Architecture des ordinateurs

Cours 4

4 novembre 2011

Micro-architecture

Intro

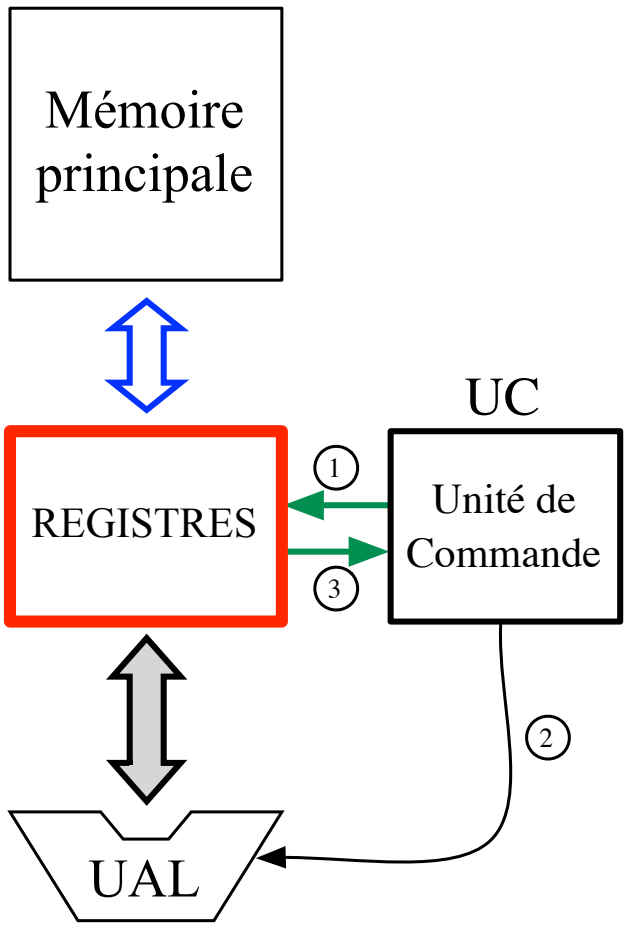
- Comment assembler les différents circuits vus dans les cours précédents pour fabriquer un processeur ?
- Comment interagir avec la mémoire ?
- L'UAL est en quelque sorte le cerveau de la machine. Comment peut-on la commander ? Quel langage utilise-t-on pour communiquer avec l'UAL ?
- Comment de simples opérations réalisables par l'UAL peuvent-elles aboutir à un programme ?

Il n'y a pas de réponse universelle, pas de modèle d'architecture. Pour chaque micro-processeur, des choix sont faits, en essayant d'équilibrer le coût et les performances.

Dans ce cours : exemple théorique d'architecture, version simplifiée de celle présentée dans le livre d'A. Tanenbaum.

Chemin des données : le rôle central des registres

Comment les différents éléments présent dans le processeur interagissent-ils ?

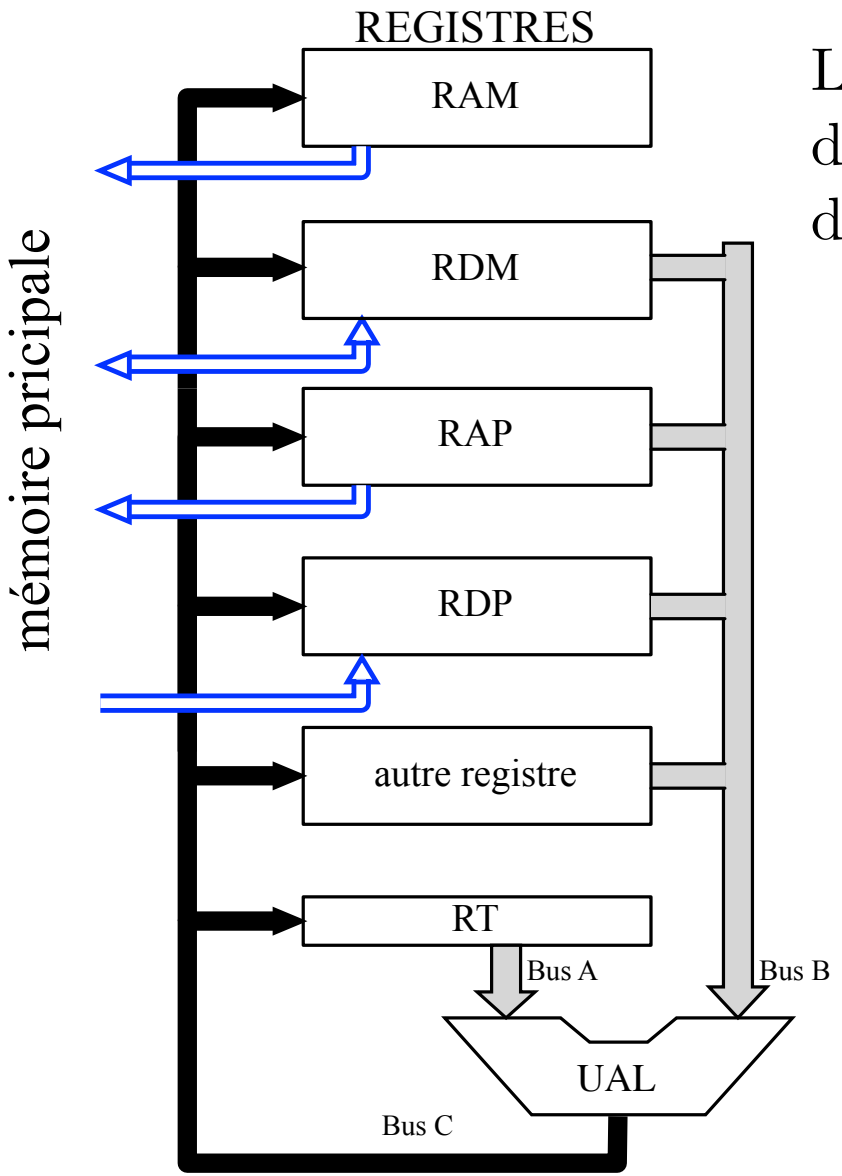


1. l'UC active certains registres pour :
   - *lire* en mémoire
   - *écrire* en mémoire
   - transférer des données *vers* l'UAL
   - transférer des données *depuis* l'UAL
2. l'UC commande l'action de l'UAL
3. l'état des registres permet de choisir la prochaine commande

De façon indirecte (par les registres) :

- La mémoire, l'UAL influent sur l'UC.
- L'UC influe sur la mémoire.

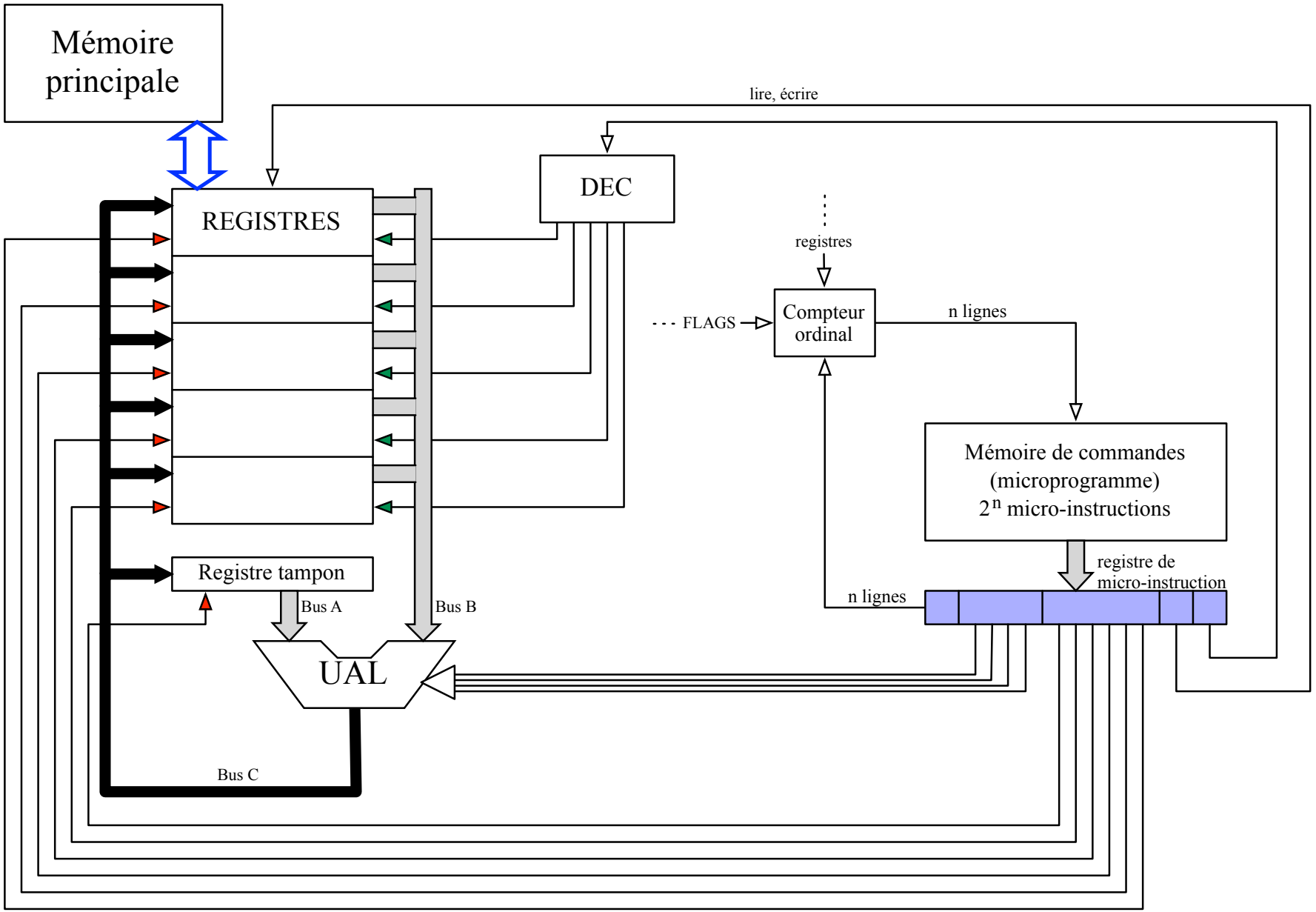
Au niveau des registres



Les registres sont spécialisés, afin d'aiguiller les informations entre les différents composants :

- RAM : Registre d'Adresses Mémoire
- RDM : Registre de Données Mémoire
- RAP : Registre d'Adresses de Prog.
- RDP : Registre de Données de Prog.
- RT : Registre Tampon
- autres registres :
  - SP (adresse du sommet de la pile),
  - BP (adresse de la base de la pile),
  - registres de calcul,
  - ...

Micro-architecture simple



Boucle d'exécution des micro-instructions

Boucle CHARGEMENT - DÉCODAGE - EXÉCUTION

Tant que VRAI (le processeur est alimenté) **faire** :

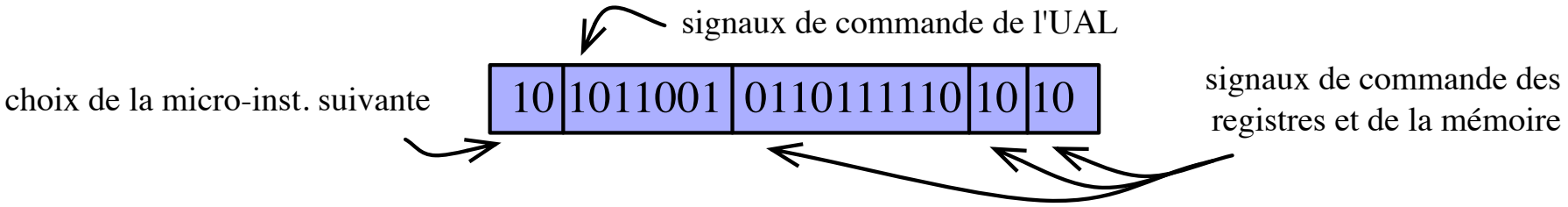
1. Récupérer la micro-instruction courante en mémoire et la placer dans le registre de micro-instruction.
2. Modifier la valeur du compteur ordinal pour qu'il pointe vers l'adresse de la prochaine micro-instruction.
3. Si besoin, localiser un mot (donnée/prog.) en mémoire principale (charger le registre d'adresse).
4. Si besoin, charger le mot mémoire dans un registre de données.
5. Exécuter la micro-instruction.

Fin Tant que

Micro-programme

Microcode

Micro-instruction : mot binaire codant des signaux de commande :



Micro-code : suite de micro-instructions stockées en mémoire ROM

Instruction : (couche ISA) : bloc de micro-instructions réalisant une “opération simple”

Exemples d’instructions :

- instructions de déplacement : MOV, XCHG
- manipulation de la pile : PUSH et POP
- instructions arithmétiques et logiques : ADD, SUB, AND, OR et NOT
- sauts et boucles : JUMP, Jxx, LOOPxx
- appels de fonctions : CALL et RET

Problèmes de gestion de l'espace mémoire

Problème n°1 Calculer $((1+2)\times(3+4))+((5+6)\times(7+8))$ en mémorisant les calculs intermédiaires.

- Combien faut-il de registres dans ce cas là ?
- Quel est le nombre *maximal* de registres utilisés dans un calcul ?

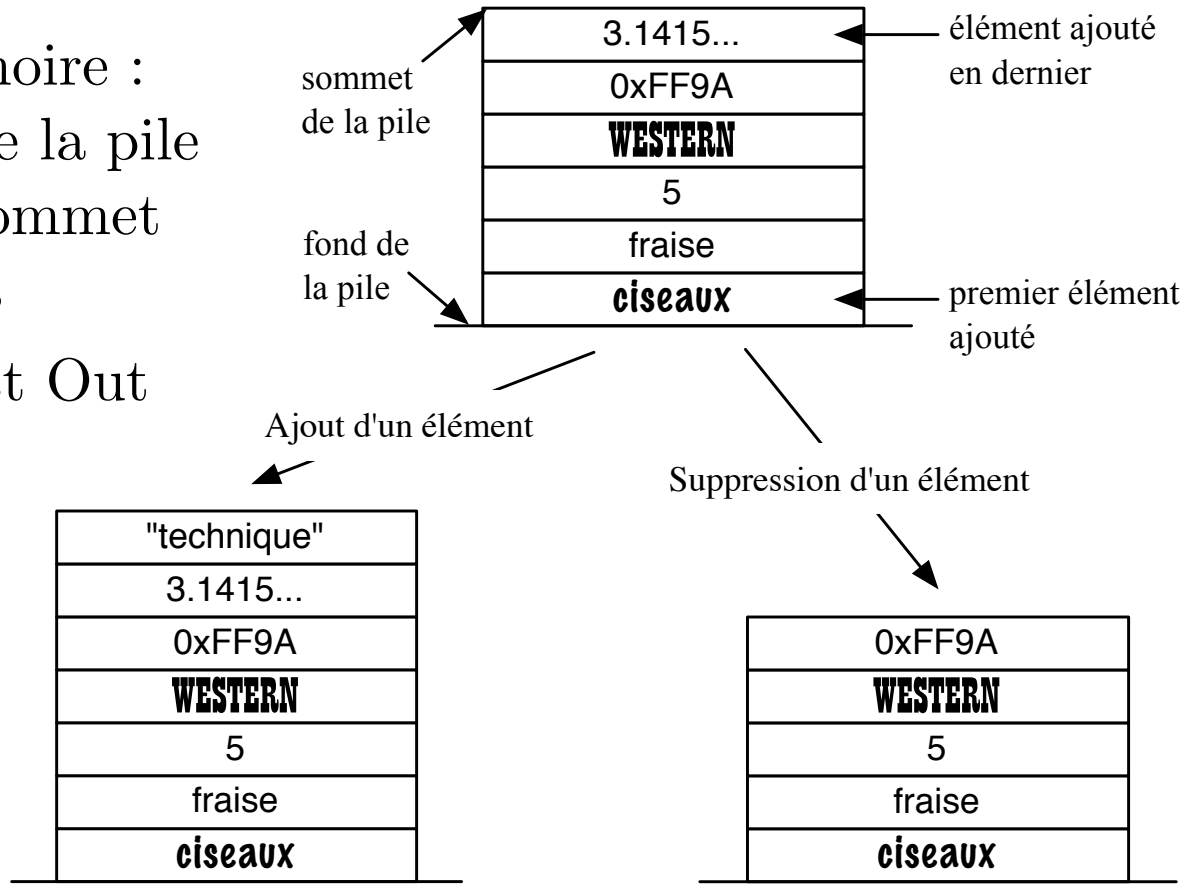
Problème n°2 Stocker les variables locales des fonctions.

- Où sont conservées les variables locales des fonctions ?
- Solution simple : attribuer à toutes les variables des adresses fixes différentes.
- Problème : que se passe-t-il si une fonction en appelle une autre ou s'appelle elle-même ?

Solution : la Pile

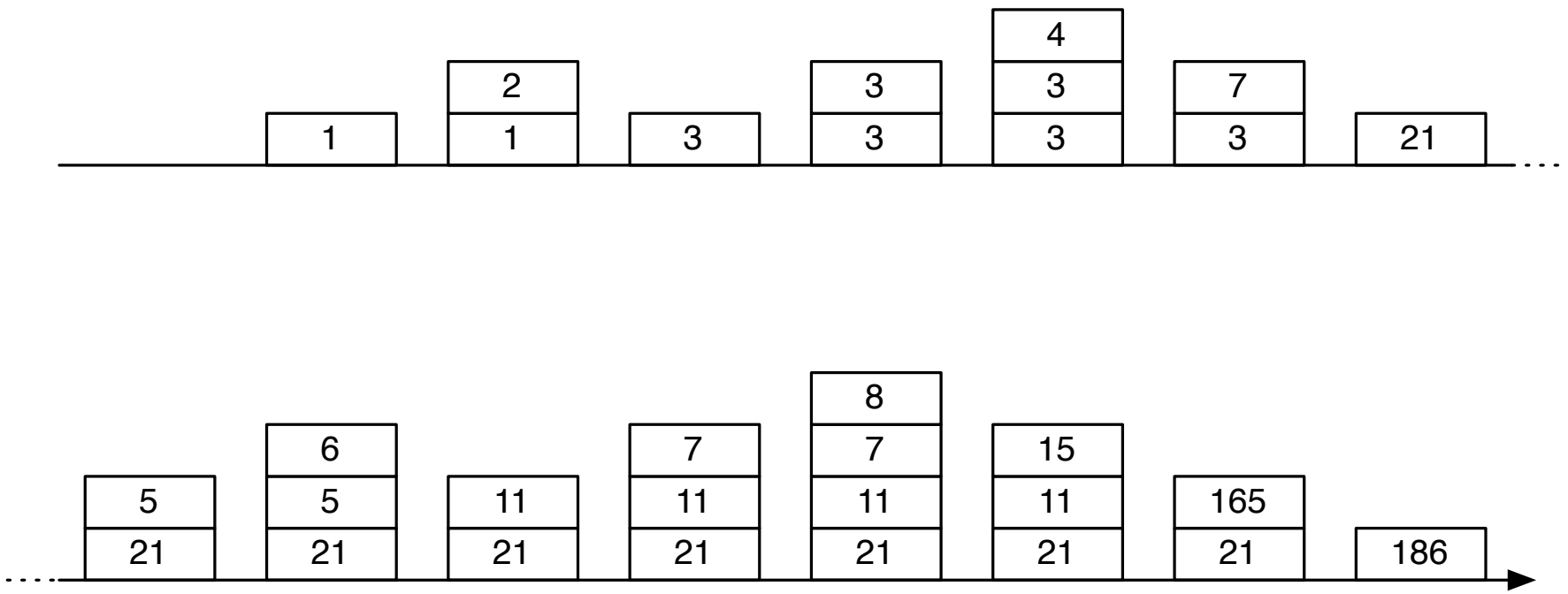
la **pile** = espace de stockage situé en mémoire principale.

- taille variable
- accès par le sommet
- localisation en mémoire :
  - adresse du fond de la pile
  - pointeur vers le sommet
  - adresses contiguës
- LIFO : Last In First Out
- Opérations :
  - PUSH (ajout)
  - POP (retrait)



Pile d'opérandes

Calcul de $((1+2)\times(3+4))+((5+6)\times(7+8))$



Pile = bloc de variables locales

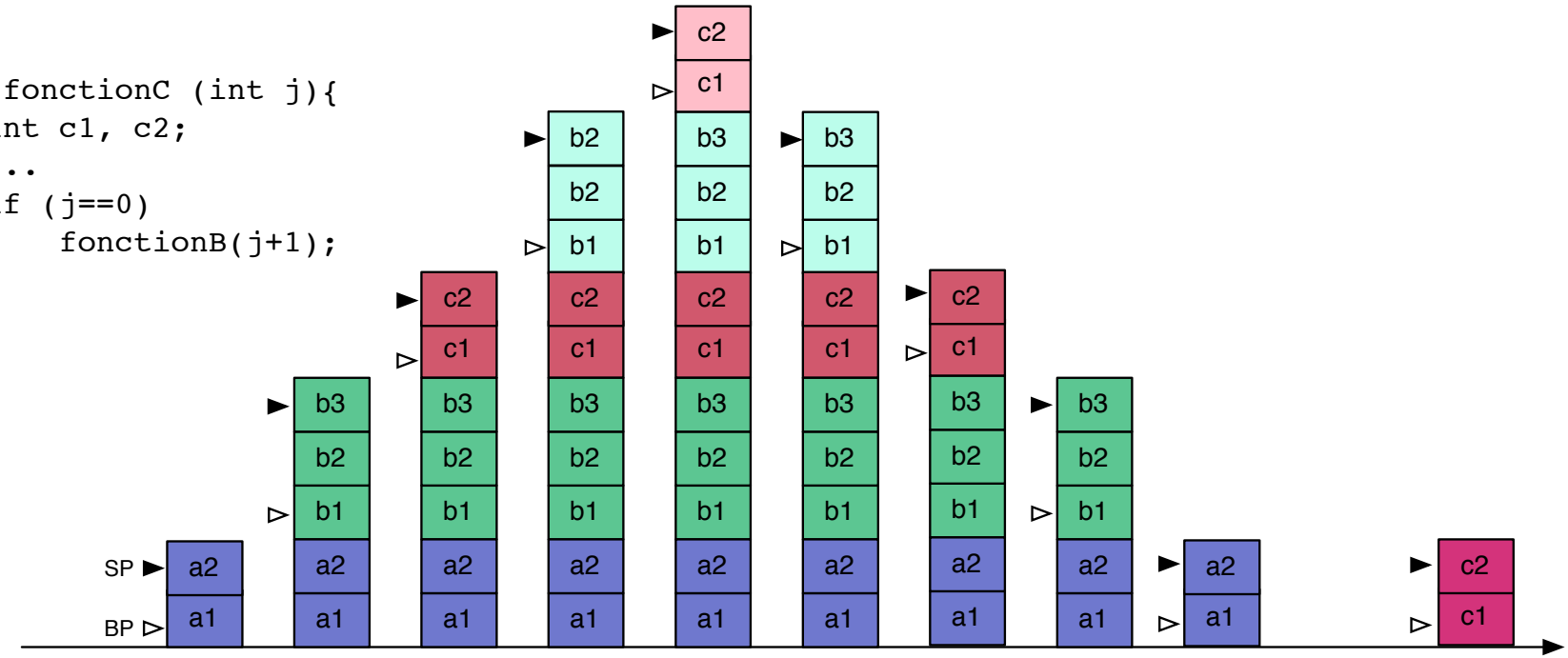
```
void fonctionA (void){
    int a1, a2;
    ...
    fonctionB(0);
}

void fonctionB (int i){
    int b1, b2, b3;
    ...
    fonctionC(i);
}

void fonctionC (int j){
    int c1, c2;
    ...
    if (j==0)
        fonctionB(j+1);
}
```

```
int main(int argc, char *argv[]){
    fonctionA();
    fonctionC(1);

    return 0;
}
```



Exemple d'instruction arithmétique : ADD

Rôle : "retirer" les deux éléments au sommet de la pile, les additionner et placer le résultat au sommet de la pile.

1ère étape : récupérer les mots au sommets de la pile.

- placer l'adresse du sommet de la pile (contenue dans le registre SP) dans RAM ;
- lire dans RDM le sommet de la pile et le placer dans RT ;
- calculer l'adresse du mot juste "en dessous" du sommet de la pile et la placer dans RAM ainsi que dans le registre SP (supprime le 1er mot à additionner).

2ème étape : calculer la somme et la placer au sommet de la pile.

- lire en mémoire le mot pointé par RAM (placé dans RDM) ; additionner RDM et RT ; mettre le résultat dans RDM ;
- écrire en mémoire le contenu de RDM à l'adresse contenue dans RAM (à la place du second mot à additionner).

Exemple d'opération sur la pile : PUSH *num_var*

Rôle : mettre la variable locale *num_var* au sommet de la pile (*num_var* est dans le registre de données de programme RDP).

1ère étape : récupérer la variable locale désignée par *num_var*.

- placer l'adresse du fond de la pile (gardée dans le registre BP) dans le registre tampon RT ;
- calculer l'adresse de la variable locale RT + RDP et mettre le résultat dans le registre d'adresse mémoire RAM ;
- lire la valeur de la variable locale dans le registre de données mémoire RDM ;

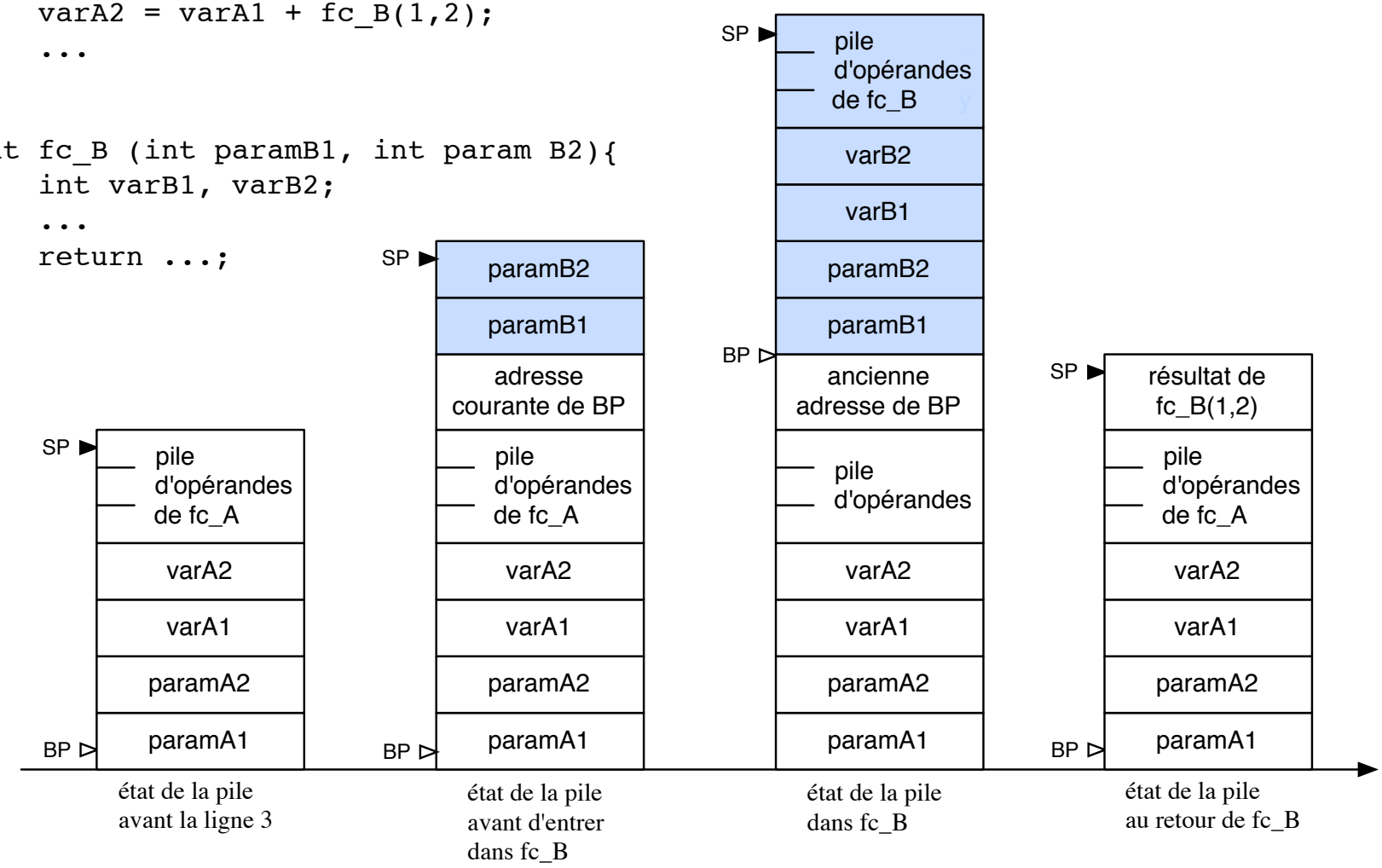
2ème étape : placer la variable locale au sommet de la pile.

- calculer l'adresse du nouveau sommet de la pile et la placer dans RAM ainsi que dans le registre SP ;
- écrire en mémoire le contenu de RDM à l'adresse pointée par le registre RAM.

Appels de fonctions (CALL et RET)

```
1   fc_A (int paramA1, int param A2){
2       int varA1, varA2;
        ...
3       varA2 = varA1 + fc_B(1,2);
        ...
    }

4   int fc_B (int paramB1, int param B2){
5       int varB1, varB2;
        ...
6       return ...;
    }
```



Performances

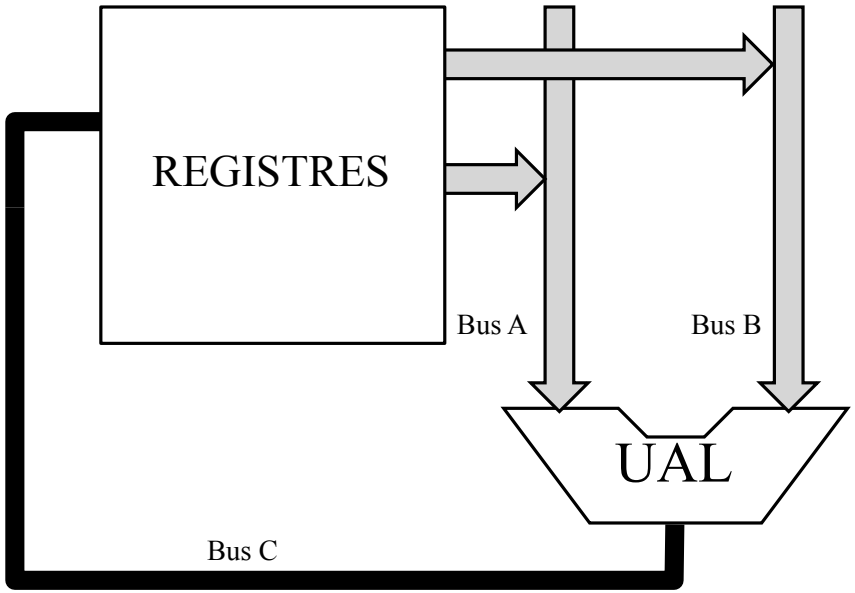
Comment améliorer l'architecture ?

Lors de la conception d'une micro-architecture, plusieurs paramètres entre en considération, notamment :

- la rapidité
- le coût

⇒ Trouver des compromis : par exemple, rajouter des registres pour accéder rapidement à plus de données, mais pas trop car les registres coûtent très cher.

Exple : la **mémoire cache**



autre exemple : rajout d'un bus A complet.

Mémoire cache : principe

En pratique, les accès à la mémoire principale sont très lents ; plusieurs cycles d'horloge sont souvent nécessaires pour les opérations en mémoire.

Solution : rapprocher les données les plus souvent utilisées !

Principe de localité : à tout instant, un programme accède à une *petite* partie de son espace d'adressage ; 2 types de localité :

- **Temporelle** : Si un mot a été utilisé récemment, il a plus de chances d'être *réutilisé* (exemple : boucles) ;
- **Spatiale** : Si un mot a été utilisé récemment, les mots avec des *adresses voisines* ont plus de chances d'être utilisés (exemple : pile, tableaux) ;

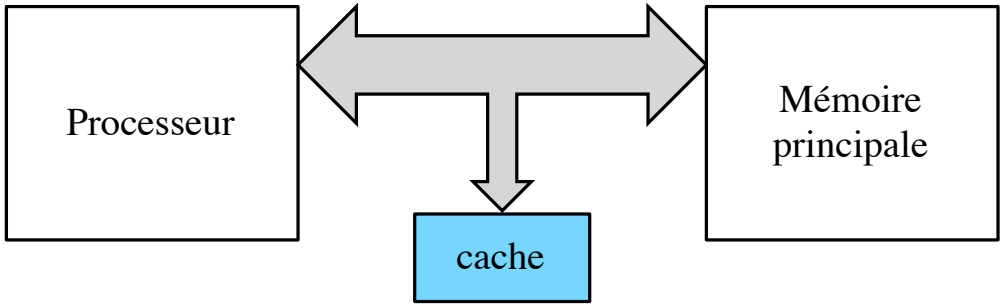
Idée : on place dans le **cache** (mémoire proche du processeur) :

▷ Les données les *plus récemment adressées* ;

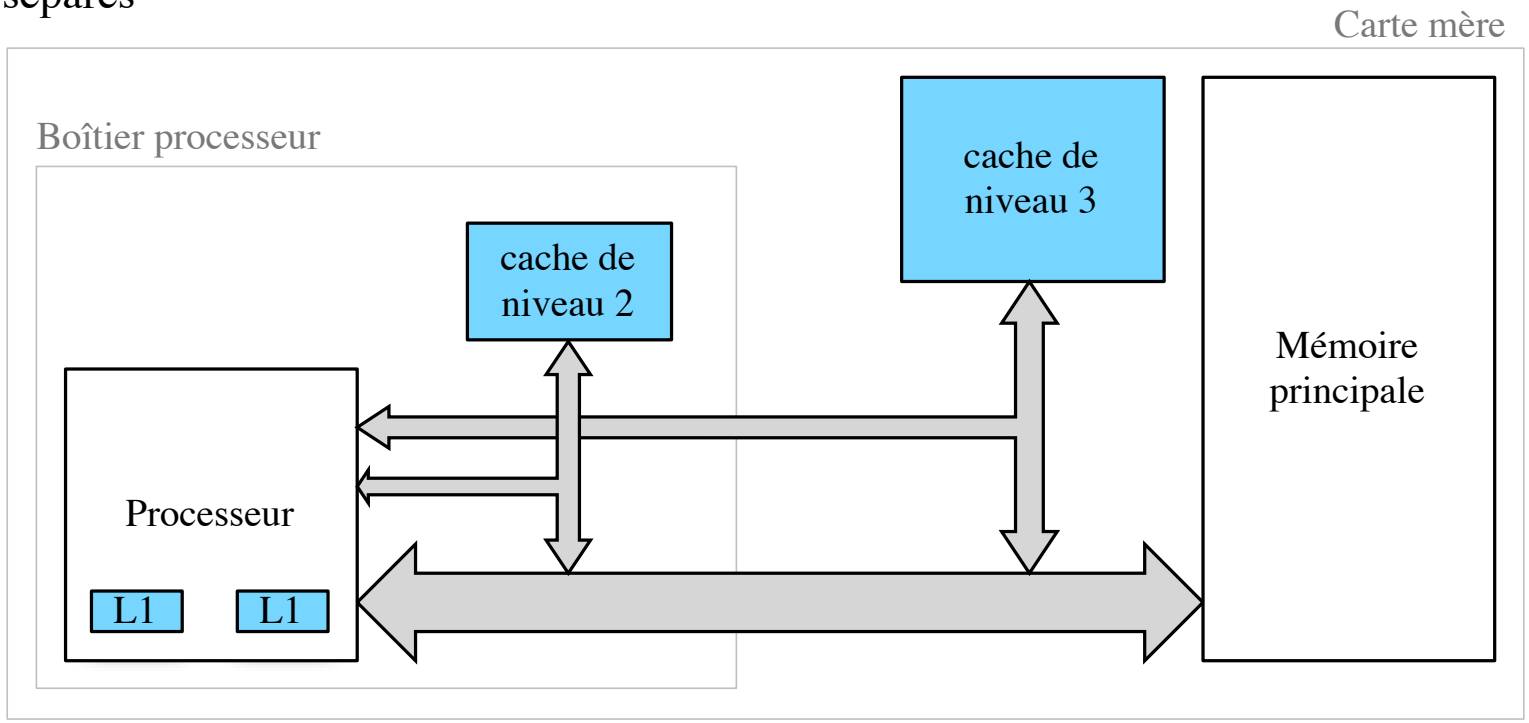
▷ Les données en *blocs* (lignes de cache).

Hiérarchie des caches

Cache unique

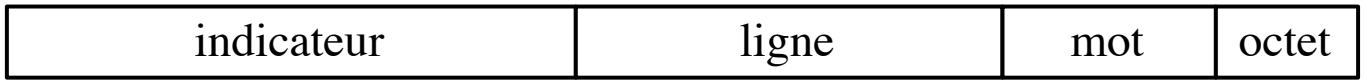


Caches séparés

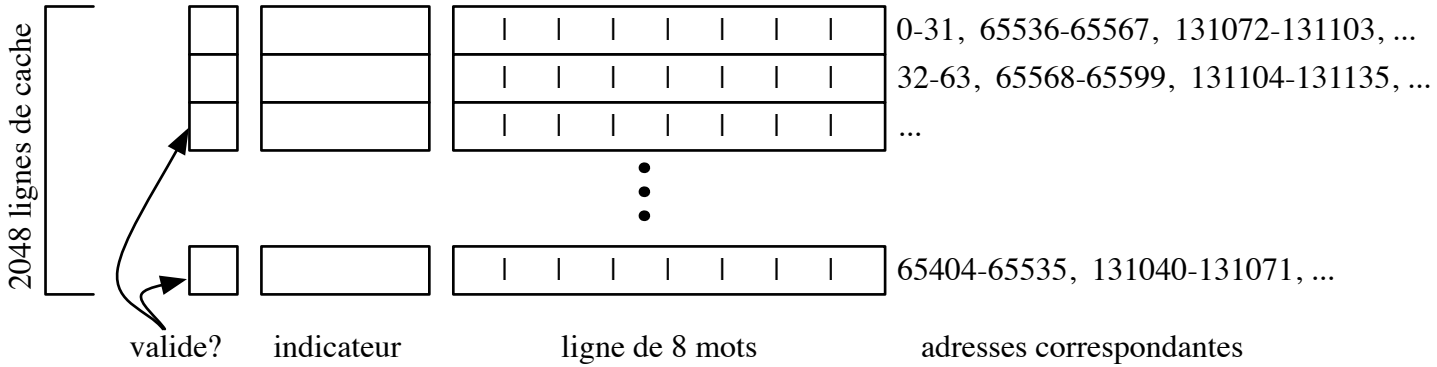


Cache direct

- ligne de cache = 32 octets (en général, entre 4 et 64), soit 8 mots.
- à chaque ligne on associe, 1 bit *valide* et un *indicateur* de 16 bits qui identifie les adresses mémoire correspondant à cette ligne.
- adresse mémoire ⇔ adresse virtuelle

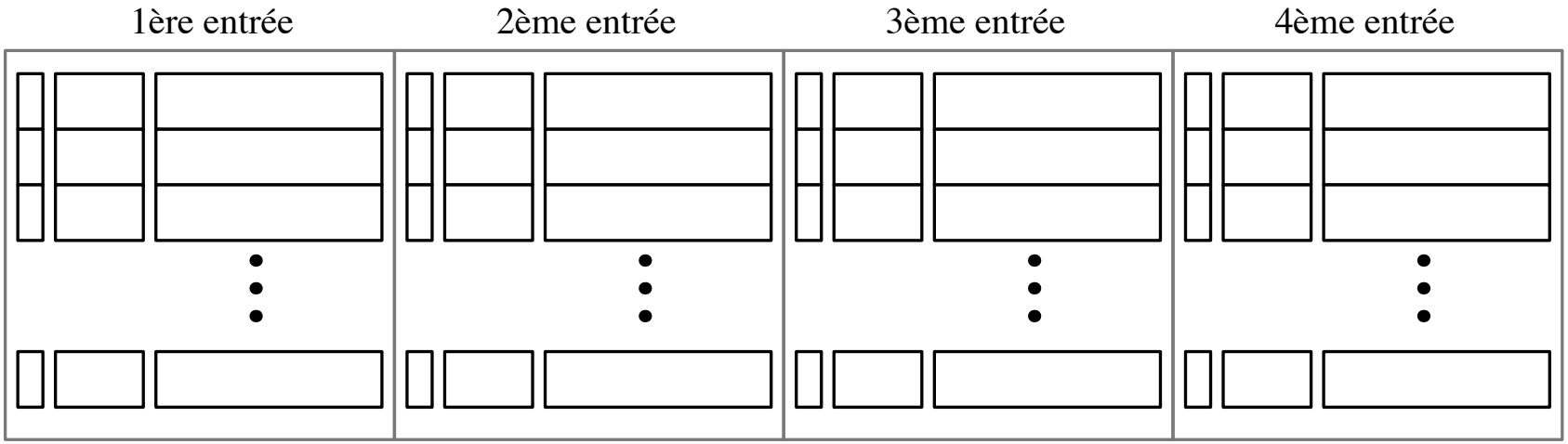


- 1 seul emplacement possible dans le cache pour un mot donné !



1. trouver la ligne de cache,
2. vérifier si elle est valide,
3. vérifier l'indicateur ; en cas d'échec, remplacement de la ligne.

Cache associatif



- Chaque entrée du cache contient *plusieurs blocs mémoire* (ici 4) en même temps.
- Besoin d'un algorithme pour choisir la page à effacer si une page est manquante : LRU (*Least Recently Used*).
- Compromis entre le nombre d'entrées et la complexité de l'algorithme.
- *Rmq :* À quel moment faire la mise à jour en mémoire ?