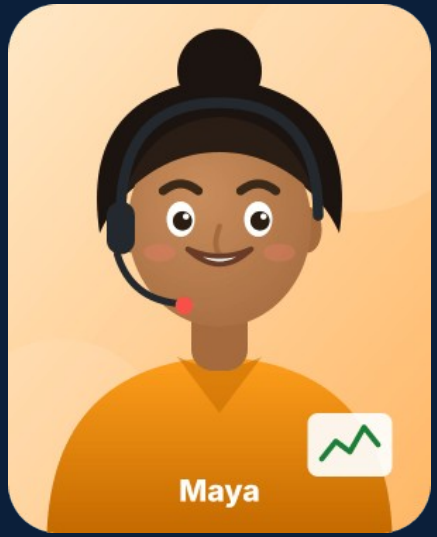
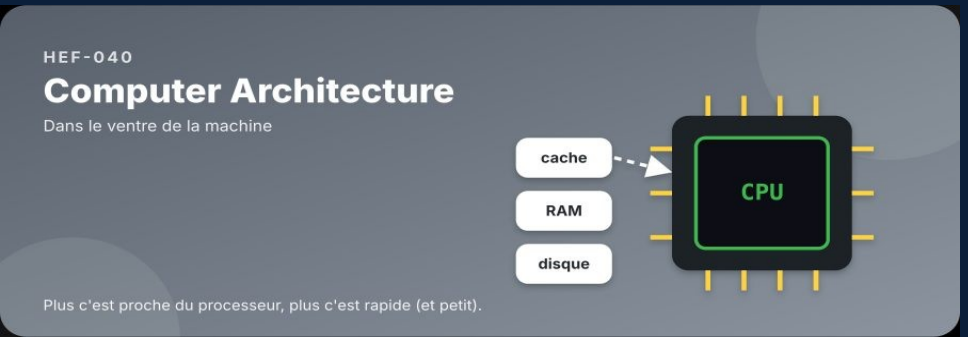
HEF-040 · MISSION 02 / 10

Le CPU et son langage : le jeu d'instructions

Un processeur sait faire très peu de choses. Mais il les fait des milliards de fois par seconde.



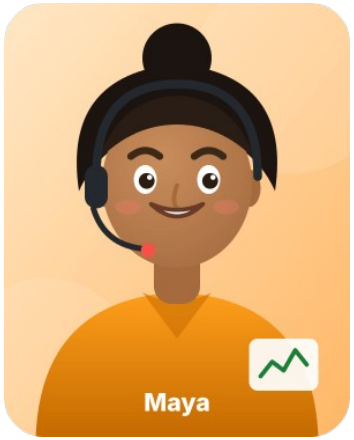


Badge : Parleur de processeur

75 min · débutant

L'histoire

Kodi demande à Sam : « Quand j'écris `a = b + c`, que fait vraiment le processeur ? »



> *« Il ne comprend pas Python, dit Sam. Il comprend une liste très courte d'ordres simples : charger, additionner, ranger. Ton programme finit par devenir une longue suite de ces ordres. »*

Tu découvres le vocabulaire du processeur et tu écris ton premier programme dans un langage machine miniature.

Ta mission

À la fin, je serai capable de…

1. définir CPU, instruction, opcode, opérande
2. expliquer ce qu'est un jeu d'instructions (ISA)
3. distinguer langage machine et assembleur
4. écrire un petit programme dans un jeu d'instructions simplifié
5. expliquer comment une opération de haut niveau se décompose en instructions

Défi d'échauffement

Tu dois expliquer à un robot très obéissant, mais qui ne comprend que **cinq ordres** (prendre, poser, avancer, tourner, attendre), comment préparer un sandwich. Écris la liste d'ordres. Que remarques-tu sur la longueur de la liste par rapport à une explication à un humain ?

Le concept

Le CPU

Le **processeur** (CPU, *Central Processing Unit*) exécute des **instructions**. Il en fait des milliards par seconde (une fréquence de 3 GHz = 3 milliards de cycles par seconde).

L'instruction

Une **instruction** est un ordre élémentaire codé en binaire. Elle contient :

- un **opcode** (code d'opération) : *quoi faire* ;
- des **opérandes** : *sur quoi* (registres, valeur immédiate, adresse mémoire).

Le concept

L'instruction (suite)

```
ADD R1, R2, R3      ; R1 ← R2 + R3
LOAD R1, [100]      ; R1 ← valeur à l'adresse mémoire 100
STORE R1, [104]     ; mémoire[104] ← R1
JMP 20              ; saute à l'instruction numéro 20
```

Le jeu d'instructions (ISA)

Un **ISA** (*Instruction Set Architecture*) est la liste des instructions qu'un processeur comprend, avec leur codage. C'est le **contrat** entre le matériel et le logiciel.

Famille	Exemples	Idée
x86-64	PC, serveurs Intel/AMD	riche, nombreuses instructions
ARM	téléphones, Mac récents	plus simple, économe en énergie
RISC-V	open source	simple et modulaire

Le concept

Le jeu d'instructions (ISA) (suite)

Un programme compilé pour x86-64 ne tourne pas tel quel sur ARM : ce sont deux « langues machine » différentes.

Langage machine et assembleur

- **Langage machine** : les instructions en binaire (ce que le CPU lit).
- **Assembleur** : une écriture lisible pour un humain (`ADD`, `LOAD`…), traduite 1 pour 1 en langage machine.

Les grandes familles d'instructions

Famille	Exemples
Arithmétique et logique	ADD, SUB, MUL, AND, OR, XOR
Mémoire	LOAD, STORE

Le concept

Les grandes familles d'instructions (suite)

Famille	Exemples
Contrôle	JMP (saut), BEQ (saut si égal), CALL, RET
Comparaison	CMP

De Python au processeur

`a = b + c` devient, en gros :

```
LOAD R1, [adresse_de_b]
LOAD R2, [adresse_de_c]
ADD  R3, R1, R2
STORE R3, [adresse_de_a]
```

Une boucle `for` devient une comparaison suivie d'un saut conditionnel qui revient en arrière.

Exemple pas à pas

Problème : calculer 3 × 4 en n'ayant que l'addition, avec une mini-ISA.

Instructions de notre mini-machine :

Instruction	Effet
`SET Rx, n`	Rx ← n
`ADD Rx, Ry`	Rx ← Rx + Ry
`DEC Rx`	Rx ← Rx − 1
`JNZ Rx, adr`	si Rx ≠ 0, saute à l'instruction `adr`
`HALT`	arrête

Programme : `R0 = 0`, `R1 = 3` (compteur), `R2 = 4`. Tant que R1 ≠ 0 : `R0 += R2`, `R1 -= 1`.

Exemple pas à pas (suite)

```
def executer(programme):
    R = {"R0": 0, "R1": 0, "R2": 0, "R3": 0}
    pc = 0
    while True:
        instr = programme[pc]
        op = instr[0]
        if op == "SET":
            R[instr[1]] = instr[2]
        elif op == "ADD":
            R[instr[1]] += R[instr[2]]
        elif op == "DEC":
            R[instr[1]] -= 1
        elif op == "JNZ":
            if R[instr[1]] != 0:
                pc = instr[2]
                continue
        elif op == "HALT":
            return R
        pc += 1
```

Exemple pas à pas (suite)

```
programme = [
    ("SET", "R0", 0),
    ("SET", "R1", 3),
    ("SET", "R2", 4),
    ("ADD", "R0", "R2"),     # adresse 3
    ("DEC", "R1"),
    ("JNZ", "R1", 3),
    ("HALT",),
]
print(executer(programme)["R0"])   # 12
```

Ce qu'on voit : la multiplication n'existe pas dans cette machine, elle est **construite** avec une boucle. Un vrai CPU a une instruction `MUL` câblée, mais le principe est le même : tout programme se ramène à de petites instructions.

À toi, avec aide

Avec la mini-machine de l'exemple, écris un programme qui calcule la **somme 1 + 2 + 3 + 4 + 5** (résultat 15 dans R0).

Indice d'algorithme : un compteur R1 démarre à 5 ; on ajoute R1 à R0 puis on décrémente R1 ; tant que R1 ≠ 0, on recommence.

À toi, avec aide · réponse

Indices progressifs

Indice 1

R0 reçoit la somme, R1 est le compteur initialisé à 5.

Indice 2

Dans la boucle : `ADD R0, R1` puis `DEC R1`.

Indice 3

Termine avec `JNZ R1, adresse` pointant sur le début de la boucle.

À toi, avec aide · réponse

Réponse

```
def executer(programme):
    R = {"R0": 0, "R1": 0, "R2": 0, "R3": 0}
    pc = 0
    while True:
        instr = programme[pc]
        op = instr[0]
        if op == "SET":
            R[instr[1]] = instr[2]
        elif op == "ADD":
            R[instr[1]] += R[instr[2]]
        elif op == "DEC":
            R[instr[1]] -= 1
        elif op == "JNZ":
            if R[instr[1]] != 0:
                pc = instr[2]
                continue
        elif op == "HALT":
            return R
        pc += 1
```

À toi, avec aide · réponse

Réponse (suite)

```
programme = [
    ("SET", "R0", 0),
    ("SET", "R1", 5),
    ("ADD", "R0", "R1"),    # adresse 2
    ("DEC", "R1"),
    ("JNZ", "R1", 2),
    ("HALT",),
]
print(executer(programme)["R0"])   # 15
```

À toi, en solo

Étends la mini-machine avec quatre nouvelles instructions : `SUB Rx, Ry` (Rx ← Rx − Ry), `MOV Rx, Ry` (Rx ← Ry), `INC Rx` (Rx ← Rx + 1) et `JNEG Rx, adr` (saute à `adr` si Rx < 0).

Puis écris un programme qui calcule la **division entière** de R1 (dividende) par R2 (diviseur), avec le quotient dans R0 et le reste dans R1. Teste avec 17 ÷ 5 (quotient 3, reste 2).

Pistes de correction

```
def executer(programme, init=None):
    R = {"R0": 0, "R1": 0, "R2": 0, "R3": 0}
    R.update(init or {})
    pc = 0
    for _ in range(10000):
        instr = programme[pc]
        op = instr[0]
        if op == "SET": R[instr[1]] = instr[2]
        elif op == "ADD": R[instr[1]] += R[instr[2]]
        elif op == "SUB": R[instr[1]] -= R[instr[2]]
        elif op == "MOV": R[instr[1]] = R[instr[2]]
        elif op == "DEC": R[instr[1]] -= 1
        elif op == "INC": R[instr[1]] += 1
        elif op == "JNZ":
            if R[instr[1]] != 0:
                pc = instr[2]; continue
        elif op == "JNEG":
            if R[instr[1]] < 0:
                pc = instr[2]; continue
```

Pistes de correction (suite)

```
        elif op == "HALT":
            return R
        pc += 1
    raise RuntimeError("trop d'instructions")

# R3 = R1 - R2 ; si R3 < 0 : fin ; sinon R1 = R3, R0++ et on recommence
programme = [
    ("SET", "R0", 0),         # 0
    ("MOV", "R3", "R1"),      # 1
    ("SUB", "R3", "R2"),      # 2
    ("JNEG", "R3", 7),        # 3 : reste < diviseur -> fin
    ("MOV", "R1", "R3"),      # 4
    ("INC", "R0"),            # 5
    ("JNZ", "R2", 1),         # 6 : saut inconditionnel (R2 != 0)
    ("HALT",),                # 7
]
r = executer(programme, {"R1": 17, "R2": 5})
print(r["R0"], r["R1"])   # 3 2
```

Pistes de correction (suite)

Les ISA réels offrent des instructions de comparaison et de saut encore plus riches, qui raccourcissent ce genre de programme.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Croire que le CPU « comprend » Python	Il exécute des instructions simples	Penser en termes d'instructions
Confondre assembleur et langage machine	Même chose, écrite différemment	Assembleur = lisible, machine = binaire
Boucle sans condition de sortie	Le programme ne s'arrête jamais	Vérifier le compteur et le saut
Oublier que l'ISA est un contrat	Un binaire x86 ne tourne pas sur ARM	Compiler pour la bonne cible

Mini-boss

Mini-boss

Ton processeur miniature. Étends le simulateur pour qu'il ait :

- une **mémoire** (liste de 32 cases) avec `LOAD Rx, adr` et `STORE Rx, adr` ;
- des instructions arithmétiques `ADD`, `SUB`, `MUL` ;
- un saut conditionnel `JNZ` et inconditionnel `JMP`.

Écris en assembleur miniature un programme qui calcule la **factorielle de 5** (120) et la range en mémoire à l'adresse 0.

Compte aussi combien d'instructions ont été exécutées. Combien en faudrait-il pour factorielle de 10 ?

Mini-boss · réponse

Pistes de correction

```
def executer(programme, regs=None):
    R = {f"R{i}": 0 for i in range(4)}
    R.update(regs or {})
    mem = [0] * 32
    pc = compte = 0
    while True:
        instr = programme[pc]
        compte += 1
        op = instr[0]
        if op == "SET": R[instr[1]] = instr[2]
        elif op == "ADD": R[instr[1]] += R[instr[2]]
        elif op == "SUB": R[instr[1]] -= R[instr[2]]
        elif op == "MUL": R[instr[1]] *= R[instr[2]]
        elif op == "DEC": R[instr[1]] -= 1
        elif op == "LOAD": R[instr[1]] = mem[instr[2]]
        elif op == "STORE": mem[instr[2]] = R[instr[1]]
        elif op == "JMP": pc = instr[1]; continue
        elif op == "JNZ":
            if R[instr[1]] != 0:
```

Pistes de correction (suite)

```
                pc = instr[2]; continue
        elif op == "HALT":
            return mem, compte
        pc += 1

def fact(n):
    prog = [
        ("SET", "R0", 1),        # résultat
        ("SET", "R1", n),        # compteur
        ("MUL", "R0", "R1"),     # 2 : R0 *= R1
        ("DEC", "R1"),
        ("JNZ", "R1", 2),
        ("STORE", "R0", 0),
        ("HALT",),
    ]
    return executer(prog)

mem5, c5 = fact(5)
mem10, c10 = fact(10)
```

Pistes de correction (suite)

```
print(mem5[0], c5, mem10[0], c10)   # 120 19 3628800 34
```

Le nombre d'instructions exécutées croît linéairement avec n : 3 par tour de boucle, plus 4 fixes (deux initialisations, le STORE et le HALT), soit 19 pour n = 5 et 34 pour n = 10.

Quiz éclair

Quiz éclair

1. Un opcode indique :

A. où stocker le résultat
B. la vitesse du CPU
C. quelle opération effectuer

2. Un ISA est :

A. la liste des instructions qu'un processeur comprend
B. un type de mémoire
C. un langage de haut niveau

3. L'assembleur est :

A. un langage de haut niveau

Quiz éclair

Quiz éclair (suite)

B. une écriture lisible du langage machine
C. un type de processeur

4. 🔁 *Révision mission 01* : Un octet peut prendre :

A. 8 valeurs
B. 1 024 valeurs
C. 256 valeurs

5. 🔁 *Révision mission 01* : Un chiffre hexadécimal représente :

A. 4 bits
B. 8 bits
C. 2 bits

Quiz éclair · réponse

Réponses

1. **C**.
2. **A**.
3. **B**.
4. **C**.
5. **A**.

Je réfléchis

1. Qu'est-ce qui te surprend le plus dans le fait qu'un CPU n'ait que quelques dizaines d'instructions ?
2. Pourquoi un même programme doit-il être recompilé pour une architecture différente ?

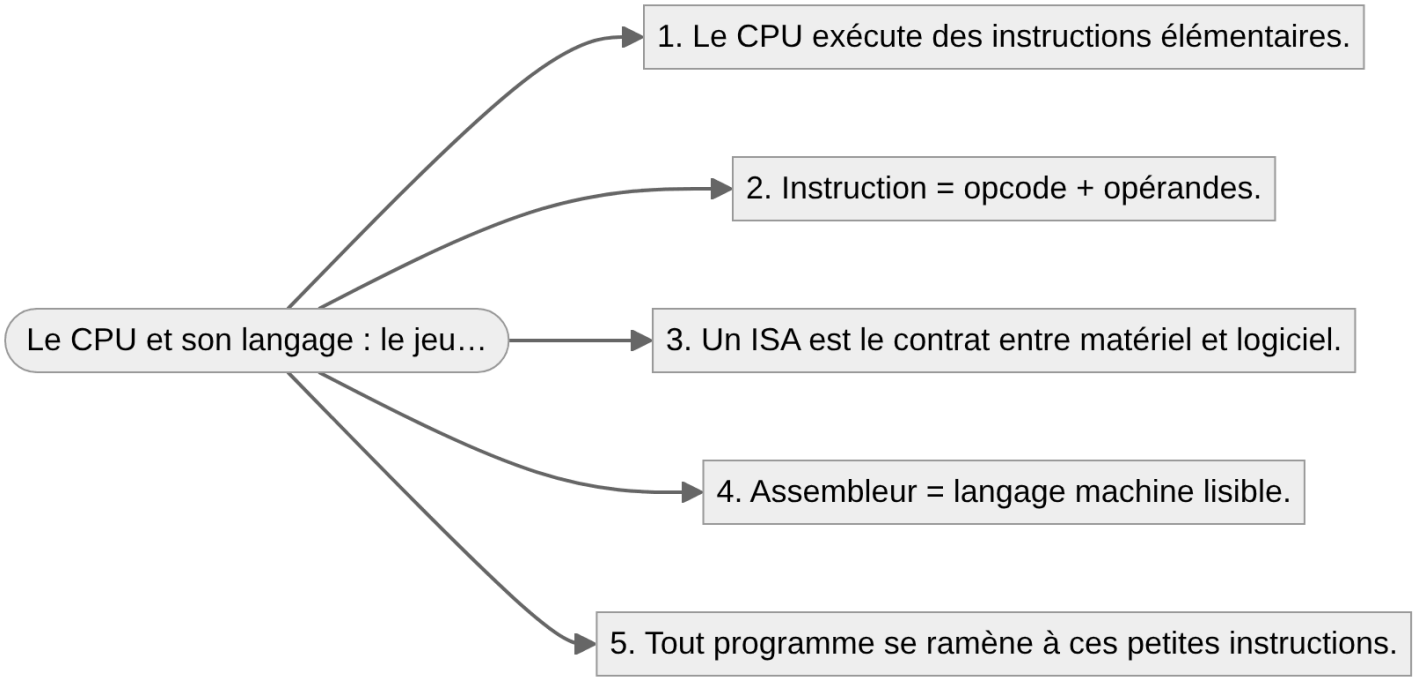
À retenir

À retenir

1. Le CPU exécute des instructions élémentaires.
2. Instruction = opcode + opérandes.
3. Un ISA est le contrat entre matériel et logiciel.
4. Assembleur = langage machine lisible.
5. Tout programme se ramène à ces petites instructions.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Parleur de processeur

Prochaine mission : Registres et cycle fetch-decode-execute

