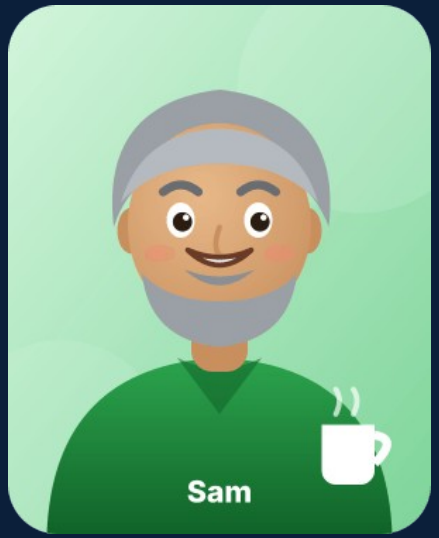
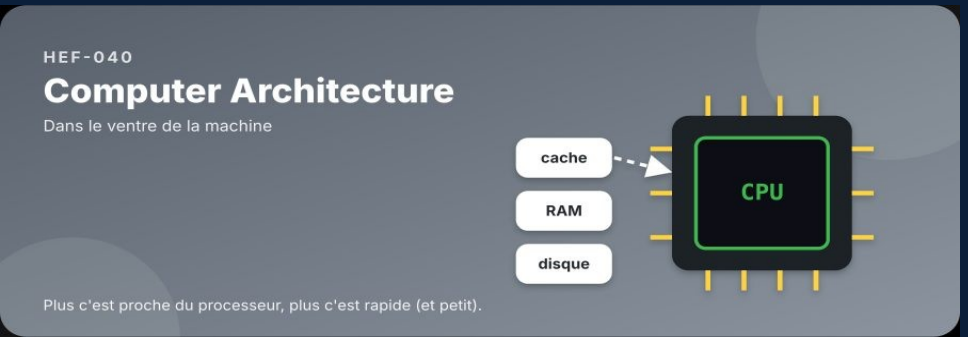
HEF-040 · MISSION 03 / 10

Registres et cycle fetch-decode-execute

Le CPU répète sans cesse trois gestes : chercher une instruction, la comprendre, l'exécuter.





Badge : Horloger du CPU

75 min · débutant

L'histoire

Kodi a écrit un petit simulateur d'instructions, mais il ne comprend pas comment le vrai processeur « sait » quelle instruction exécuter ensuite.



> *« Il y a un petit compteur qui pointe l'instruction suivante, dit Sam. À chaque tour : il va la chercher, la décode, l'exécute, puis avance le compteur. Toujours dans le même ordre. »*

Tu apprends le cycle qui fait battre le cœur de tous les ordinateurs.

Ta mission

À la fin, je serai capable de…

1. décrire le cycle fetch-decode-execute
2. expliquer le rôle du compteur de programme (PC) et des registres
3. expliquer pourquoi les registres sont le stockage le plus rapide
4. simuler un cycle complet avec un programme codé en mémoire
5. expliquer ce qu'est un saut et comment il change le flux d'exécution

Défi d'échauffement

Défi d'échauffement

Tu suis une recette de cuisine avec ton doigt posé sur la ligne en cours. À chaque étape : tu lis la ligne, tu comprends l'action, tu la fais, puis tu déplaces ton doigt à la ligne suivante. Que se passe-t-il si la recette dit « retourne à l'étape 3 » ? Quel élément de la recette joue le rôle du doigt ?

Le concept

Les registres

Les **registres** sont de minuscules mémoires **à l'intérieur du CPU**. Ils sont les plus rapides de tout l'ordinateur (accès en un cycle ou moins). Un CPU en a quelques dizaines (par exemple 16 registres de 64 bits en x86-64).

Registres particuliers :

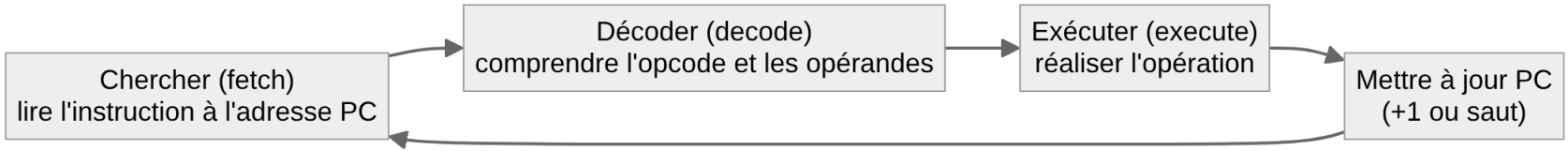
Registre	Rôle
PC (*Program Counter*)	adresse de la prochaine instruction
IR (*Instruction Register*)	instruction en cours
SP (*Stack Pointer*)	sommet de la pile d'appels
Registres généraux	valeurs sur lesquelles on calcule

Le concept

Les registres (suite)

Registre	Rôle
Registre d'état (flags)	résultat de la dernière comparaison (zéro, négatif, retenue…)

Le cycle fetch-decode-execute



Description du schéma : une boucle de quatre étapes : chercher l'instruction dont l'adresse est dans PC, la décoder, l'exécuter, puis mettre à jour PC, et recommencer.

Le concept

Le programme est dans la mémoire

Instructions **et** données sont stockées dans la même mémoire (architecture dite de **von Neumann**). Le CPU lit l'instruction comme il lirait n'importe quelle donnée.
Conséquence : un programme peut en produire un autre (compilateurs), mais aussi un attaquant peut injecter du code si la mémoire n'est pas protégée.

Les sauts

Un saut remplace la valeur de PC par une nouvelle adresse. C'est ce qui permet les **boucles**, les **conditions** et les **appels de fonction**.

Le concept

Un simulateur avec mémoire

```
# Mémoire : liste d'entiers. Instruction codée en 3 cases : [opcode, a, b]
SET, ADD, DEC, JNZ, HALT = 1, 2, 3, 4, 0

def executer(memoire):
    R = [0, 0, 0, 0]
    pc = 0
    cycles = 0
    while True:
        opcode, a, b = memoire[pc], memoire[pc + 1], memoire[pc + 2]   # FETCH
        cycles += 1
        pc += 3                                                       # PC avance
        if opcode == SET:    R[a] = b                                  # DECODE +
   EXECUTE
        elif opcode == ADD:  R[a] += R[b]
        elif opcode == DEC:  R[a] -= 1
        elif opcode == JNZ:
```

Le concept

Un simulateur avec mémoire (suite)

```
            if R[a] != 0:
                pc = b
        elif opcode == HALT:
            return R, cycles
```

🧠 ***Astuce*** *: un registre est comme une main : tu y tiens ce dont tu as besoin tout de suite. La mémoire est l'étagère, plus lente à atteindre.*

Exemple pas à pas

Exemple pas à pas

Problème : exécuter 4 + 3 + 2 + 1 avec le cycle, en observant PC et les registres.

```
SET, ADD, DEC, JNZ, HALT = 1, 2, 3, 4, 0

def executer(memoire, trace=False):
    R = [0, 0, 0, 0]
    pc = cycles = 0
    while True:
        opcode, a, b = memoire[pc], memoire[pc + 1], memoire[pc + 2]
        cycles += 1
        if trace:
            print(f"cycle {cycles:2d}  PC={pc:2d}  opcode={opcode}  R={R}")
        pc += 3
        if opcode == SET:    R[a] = b
        elif opcode == ADD:  R[a] += R[b]
        elif opcode == DEC:  R[a] -= 1
        elif opcode == JNZ:
            if R[a] != 0:
```

Exemple pas à pas

Exemple pas à pas (suite)

```
                pc = b
        elif opcode == HALT:
            return R, cycles

programme = [
    SET, 0, 0,        # PC=0  R0 = 0
    SET, 1, 4,        # PC=3  R1 = 4
    ADD, 0, 1,        # PC=6  R0 += R1
    DEC, 1, 0,        # PC=9  R1 -= 1
    JNZ, 1, 6,        # PC=12 si R1 != 0 : retour à PC=6
    HALT, 0, 0,       # PC=15
]
R, cycles = executer(programme, trace=True)
print(R[0], cycles)   # 10 et le nombre de cycles
```

Lecture de la trace : on voit PC avancer de 3 en 3, puis **revenir à 6** à chaque `JNZ` tant que R1 ≠ 0. Le comportement de la boucle vient uniquement de la valeur de PC.

À toi, avec aide

Dans le simulateur de l'exemple, modifie le programme pour calculer **5 + 4 + 3 + 2 + 1**. Prévois à la main : combien de cycles seront exécutés ? Vérifie avec le programme.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Change la valeur initiale de R1 dans la deuxième instruction.

Indice 2

Il y a 2 instructions de départ, 3 par tour (ADD, DEC, JNZ), et HALT.

Indice 3

Nombre de cycles = 2 + 3 × 5 + 1.

À toi, avec aide · réponse

Réponse

```
SET, ADD, DEC, JNZ, HALT = 1, 2, 3, 4, 0

def executer(memoire):
    R = [0, 0, 0, 0]
    pc = cycles = 0
    while True:
        opcode, a, b = memoire[pc], memoire[pc + 1], memoire[pc + 2]
        cycles += 1
        pc += 3
        if opcode == SET:    R[a] = b
        elif opcode == ADD:  R[a] += R[b]
        elif opcode == DEC:  R[a] -= 1
        elif opcode == JNZ:
            if R[a] != 0:
                pc = b
        elif opcode == HALT:
            return R, cycles

programme = [SET, 0, 0, SET, 1, 5, ADD, 0, 1, DEC, 1, 0, JNZ, 1, 6, HALT, 0, 0]
```

À toi, avec aide · réponse

Réponse (suite)

```
R, cycles = executer(programme)
print(R[0], cycles)   # 15 18
```

Prévision : 2 + 3 × 5 + 1 = 18 cycles.

À toi, en solo

Ajoute au simulateur :

1. une instruction `LOAD Rx, adr` et `STORE Rx, adr` sur une zone de données de la mémoire (par exemple à partir de l'adresse 100) ;
2. un registre de comptage `cycles` qui affiche aussi le **nombre d'accès à la mémoire** (chaque FETCH, LOAD, STORE compte).

Écris un programme qui range 3, 2, 1 aux adresses 100, 101, 102, puis les relit et les additionne. Que remarques-tu sur le rapport entre le nombre de cycles et le nombre d'accès mémoire ?

À toi, en solo · réponse

Pistes de correction

```
SET, ADD, DEC, JNZ, HALT, LOAD, STORE = 1, 2, 3, 4, 0, 5, 6

def executer(memoire):
    R = [0, 0, 0, 0]
    pc = cycles = acces = 0
    while True:
        opcode, a, b = memoire[pc], memoire[pc + 1], memoire[pc + 2]
        cycles += 1
        acces += 1                          # FETCH
        pc += 3
        if opcode == SET:    R[a] = b
        elif opcode == ADD:  R[a] += R[b]
        elif opcode == DEC:  R[a] -= 1
        elif opcode == LOAD: R[a] = memoire[b]; acces += 1
        elif opcode == STORE: memoire[b] = R[a]; acces += 1
        elif opcode == JNZ:
            if R[a] != 0:
                pc = b
        elif opcode == HALT:
```

Pistes de correction (suite)

```
            return R, cycles, acces

mem = [0] * 200
prog = [
    SET, 0, 3, STORE, 0, 100,
    SET, 0, 2, STORE, 0, 101,
    SET, 0, 1, STORE, 0, 102,
    SET, 0, 0,
    LOAD, 1, 100, ADD, 0, 1,
    LOAD, 1, 101, ADD, 0, 1,
    LOAD, 1, 102, ADD, 0, 1,
    HALT, 0, 0,
]
mem[:len(prog)] = prog
R, c, a = executer(mem)
print(R[0], c, a)    # 6 et cycles / accès
```

Pistes de correction (suite)

Les opérations sur les registres ne coûtent qu'un accès (le fetch) ; chaque LOAD/STORE en ajoute un. La mémoire est donc sollicitée beaucoup plus que les registres, d'où l'importance du cache (missions suivantes).

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Oublier d'avancer PC	Le CPU relit la même instruction à l'infini	PC avance à chaque cycle, sauf saut
Confondre registre et mémoire	Les registres sont dans le CPU, la mémoire est à part	Registre = tiroir proche ; mémoire = étagère
Penser que le CPU « voit » le programme entier	Il traite une instruction à la fois	Raisonner cycle par cycle
Sauter à une mauvaise adresse	Exécute des données comme du code	Vérifier les cibles de saut

Mini-boss

Mini-boss

Déboguer un processeur. Ce programme devait calculer 2 × 6 en additionnant, mais il ne s'arrête pas :

```
SET R0, 0
SET R1, 6
ADD R0, R1
JNZ R1, boucle   ; oublie DEC
HALT
```

1. Traduis-le dans le format mémoire du simulateur.
2. Ajoute un **garde-fou** : si le nombre de cycles dépasse 1 000, le simulateur lève une erreur.
3. Observe l'erreur, puis corrige le programme.
4. Ajoute au simulateur une option `trace=True` qui affiche PC et registres à chaque cycle.

Mini-boss · réponse

Pistes de correction

```
SET, ADD, DEC, JNZ, HALT = 1, 2, 3, 4, 0

def executer(memoire, trace=False, limite=1000):
    R = [0, 0, 0, 0]
    pc = cycles = 0
    while True:
        if cycles >= limite:
            raise RuntimeError(f"programme bloqué (PC={pc}, R={R})")
        opcode, a, b = memoire[pc], memoire[pc + 1], memoire[pc + 2]
        cycles += 1
        if trace:
            print(cycles, pc, R)
        pc += 3
        if opcode == SET:    R[a] = b
        elif opcode == ADD:  R[a] += R[b]
        elif opcode == DEC:  R[a] -= 1
        elif opcode == JNZ:
            if R[a] != 0:
                pc = b
```

Mini-boss · réponse

Pistes de correction (suite)

```
        elif opcode == HALT:
            return R, cycles

bogue = [SET, 0, 0, SET, 1, 6, ADD, 0, 1, JNZ, 1, 6, HALT, 0, 0]
try:
    executer(bogue)
except RuntimeError as e:
    print("erreur :", e)

# Ce programme additionne 6 + 5 + 4 + 3 + 2 + 1 (pas 2 x 6). Pour 2 x 6 il faut de
   ux compteurs.
corrige = [SET, 0, 0, SET, 1, 2, SET, 2, 6, ADD, 0, 2, DEC, 1, 0, JNZ, 1, 9, HALT,
    0, 0]
R, cycles = executer(corrige)
print(R[0], cycles)   # 12 et le nombre de cycles
```

Leçon : un garde-fou transforme un blocage silencieux en erreur claire. C'est le même principe que les délais d'attente (*timeouts*) d'un serveur.

Quiz éclair

Quiz éclair

1. Le registre PC contient :

A. le résultat du dernier calcul
B. l'adresse de la prochaine instruction
C. la taille de la mémoire

2. Le cycle du CPU est :

A. lire, écrire, effacer
B. compiler, lier, lancer
C. chercher, décoder, exécuter

3. Les registres sont :

A. la mémoire la plus rapide, dans le CPU

Quiz éclair

Quiz éclair (suite)

B. sur le disque
C. dans la carte graphique

4. 🔁 *Révision mission 02* : Un ISA est :

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

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

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

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Dans quelles situations de ta vie suis-tu un « compteur de programme » (une liste d'étapes avec un doigt posé dessus) ?
2. Qu'est-ce qui rend une boucle infinie si facile à écrire ?

À retenir

À retenir

1. PC pointe l'instruction suivante.
2. Cycle : fetch, decode, execute, mise à jour de PC.
3. Les registres sont les plus rapides.
4. Programme et données partagent la mémoire (von Neumann).
5. Les sauts modifient PC : boucles, conditions, appels.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Horloger du CPU

Prochaine mission : La hiérarchie mémoire

