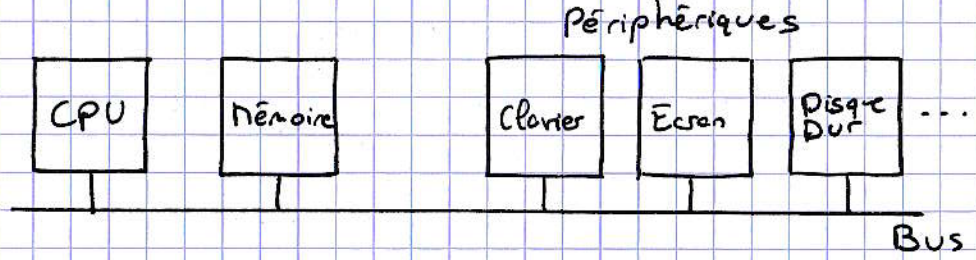
Principe de fonctionnement des ordinateurs : architecture, notions d'assembleur

Introduction : important de comprendre l'interface entre logiciel et matériel. Sécurité, parallélisme...

I/ Vue d'ensemble



On identifie 5 composants :
- entrée
- sortie
- mémoire : stocke les données sur lesquelles on travaille
- CPU { - chemin de données : effectue une opération
  - contrôle : détermine les actions à effectuer par les autres composants

Programme : traitement à appliquer à des données
Données → binaire → mémoire
Traitement → Instructions (langage machine) → mémoire
⇒ Pour chaque instruction :
- récupérer l'instruction en mémoire
- la décoder, récupérer les opérandes
- exécuter l'instruction

NB : on appelle PC (Program Counter) l'adresse de l'instruction à exécuter

Langage machine = représentation binaire des instructions du programme
Langage haut niveau → langage assembleur → langage machine

II/ Rendre la machine programmable : le langage assembleur

ISA ("Instruction Set Architecture") = ensemble d'instructions assembleur + traduction binaire, interface entre le matériel et le logiciel bas niveau. Dépend du processeur.
Ex : RISC-V, Mips, x86...

1) Opérandes possibles

Registres : espaces de stockage, en nombre limité, qui font partie du processeur. RISC-V : 32 registres de 64 bits chacun.

Mémoire : "tableau" de données, accessible via adresse
RISC-V : manipulable uniquement via les instructions load et store

Immédiate : constante, en dur dans l'instruction

2) Opérations arithmétiques et logiques

Format : reg_d, reg_{s_1}, reg_{s_2} ou reg_d, reg_s, imm

Supposons que variables a → registre s0, b → s1, c → s2, d → s3, e → s4

a = b + c → add s0, s1, s2
d = a - e → sub s3, s0, s4
a = (b+c) - (d+e) → add t0, s1, s2
add t1, s3, s4
sub s0, t0, t1
a = b + 4 → addi s0, s1, 4

Même principe pour les opérations logiques : and, andi, or, ori, xor, xori, ...

3) Transfert de données

Transfert mémoire → registre (load) ou registre → mémoire (store)
Format : reg, imm(reg)
↓ ↓ ↓
dest ou src décalage base de l'adresse

adresse = reg + imm

Supposons que variable a → reg s0, adresse du tableau T → s1

```
a = A[0]  -> ld s0, 0(s1)
A[4] = a  -> sd s0, 32(s1)
A[12] = a+A[8] -> ld t0, 64(s1)
                  add t0, s0, t0
                  sd t0, 96(s1)
```

4) Branchement conditionnel

Format : reg_1, reg_2, imm
Si la condition (indiquée par l'opcode) est vraie pour reg_1 et reg_2, la prochaine instruction sera à PC+imm. Sinon, PC+4.
En pratique, on représente les cibles avec des labels.

```
if (i != j)      -> : beq s0, s1, Exit
  a = b+c;            add s2, s3, s4
                      Exit:
```

5) Fonctions

On a besoin de :
- passer des paramètres, et retourner des valeurs

RISC-V : registres x10 à x17, par convention
- sauter vers le code de la fonction, et revenir

RISCV : jal x1, imm → stocke l'adresse de la prochaine instruction en x1 et branche vers imm
jalr x0, 0(x1) → stocke l'adresse et branche vers x1 (+0)

Dev 1 : analyse de code assembleur

6) Représentation binaire

Le format d'instruction, propre à l'ISA, définit la représentation en langage machine de l'instruction assembleur.
RISC-V : instructions sur 32 bits, plusieurs formats
L'instruction contient deux informations :
- l'opcode permet d'identifier l'instruction
- les opérandes sont les paramètres

Ex: R-type	funct7	rs2	rs1	funct3	rd	opcode
	7	5	5	3	5	7

add s20, s21, s22
↳ 0000000 | 10110 | 10101 | 000 | 10100 | 0110011

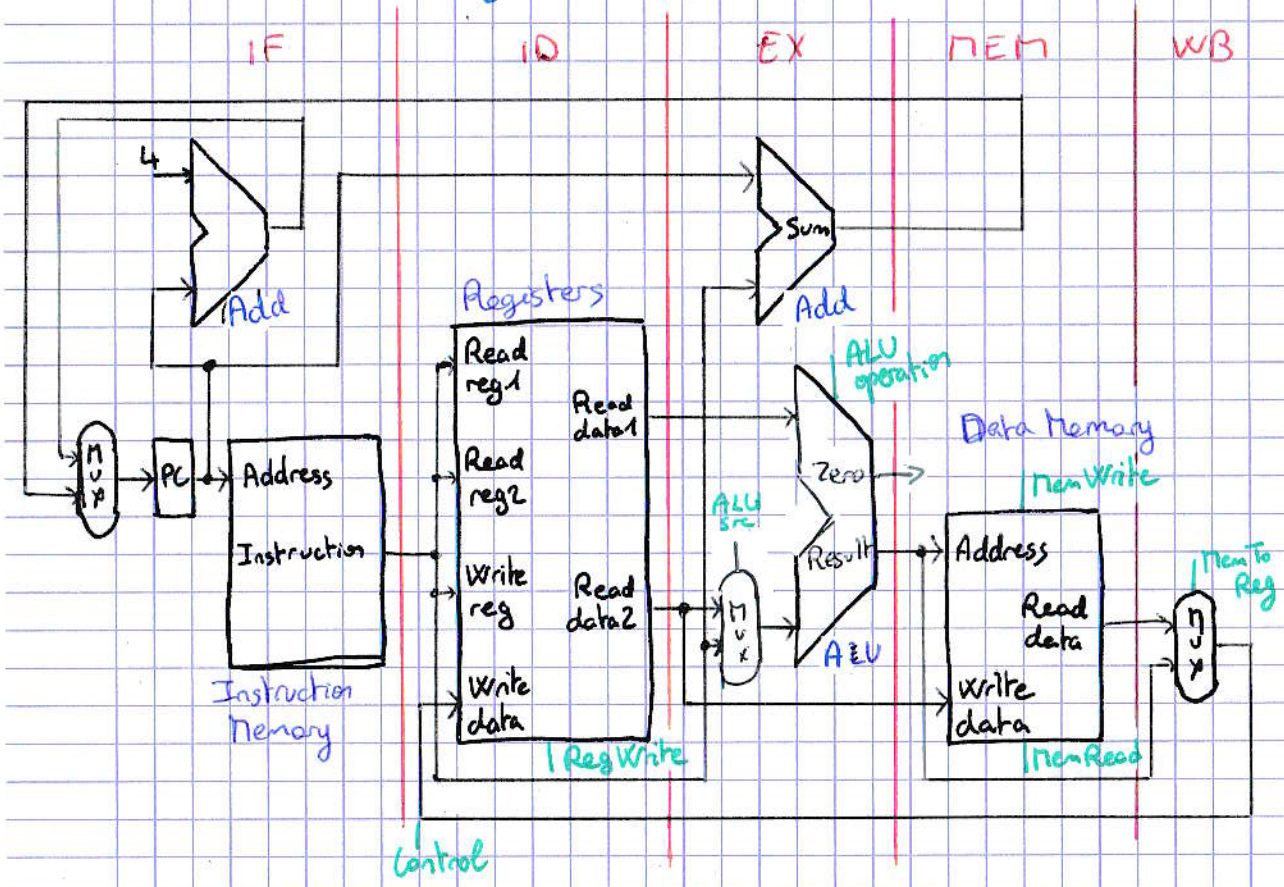
I-type	immediate	rs1	funct3	rd	opcode
	12	5	3	5	7

addi s21, s20, 42
↳ 0101010 | 10100 | 000 | 10101 | 0010011

III / Architecture du processeur : chemin de données

Illustration d'une architecture pour un sous-ensemble de RISC-V : load, store, add, sub, and, or, beq

Pour chaque instruction :
1. IF. Récupérer l'instruction, stockée en mémoire à l'adresse PC
2. ID. Décoder l'instruction et récupérer les opérandes en registre
3. EX. Effectuer calcul avec ALU
4. MEM. Accéder à la mémoire (éventuellement)
5. WB. Écrire en registre (éventuellement)

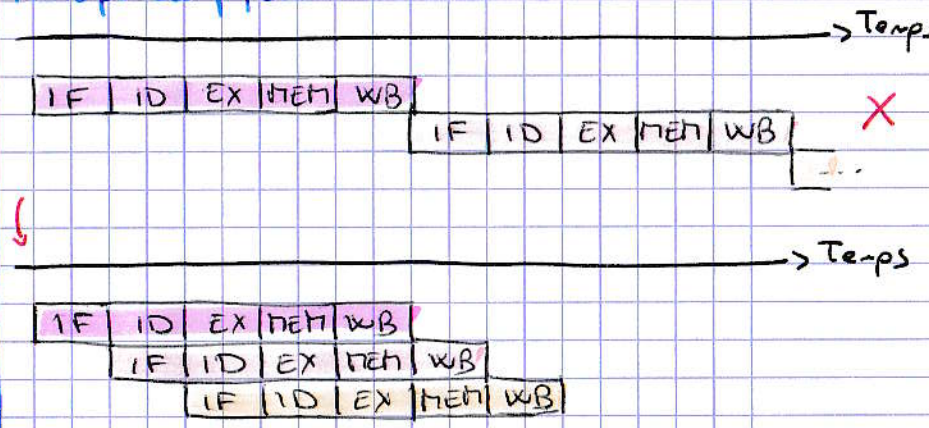


Dev 2 : construction progressive du ~~schém~~chemin de données en fonction des instructions

1) Parallélisme d'instruction : pipeline

Pour chaque instruction, il y a (ici) 5 étapes, correspondant à 5 parties du processeur.
NB : on aurait pu diviser en 3, ou en 10.

Principe du pipeline



Problèmes possibles :
- structurel : si les étapes ne sont pas indépendantes
- données : si on a besoin d'une donnée calculée plus-dessus
- contrôle : si on a besoin d'une instruction précédente pour connaître la prochaine instruction

Ouverture : Problèmes de sécurité (débordement de tampon, spectre...)
Autres types de parallélisme