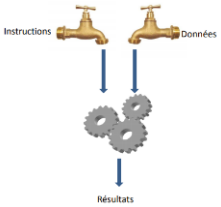
Synthèse :
Architecture des ordinateurs 1 : Concepts

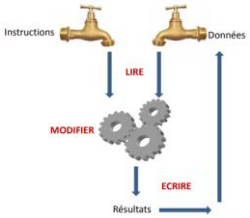
I. Concepts de base

Représentation simplifiée de l'ordinateur :

- Parties d'un ordinateur
  - Unité de traitement / calcul
    - Processeur
    - Carte graphique, …
  - Entrée / sortie
    - Clavier
    - Souris
    - Carte son, …
  - Mémoire
    - HDD
    - SSD, …
  - Outils
    - Carte mère
    - Câble, …
  - Bus
- Première représentation
  - Séparation du flux d'instructions et du flux de données
  - Notion de flux continus
  - Opérations effectuées par l'ordinateur → Opérations arithmétiques



- Deuxième représentation
  - Ajout de la lecture, de l'écriture et de la modification
  - Lire les nombres
  - Ecrire les nombres
  - Modifier les nombres
  - But visé → Améliorer une de ces actions
    - Lecture/Ecriture → Nécessité d'une zone modifiable pour le stockage des données.
    - Modifier → Nécessité d'une unité de traitement des données. L'ALU (Arithmetic and Processing Unit).



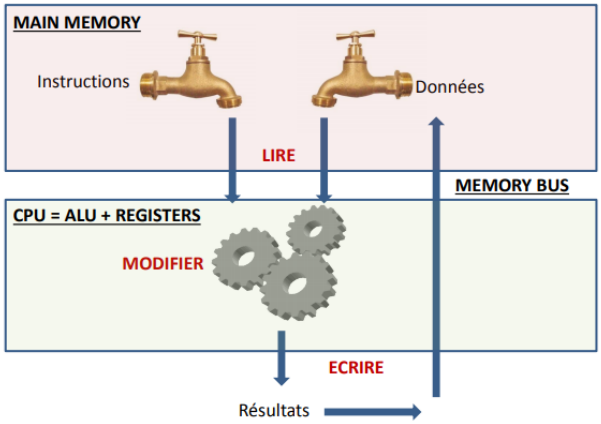
Les registres (Accélération de la lecture) :

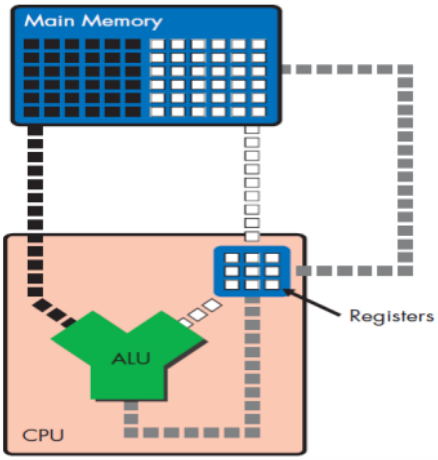
- Afin de rendre la lecture plus rapide. On essaie de maintenir les données utilisées proche de l'ALU.
  - Utilisation des registres
- Registres = Un petit espace de stockage, très rapide, incorporé au CPU.
  - Toutes les données ne sont pas présentes dans les registres.
  - Nombre de registres limités → Coût de fabrication élevé.

La mémoire RAM (Random Access Memory) :

- Création d'une mémoire centrale (main memory) contenant toutes les données car les registres ne peuvent pas contenir toutes les données (Petite taille).
- Au + la mémoire est éloignée du CPU, au + le transfert prendra du temps.
- Transfert :
  - Depuis la mémoire centrale → Délai
  - Depuis le registre → Instantané

Nouvelles représentations





- Exemple (Addition / A+B = C) :
  - Charger les deux opérandes depuis la mémoire centrale vers deux registres sources.
  - Additionner :
    - Lire le contenu des registres A & B
    - Additionner les contenus des registres A et B
    - Ecrire le résultat dans le registre C
  - Stocker le contenu du registre C en mémoire centrale

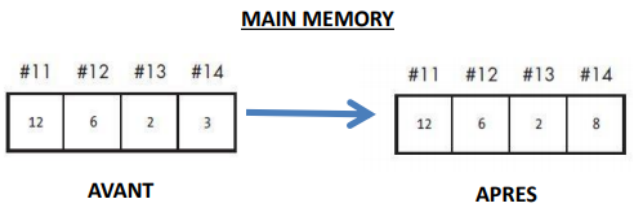
Le programme : code stream

- Le code = Une série d'instructions ou commandes qui précise à la machine toute entière ce qu'elle doit faire.
- Le code va donc au-delà des simples opérations arithmétiques.
- Les instructions
  - LOAD → Charger les nombres depuis la mémoire centrale vers les registres
  - ADD → ALU réalise l'addition
  - STORE → Placer le résultat en mémoire centrale
- Catégories d'instructions
  - Instructions arithmétiques : ADD / SUB / MUL / DIV
  - Instructions d'accès mémoire : LOAD / STORE
  - Instructions d'embranchement (branch instruction)
  - Instructions logiques : AND / OR / NOT / …

Architecture basiques et format d'instructions :

- Format des instructions arithmétiques
  - instruction source1, source2, destination
- Format des instructions d'accès mémoire
  - instruction source, destination
- Exemple :

load #12, A → A = 6
load #13, B → B = 2
add A, B, C → A + B = 8 ⟹ C
store C, #14 → #14 vaut C ⟹ 8



- Problèmes de l'exemple précédent
  - Il y a des milliards de cellules en mémoire. Impossible de savoir ce qu'elles contiennent.

Valeurs immédiates et jeux sur les adresses :

- Utilisation d'une valeur à la place d'une adresse
  - add A, 2, A
- Utilisation du symbole # pour marquer les adresses des cellules
- Exemple :

PROGRAMME 1

- load #12, A
- load #13, B
- add A, B, C
- store C, #14

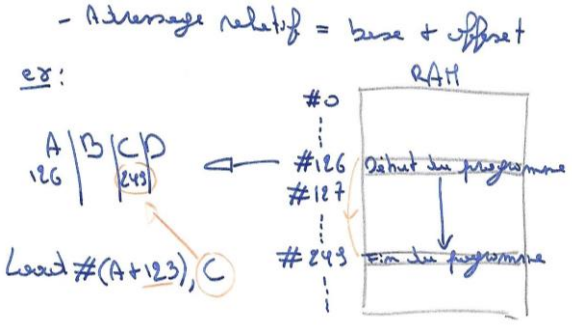
PROGRAMME 2: le contenu du registre D est la valeur 12

- load #D, A
- load #13, B
- add A, B, C
- store C, #14

MAIN MEMORY

AVANT APRES

- Compliquer pour rien ? Quel est l'intérêt ?
  - Utilisation d'une adresse relative basée sur une adresse de base fixe
  - La mémoire est divisée en segments (bits consécutifs)
    - Certains stockent des données
    - D'autres stockent du code
  - Pour charger des cellules dans un segment, il suffit donc de connaître le début du segment et la distance entre la cellule désirée et le début de ce segment (OFFSET)
    - Adresse relative = Adresse de base + Offset
  - Exemples :
    - load #(D+108), A
    - store B, #(D+108)
- Avantages pour le programmeur
  - Pas besoin de connaître la localisation exacte des données en mémoire
  - Juste besoin de connaître le registre dans lequel l'OS place l'adresse de base du segment.
  - Mémoire RAM : RANDOM → La localisation du segment en mémoire peut changer mais tant qu'on sait dans quel registre trouver son adresse de base, on peut retrouver n'importe quelle partie du segment
- Conséquences sur le matériel
  - Possibilité d'utiliser la mémoire RAM
  - Nécessité de calculer les adresses à partir de l'adresse de base → les unités load-store des processeurs actuels supportent des solutions hardware très rapide pour effectuer les additions d'entiers.
  - Les adresses mémoire et les nombres entier sont stockés dans les mêmes registres appelés : general-purpose registers (GPRs)

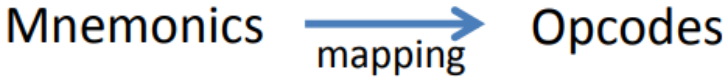


II. Les mécanismes d'exécution d'un programme

- Rappel :
  - Load #A, D → Charger le contenu de la cellule mémoire dont l'adresse est la valeur du registre A. Stocker ce contenu dans le registre D.

Opcodes et langages machines :

- En machine, une instruction est une suite de nombres binaires.
- Un programme est une très longue suite de 0 ou 1 stockée en mémoire centrale.



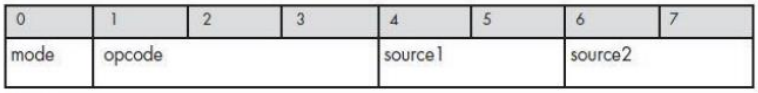
- Mnemonics :
  - Langage humain
  - Symboles représentants les combinaisons de bits du langage machine sous une forme plus facile à manipuler pour l'homme.
  - ex. : add, load, …
- Mapping :
  - Traduction
- Opcodes :
  - Langage machine
  - Partie d'instruction de langage machine
  - ex. : 01011110…

Mapping des mnémoniques et des registres :

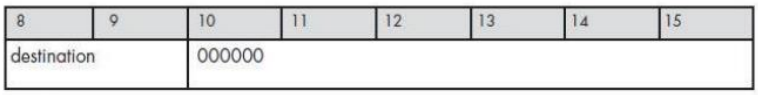
Mnemonic	Opcode	Register	Binary Code
Add	000	A	00
Sub	001	B	01
Load	010	C	10
Store	011	D	11

Les instructions arithmétiques en binaire :

- Exemple sur un format 16 bits



Byte 1



Byte 2

  - Mode :
    - Registre → 0
    - Valeur immédiate → 1
  - Opcode :
    - cf. tableau au-dessus (ex. sub → 001)
  - Source1, 2 :
    - Registres sources en binaire
  - Destination :
    - Registre de destination
  - 000000 :
    - Padding (Ne sert qu'à remplir l'espace vide

Exemple : Add D, B, C	Octet 1 : 0 | 000 | 11 | 01 Mode | Add | D | B Octet 2 : 10 | 000000 C | Padding

- Exemple avec des valeurs immédiates

0	1	2	3	4	5	6	7
mode	opcode			source		destination	

Byte 1

8	9	10	11	12	13	14	15
8-bit immediate value							

Byte 2

  - Transformer la valeur immédiate en binaire et la mettre dans le deuxième octet.
- Remarques
  - Augmenter le nombre de bits utilisés pour coder une opération ou un registre permet d'augmenter le nombre d'opérations et de registres disponibles.
  - Adressage sur 2 bits → 4 possibilités, 3 bits → 8 possibilités, …
  - Les opérations disponibles forment **l'instruction set**

L'instruction load :

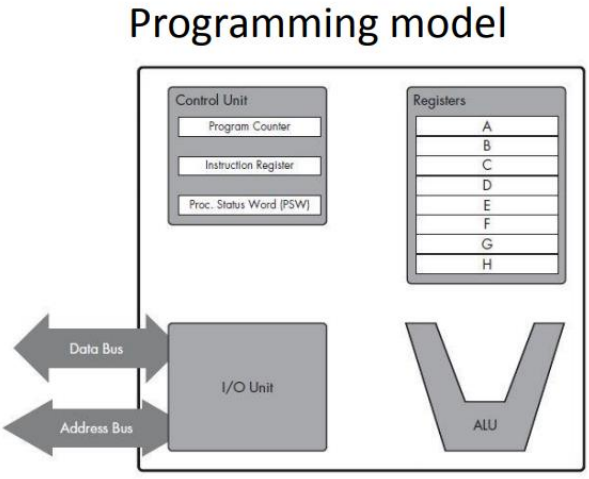
- Immediate-type load
  - Cellule mémoire → Registre
  - Source à 0
- Register-type load
  - Registre → registre
- Register-relative addressed load
  - Register → Offset
  - Basé sur l'adressage relatif

L'instruction store :

- Même principe que pour le load.

L'assembleur :

- Outil qui fait la traduction du langage mnémoniques en langage machine.
- Simplifie l'écriture d'un programme mais ne dispense pas de maitriser les ressources disponibles comme :
  - Nombre de registres
  - Instructions supportées
  - …
- Nécessite d'un modèle qui décrit la machine utilisée : programming model



Utilisation du programming model :

- I/O Unit
  - Interactions avec la mémoire et le reste du système (load, store, …)
- Program counter & instruction register
  - Le program counter indique tout simplement l'adresse de la prochaine instruction
  - L'instruction register contiendra cette instruction. Elle y sera décodée.

La boucle Fetch-Decode-Execute :

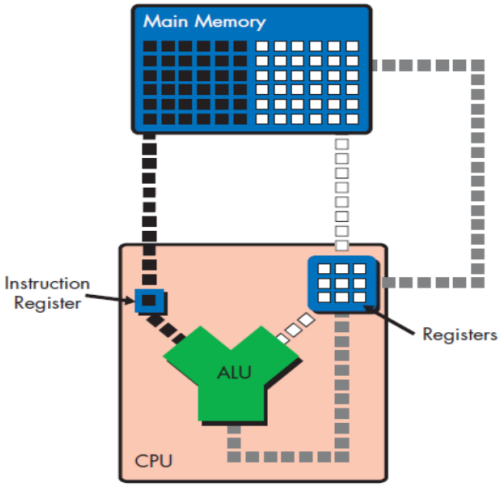
- Fetch → Une instruction load particulière
  - Source = Adresse située dans le program counter
  - Destination = Instruction register
- Decode → L'instruction dans l'instruction register est décodée
- Execute → Puis exécutée.
  - Pendant ce temps, le program counter est incrémenté (L'incrémentation se fait à la fin de l'instruction Fetch)
    - Incrémentation de 2 en 2.

L'horloge :

- Un cycle fetch-decode-execute doit être réalisé complètement sur un battement d'horloge.
- Lorsque que la vitesse d'horloge maximale est atteinte, passage au dualcore.

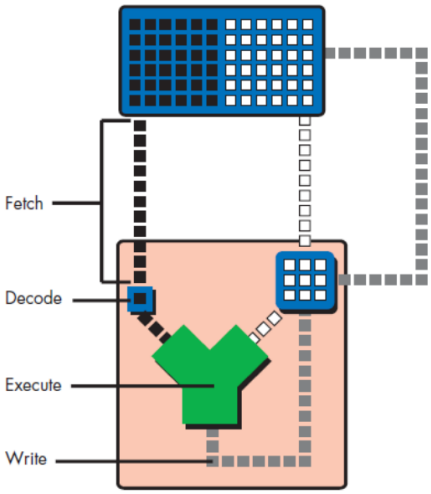
Branch Instructions :

- Unconditional Branch : JUMP #target
  - Permet un saut vers une autre partie du programme (Saut vers une autre cellule mémoire)
  - S'exécute en modifiant l'adresse dans le Program Counter
- Conditional Branch :
  - Permet un saut vers une autre partie du programme si une condition est remplie.
  - La condition peut porter sur le résultat d'une précédente opération.
  - Il faut donc garder une trace de ce résultat
    - Rôle du PSW (Processor Status Word)
    - ex. : Erreur, violation de mémoire, …
  - Mettre le PSW à 1 ou 0 (FLAG)
- Exemples de JUMP (jumpz #106)
  - jumpz → Si résultat = 0, condition remplie
  - jumpn → Si résultat négatif, condition remplie
  - jumpo → Si overflow, condition remplie
- Labels
  - Plus faciles d'utiliser des labels à la place des adresse mémoire
  - ex. :
    - jumpz LBL1
    - LBL1: add A, B, B



III. Pipelined Execution

- Rappel cycle de base
  - Fetch
  - Decode
  - Execute
    - Read
    - Add
    - Write
- Séquence à utiliser
  - Fetch
  - Decode
  - Execute
    - Read
    - Add
  - Write



Program Execution & Completion Rate :

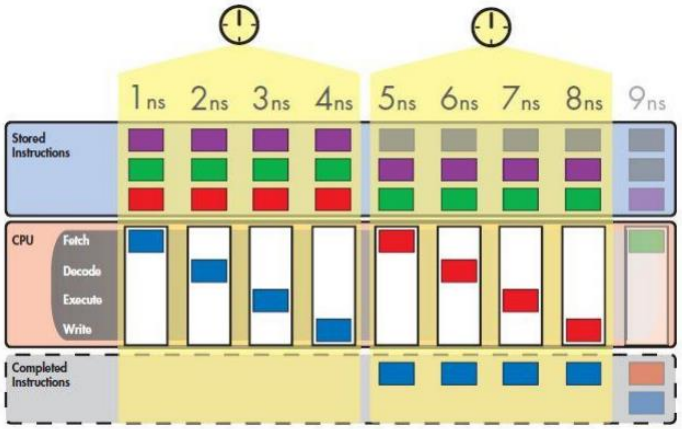
- Program Execution Time (Nanoseconde → 10^{-9} secondes)
  - == Number of Instructions / Instruction Completion Rate
  - Les performances peuvent généralement être établies en observant le Program Execution Time
  - Temps d'exécution pour terminer 1 instruction
- Complettion Rate (Inst/ns) :
  - Nombres d'instructions terminées par unité de temps
- Dans un processeur sans pipeline, le rapport entre les deux est de type inversement proportionnel.
- Le pipeline parvient à augmenter le completion rate sans modifier le temps d'exécution d'une instruction.

Augmenter les performances :

- Le but étant d'augmenter le completion rate.
  - Analogie d'une fabrique de voiture
    - 1er approche : 1 voiture toutes les 5 heures
    - 2ème approche : 1 voiture par heure à partir de la 5ème heure
    - On améliore donc d'un facteur 5

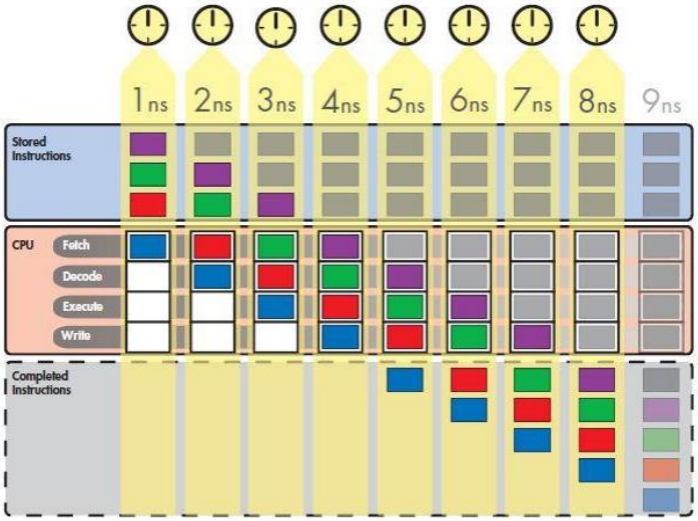
Processeurs single cycle :

- Single cycle = Absence de pipeline
- 1 cycle de vie d'instruction = 1 cycle d'horloge



- 4 phase doivent être exécutées sur 1 pulsation d'horloge
- Zone blanche = Processeur qui ne travaille pas !
  - = Gaspillage de temps et de ressources
- Pour améliorer le completion rate, il faut cadencer l'horloge de + en + vite.

Processeur avec pipeline :



- Profondeur de pipeline de 4 (zone rose)
- Dès le début de la 5ème ns, une instruction est exécutée toutes les ns.
- Les performances passent de 0,25 instr./ns à 1 instr./ns → Un gain de 4

Incidence sur l'horloge :

- L'horloge ne cadence plus une instruction complète mais les phases d'une instruction.
- Pulsations bien + courtes
- Une instruction ne doit plus nécessairement être finie à la fin de chaque cycle d'horloge

Incidence sur la durée d'exécution du programme :

- Dure d'exécution d'une instruction n'a pas été modifié mais la dure d'exécution totale du programme entier (suite d'instructions) a été fortement réduite.
- Réalisé en augmentant le nombre d'instructions terminées par unité de temps.
- Le pipeline rend plus efficace l'utilisation des ressources CPU en faisant travailler simultanément ses différentes sous-unités.

Améliorer encore les performances :

- Augmenter de + en + la profondeur du pipeline.
- A retenir :
  - Le pipeline ne diminue pas la durée d'exécution d'une instruction mais diminué la durée d'exécution du programme en augmentant le nombre d'instructions terminées par unités de temps. (→ Instruction Completion Rate)

En résumé :

- Exécution en pipeline (Chaque pulsation d'horloge = 1 instruction terminée)
  - Temps d'exécution d'une instruction égal voir + grand.
  - Instruction completion rate + grand
  - Program Execution Time + petit
- /!\ Chaque sous-étapes doivent être de durée identique !

ns	0,5	1	1,5	2	2,5	3	3,5	4	4,5	5	5,5	6	6,5	7
0,5 ns FETCH	x	o	*	*	*	*	+	+	+	+	¤	¤	¤	¤
0,5 ns DECODE		x	o	o	o	o	*	*	*	*	+	+	+	+
2 ns EXEC1			x	x	x	x	o	o	o	o	*	*	*	*
0,5 ns EXEC2							x				o			
0,5 ns WRITE								x				o		
Instructions Finies									x				o	

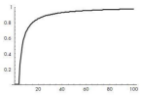
- Temps d'exec d'1 instruction : 4ns
- Instruction completion rate : 1 instruction / 2 ns

Calculs :

- Programme de 10 instructions
  - Temps d'exécution du programme = 4ns + 10ns = 14ns
  - → Instruction completion rate = 10/14 inst/ns = 0,714 instr./ns
  - Temps de charge
  - 1 Instr./ns

Limites du Pipeline :

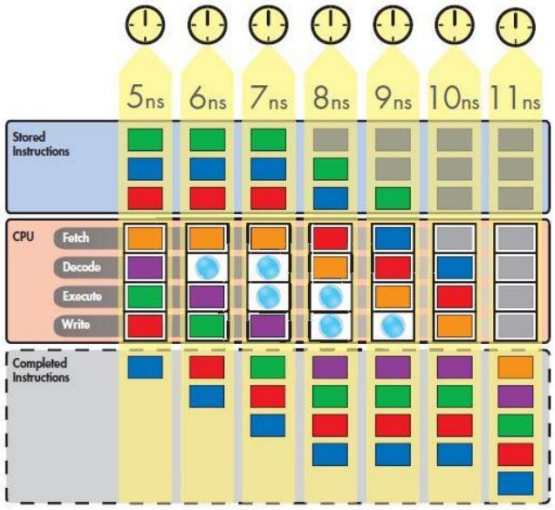
- Si peu d'instructions → Pipeline moins impactant
- Sur 1000 ns, un processeur à pipeline de profondeur 4 aura exécuté 996 instructions au lieu de 250 du processeur sans pipeline.
  - Soit un gain de 3,984 ≈ 4
  - 4 est en réalité un maximum théorique et 3,984 est la moyenne réelle.
- Pour se rapprocher du gain maximal théorique :
  - Augmenter le nbre d'instructions
  - Temps que le programme demande
- Graphiquement

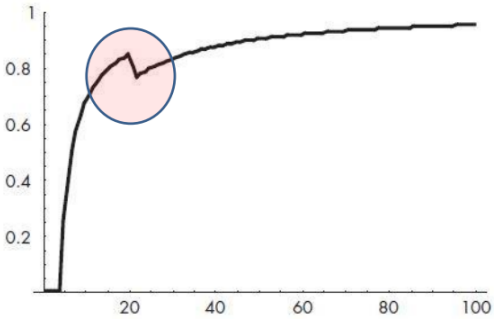


  - Ce n'est qu'après les premières nanoseconde que le pipeline est rempli et commence à sortir des instructions et que la moyenne augmente donc en conséquence
- Dans la pratique, il n'est pas toujours facile de découper une instruction en autant d'étapes équivalentes qu'on le voudrait. (ex. : Etapes indivisibles) (Si étapes indivisibles, cela ne sert à rien de diviser les autres)
- On ne peut généralement pas atteindre l'horloge théorique idéale. (Limitée à une cadence maximale)
- Coût financier du pipeline non négligeable
- Une architecture à pipeline est plus complexe et requière des composants supplémentaires (Ces composants ont évidemment un coût)

Décrochage du pipeline :

- Il peut arriver que des étapes du pipeline nécessitent plus d'un cycle d'horloge.
  - Apparition de « bulles » réduisant les performances
  - La latence d'une instruction peut traduire le temps réel qu'elle met pour traverser le pipeline en prenant en compte les bulles.
- Accès mémoire → Bloque le programme
- L'instruction orange reste bloquée à l'étape fetch pendant 3 cycle d'horloge
- 8ème & 9ème ns → Pas de nouvelles instructions finies.





- Conclusion :
  - Les bulles doivent être évitées
  - Dues à des accès trop lents à la mémoire centrale
  - Solution = Mémoire cache (Entre le CPU et la RAM)

IV. La virtualisation

Introduction :

- La virtualisation est un ensemble de technologies hardware et software permettant de virtualiser du matériel et du logiciel.
- Grands types de virtualisation :
  - Hardware (Processeur, …)
    - Utilisation traditionnelle, lorsque l'OS est installé sur une représentation software des ressources hardware.
  - SDS : Software Defined Storage (Disque dur qui prend une partie de notre HDD)
    - Une couche software est créée entre les HDD réels et l'ordinateur y accédant. Le but étant de les rendre + accessibles.
  - SDN : Software Defined Network (Réseaux (internes, NAT, ..)) (Switch, Router, …)
    - Les utilisateurs peuvent créer une infrastructure réseau (logique) sur un réseau physique. Cela facilite la mise en œuvre des besoins.
- Tendances à s'éloigner du matériel
  - ISA + Microcode
    - ISA : Intermédiaire entre le matériel et le logiciel + traduction des instructions. (Change sa dentition selon le hardware)
- 2 familles de technologies
  - Virtualisation d'OS
  - Virtualisation d'applications (ex. : Word Online, …)
- La virtualisation d'OS est un ensemble de techniques matérielles et logicielles qui permettent de faire fonctionner plusieurs systèmes d'exploitations sur une même machine.

Virtualisation d'applications :

- Isolation
  - OS Commun
  - Chacune des applications a un espace de noms
  - Chacune dans sa bulle
    - Seulement appel à l'OS mais pas aux autres applications
- Container

Avantages :

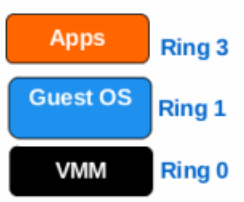
- Consolidation
  - Meilleure utilisation du CPU (un serveur utiliser typiquement 5-10%)
  - Economie de surface et d'électricité en datacenter
- Load balancing
  - Permet d'équilibrer l'utilisation des serveur (A chaud)
- Tolérance de panne
  - Les VMs peuvent être transférées (en direct) vers d'autres serveurs.
- Isolation
  - Utilisateur demandant beaucoup de ressources
  - Virus
  - Raison de perfs / Sécurité
- Déboguer de programmes (+ rapide de redémarrer une VM)
- DevOps
- Compatibilité avec d'anciens OS
- Virtualisation hétérogène
- Fini les problèmes hardware
- HA, Migration, Clonage, …

Contraintes :

- Gestion par un admin
- Surcharge du CPU

Modèle de Pile hard :

- OS au dessus du matériel
- Logiciels au dessus de l'OS
- L'OS
  - A accès au matériel
  - Donne un accès sélectif aux ressources
- Rôle de l'OS
  - Isoler les processus
  - Gérer le partage de ressources entre les processus
- → OS doit avoir un accès complet au matériel
- Virtualisation : Faire croire que l'OS a cet accès.



Notions :

- VMM → Virtual Machine Monitor
  - Doit faire croire que les OS ont un accès aux ressources
- ISA x86
  - Peut coopératif avec la VMM

Méthode de virtualisation :

Applications Guest OS VMM OS Hôte HW	Applications Guest OS VMM HW

Technologies de virtualisation :

- Full Virtualisation
  - Simule le matériel
  - Limiter aux OS prévus pour la même architecture que le processeur physique
- Emulation
  - Simuler du matériel spécifique (Processeur, …)
  - Performances ↓↓
- Paravirtualisation (PVM)
  - Guest OS sait qu'il est sur un VM
  - Devient capable d'intéragir avec l'hyperviseur
  - Performances ↑↑