier le PSW et si le r sultat de l'instruction
    pr c dente est 0, sauter    l'instruction    l'adresse #106.
    Si le r sultat est != 0, continuer    la ligne 20
add A, B, C ; Additioner les nombres contenus dans les registres A
    et B et stocker le r sultat en C
```

Incrementation de 2 sur le nombres d'adresse car chaque instruction codé sur 2 bits

3.3.2 exemples de jump

1. jumpn
2. jumpo

3.3.3 label

Plus facile d'utiliser des labels à la place des adresses mémoires (absolues ou relatives)

```
sub A, B, A ; soustrait A    B, r sultat dans A
jumpz LBL1 ; jump    LBL1 si r sultat est  gal      z ro
add A, 15, A ; ajouter 15    A r sultat dans A
store A, #(D + 16) ; stocker A dans la cellule #D+16
LBL1: ; d finition du label LBL1
  add A, B, B ; ajouter A    B stocker r sultat dans B
store B, #(D+16) ; stocker B dans la cellule #D+16
```

Chapitre 4

Pipelined Execution

lecture4Lundi 07 octobre 2019 8 :30Pipelined Execution

4.1 Rappel du cycle de base

1. Fetch
2. Decode
3. Execute

Execute divisé en 3 se passe dans l'ALU, elle ne traite que des infos provenant de registres

1. Read
2. Add
3. Write

4.2 Remarques

La plupart des micro processeurs actuels traitent

1. 3a(read) et 3b(add) sont comme un groupe
2. La sous-étape 3c(write) traité séparément

4.3 4 phases d'exécution

1. Fetch
2. Decode
3. Execute
4. Read and Add
5. Write

4.3.1 Rappel de l'exécution

3 bus sont entre la RAM et le CPU

1. le bus d'adresse, regroupe les adresses adressables
2. le bus de contrôle, Commandes de lecture et écriture, contient le bus d'horloge pour dire que la donnée est valide (problème de fils plus long, exemple : changer 3 bits d'un coup ne se fait pas en même temps)
3. le bus de données, contient et transporte les données

Dans l'AR1 -> 4 registres ABCD et PC et l'IR

1. il démarre et va exécuter la première instruction -> envoi de la première adresse au PC Les adresses de démarrage sont réservées par le micro processeur
2. envoi de l'adresse jusqu'au bus d'adresse, validation de l'adresse -> je lis ce qui se trouve à l'adresse -> envoi de la cellule mémoire contenue dans la RAM vers le bus de données -> arrive dans l'IR
3. Le PC va être incrémenté après avoir fini de lire une adresse -> lecture de l'adresse suivante -> envoi dans le bus d'adresse -> lecture dans le bus de contrôle -> cellule mémoire arrive dans le bus de données
4. Envoi dans l'IR -> exécution de l'instruction (l'IR est de 2 octets de long, peut stocker une instruction longue de 10 bits)
5. exemple : instruction avec un résultat dans C (add A,B, C - load A, C)

4.4 Program Execute et Completion Rate

Program Execution Time = Number of instructions in program / Instruction Completion Rate Prise de valeur moyenne, chaque instruction prend un temps différent

4.4.1 Completion Rate

1. Nombre d'instructions terminées par l'unité de temps -> instr/ns
2. Ne pas confondre avec le temps d'exécution d'une instruction en ns
3. Dans un CPU sans **pipeline** le rapport entre les 2 est de type inversement proportionnel (l'une est égale à l'inverse de l'autre)

4.4.2 pipeline

1. Ce système permet d'augmenter le Completion Rate sas modifier le temps d'exécution du temps d'une instruction (fin de la relation inversement proportionnel)
2. Certaines instructions mettront plus de temps à s'exécuter avec un CPU à pipeline que sans, celles-ci ne peuvent pas se mettre dans la pipeline

Augmenter les performances

1. 2 pistes
   - Réduire le nombre d'instructions du programme
   - Augmenter le completion rate
2. Nombre d'intructions du programme est fixé pour l'exemple

analogie fabrique de voiture

La chaine de production est composée de 5 équipes distinctes spécialisées dans une seule tâche

1. Fabriquer le chassis, il attend la fin de la chaine pour commencer un nouveau chassis
2. Mettre le moteur sur le chassis
3. Mettre les portes, ailes et le capot
4. Attacher les roues
5. Peindre le véhicule
6. Chaque équipe fini par attendre la fin de la chaîne pour recommencer à construire une voiture

Seconde approche avec pipeline

Si chaque équipe ne s'arrête jamais de travailler et recommence leur travail spécifique au lieu d'attendre que la chaîne soit finie pour recommencer à travailler

Gain

1. Dès la 6ème heure, on passe de 1 véhicule toutes les 5h à 1 véhicule par heure
2. Sans changer le temps total pour fabriquer un véhicule on améliore d'un facteur 5 les performances
3. Pour peu que la chaine de fabrication reste pleine

4.4.3 Processeurs single cycle

1. Single Cycle = absence de pipeline
2. 1 cycle de vie d'instruction = 1 cycle d'horloge = fetch - decode - execute - write
3. les 4 phases vont être exécutées sur une impulsion d'horloge (besoin d'un cycle d'horloge car changement de bits -> attente d'un certain temps, unité de temps = impulsion horloge)
4. Chaque phase va attendre que le cycle soit terminé pour recommencer à travailler

Observations

1. Pour améliorer le Completion rate, il faut cadencer l'horloge de + en + vite
2. Sans dépasser le temps maximum que prend une instruction pour s'exécuter complètement
3. Illustration précédente : un programme de 4 instructions prendra 16ns, soit un completion rate de 0,25 instr/ns

4.4.4 Processeurs avec Pipeline

exemple

1. Avec une profondeur de pipeline de 4
2. Après la première phase, durant la deuxième phase un autre fetch s'exécute
3. chaque phase n'attend pas la fin du cycle, elle travaille directement sur la prochaine instruction
4. exemple : Fetch 1ère instruction -> decode 1ère instruction + fetch 2ème instruction -> Execute 1ère instru + decode 2ème instru + fetch 3ème instru
5. Toutes les 4 nanosecondes une instruction est effectuée est toutes les nanosecondes une instruction sort

Observations

1. Dès le début de la 5ème ns, une instruction est exécutée toutes les ns
2. Dans cette exemple un programme de 4 instructions prendra 8ns, completion rate de 1 instr/ns -> gain de 4 sur le nombre d'instructions traitées par ns(car 4 phases)

4.4.5 L'incidence sur l'horloge

1. L'horloge ne cadence plus une instruction complète mais les phases d'instructions
2. -> Les pulsations sont bien plus courtes (exemple : 4ns -> 1ns)
3. Ceci implique qu'une instruction ne doit plus nécessairement être finie à la fin de chaque cycle d'horloge (ex : avant le début de la 5ème ns dans l'illustration précédente)

Chapitre 5

Pipeline améliorations et problèmes

Lecture 5: Pipeline

Lundi 14 Octobre 2019 13 :10

5.0.1 Incidence sur la durée d'exécution du programme

1. La durée d'exécution d'une instruction n'a pas été modifiée mais la durée d'**exécution totale du programme entier (suite d'instructions) a été fortement réduite**
2. Cela a été réalisé en augmentant le nombre d'instructions terminées par unité de temps
3. Le pipeline rend plus efficace l'utilisation des ressources du CPU en faisant travailler **simultanément ses différentes sous-unités**

5.0.2 Améliorer les performances

1. En passant d'un processeur single cycle à un processeur ayant une profondeur de pipeline de 4, on observe une amélioration d'un facteur 4
2. Que se passerait-il si la profondeur du pipeline passait à 5, 6, 7... avec des durées d'étapes intermédiaires de plus en plus courtes ? L'instruction va durer le même temps, des subdivisions plus petites seront présentes dans les étapes (fetch-decode-execute-write)
3. + d'étapes -> + d'instructions en cours d'exécution dans le pipeline -> + d'intructions traités par secondes
4. Avec un découpage en 8 étapes, on obtient 1 instruction terminée toutes les 0,5 ns
5. Completion rate = 2 instr/s

5.0.3 Limites du pipeline en profondeur 4

1. En 8ns, le pipeline a permis d'exécuter 5 instructions car **certaines cases ne sont pas remplies au début du programme**
2. En single cycle aurait exécuté 2 instructions dans le même temps
3. 5 au lieu de 2 ne se représente pas vraiment comme un gain de 4
4. 5 instructions/8ns donne un completion rate de 0,625 inst/ns ce qui ne représente pas le 1 inst/ns

analyse

1. Pour que le gain du au pipeline devienne appréciable, il faudra que le nombre d'instructions dans le programme soit suffisant
2. Exemple : sur 1000ns, un processeur a pipeline de profondeur 4 aura exécuté 996 instructions au lieu des 250 sans pipeline
3. soit un gain de 3,984 proche de 4
4. 4 est en réalité **un maximum théorique et 3,984 est la moyenne réelle**

Décrochage du pipeline

1. Il peut arriver que des étapes du pipeline nécessitent plus d'un cycle horloge
2. Des "bulles" apparaissent et voyagent alors dans le pipeline, réduisant ainsi ses performances -> Repassage à une case vide -> **L'instruction prend donc plusieurs cycles machine pour s'exécuter**
3. La latence d'une instruction peut traduire le temps réel qu'elle met pour traverser le pipeline en prenant en compte les bulles

5.0.4 Valeurs réelles

1. En pratique un processeur cadencé à 3GHz peut facilement attendre 50 à 120ns des données en provenance de la mémoire centrale
2. Une centaine de ns représentent à ces cadences, quelques milliers de cycles d'horloge
3. Et cela pour un seul accès mémoire. Or il s'en produit des millions lors du déroulement d'un programme...
4. 120ns = 120 sec 3GHz = 3HZ
5. Nombre de cycles d'horloges en 120ns -> t=1/f -> t=1/3 = 3,33333[1] -> 120/t = 360.0

5.0.5 CCL sur le décrochage de pipeline

1. Les bulles doivent absolument être évitées au maximum
2. Elles sont généralement dues à des accès trop lents à la mémoire centrale

Solutions :

1. Mémoire cache
2. D'autres causes et pistes de solutions associées existent

5.0.6 Limites du pipeline

1. Il est difficile de découper une instruction en plusieurs étapes équivalentes qu'on voudrait
2. Toujours étapes finiront par être plus longues que certaines (car indivisibles)
3. Ces étapes plus longues influencent le rythme de l'horloge

En fait on peut même concevoir qu'une instruction réalisée dans un pipeline peut prendre plus de temps pour s'exécuter que dans un single cycle, exemple :

1. Une instruction prend 4ns pour se réaliser en single cycle
2. On la sépare en 4 étapes dont les durées respectives sont w,x,y, z ns (proche de 1ns mais pas parfaitement identiques)
3. x = la **plus longue qui impose le rythme** (x > 1ns) -> temps total = 4 fois x > 4ns

L'horloge est une limite

1. On ne peut pas atteindre l'horloge théorique idéale : horloge sans pipeline/ profondeur du pipeline
2. l'horloge réelle est limitée à une cadence maximale -> surchauffe -> dégradation
3. A une même cadence limite, un processeur avec une profondeur plus élevée mettra alors plus de temps pour se remplir

exemple

1. un pipeline en 4 stages peut être plus rapide qu'une en 8 stages dû aux bulles -> étapes vont durer plus longtemps que la fréquence du CPU -> multiplication des bulles -> dégradation du nombre d'instructions sorties par cycle d'horloge

Problèmes

1. Cout financier -> plus de transistors
2. L'architecture à pipeline est complexe et requiert des composants supplémentaires
3. Ces composants ont un coût
4. Dégagement de chaleurs par les circuits

Chapitre 6

Virtualisation

Lecture 6: Introduction

Lundi 14 Octobre 2019 2 :30

6.1 Introduction

1. hardware
2. SDS : Software defined storage
3. SDN : Software defined network
4. SDDC : Software defined Data center - combinaison des 3

Objectif -> Réduire les couts -> pas besoin de payer pour tester du software sur un hardware bien précis

6.1.1 Sécurité

Rien ne peut sortir du container -> en cas de malware le réseau ne sera pas affecté

6.1.2 Qu'est ce qu'une VM

Simulation de ressources matériels

1. CPU
2. RAM
3. IDE
4. SATA
5. USB
6. NIC
7. COM
8. GPU