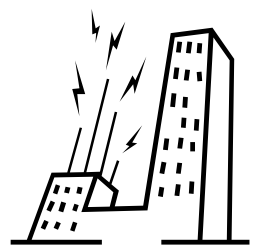
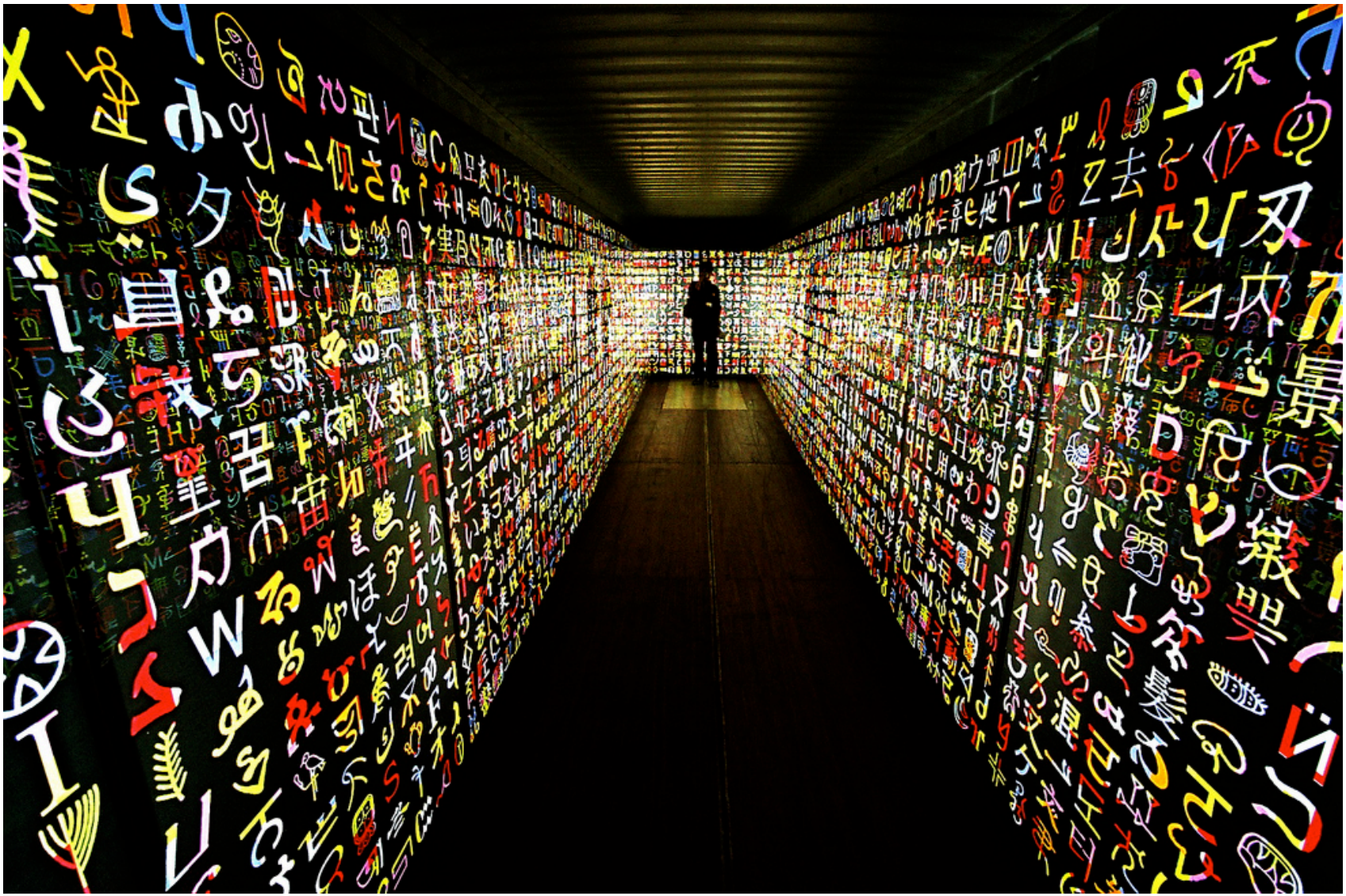
Architecture des ordinateurs IX

Frédéric Boulanger

CentraleSupélec



Traduction des langages de programmation



Fossé sémantique

Les langages de programmation proposent des concepts abstraits qui sont absents de ce que propose un processeur :

- types de données

Fossé sémantique

Les langages de programmation proposent des concepts abstraits qui sont absents de ce que propose un processeur :

- types de données
- structures de contrôle :

Fossé sémantique

Les langages de programmation proposent des concepts abstraits qui sont absents de ce que propose un processeur :

- types de données
- structures de contrôle :
  - if ... then ... else

Fossé sémantique

Les langages de programmation proposent des concepts abstraits qui sont absents de ce que propose un processeur :

- types de données
- structures de contrôle :
  - if ... then ... else
  - while

Fossé sémantique

Les langages de programmation proposent des concepts abstraits qui sont absents de ce que propose un processeur :

- types de données
- structures de contrôle :
  - if ... then ... else
  - while
  - fonctions

Fossé sémantique

Les langages de programmation proposent des concepts abstraits qui sont absents de ce que propose un processeur :

- types de données
- structures de contrôle :
  - if ... then ... else
  - while
  - fonctions

Cette différence de niveaux d'abstraction est appelée fossé sémantique

Fossé sémantique

Les langages de programmation proposent des concepts abstraits qui sont absents de ce que propose un processeur :

- types de données
- structures de contrôle :
  - if ... then ... else
  - while
  - fonctions

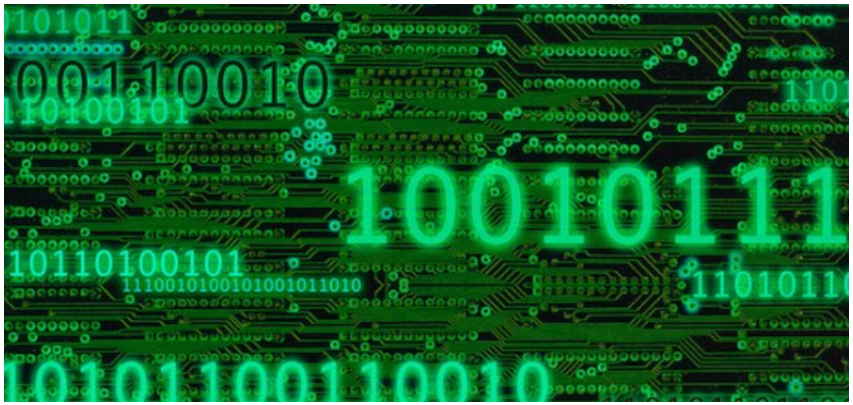
Cette différence de niveaux d'abstraction est appelée fossé sémantique

La tendance actuelle est de privilégier les performances, et de laisser les compilateurs et interpréteurs combler le fossé sémantique.

Représentation des données

Pour le microprocesseur

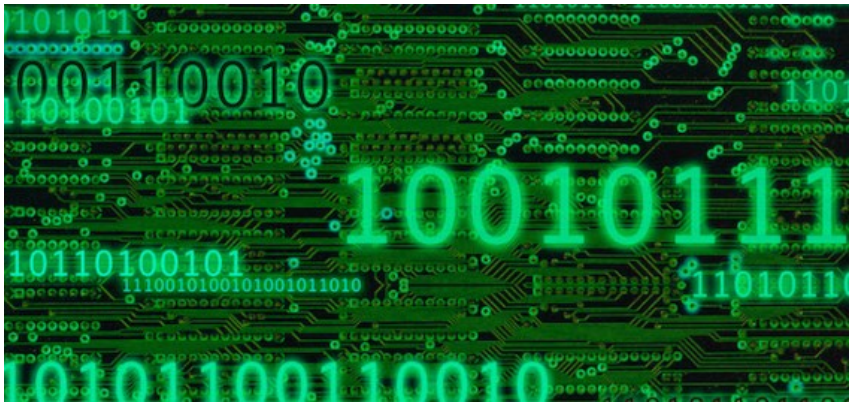
- mots de 8, 16, 32 ou 64 bits



Représentation des données

Pour le microprocesseur

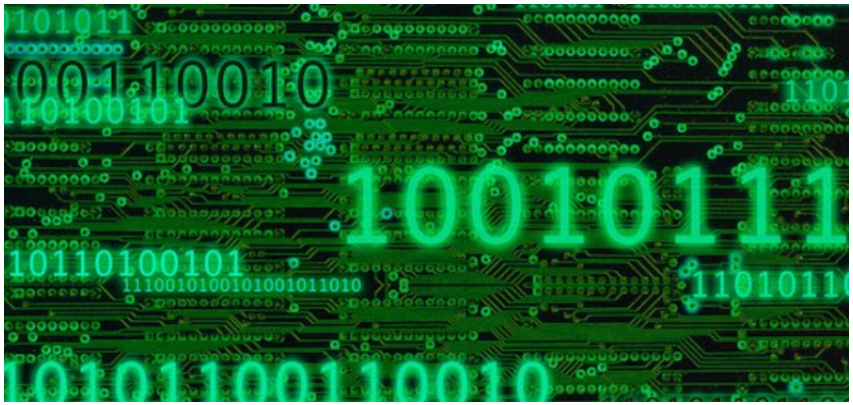
- mots de 8, 16, 32 ou 64 bits
- trois interprétations :



Représentation des données

Pour le microprocesseur

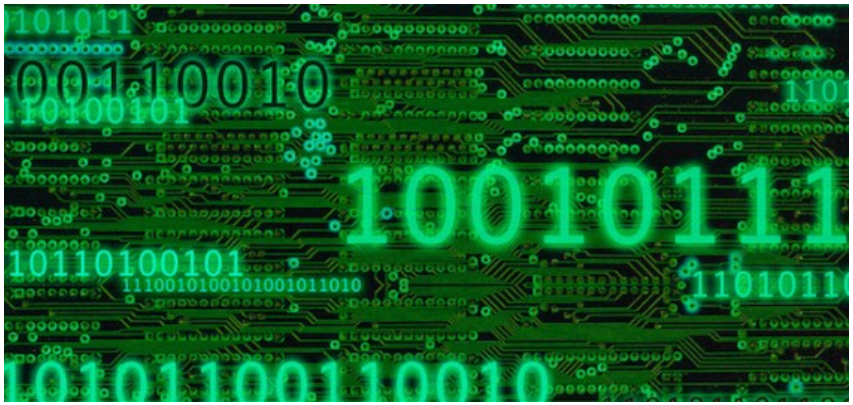
- mots de 8, 16, 32 ou 64 bits
- trois interprétations :
  - code d'une instruction



Représentation des données

Pour le microprocesseur

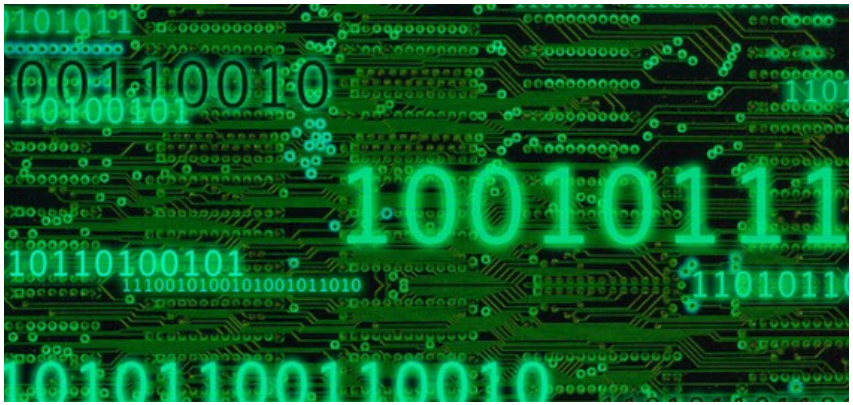
- mots de 8, 16, 32 ou 64 bits
- trois interprétations :
  - code d'une instruction
  - adresse



Représentation des données

Pour le microprocesseur

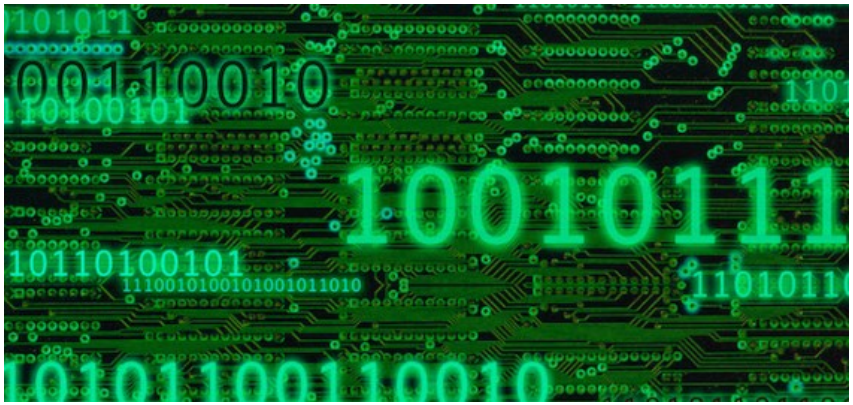
- mots de 8, 16, 32 ou 64 bits
- trois interprétations :
  - code d'une instruction
  - adresse
  - donnée



Représentation des données

Pour le microprocesseur

- mots de 8, 16, 32 ou 64 bits
- trois interprétations :
  - code d'une instruction
  - adresse
  - donnée

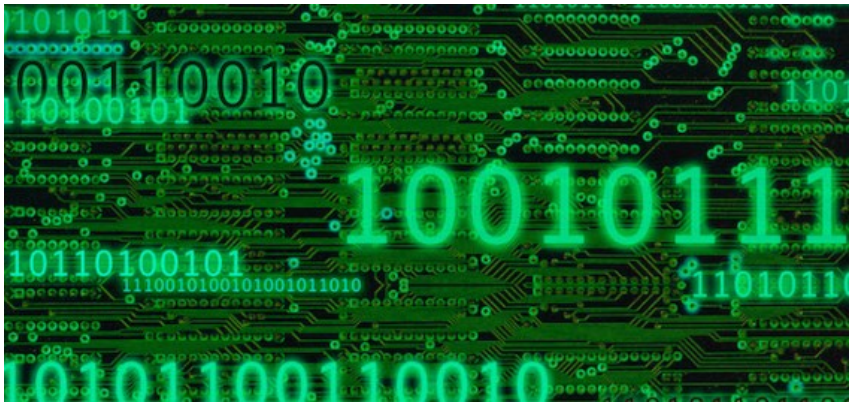


Langages de programmation

Représentation des données

Pour le microprocesseur

- mots de 8, 16, 32 ou 64 bits
- trois interprétations :
  - code d'une instruction
  - adresse
  - donnée



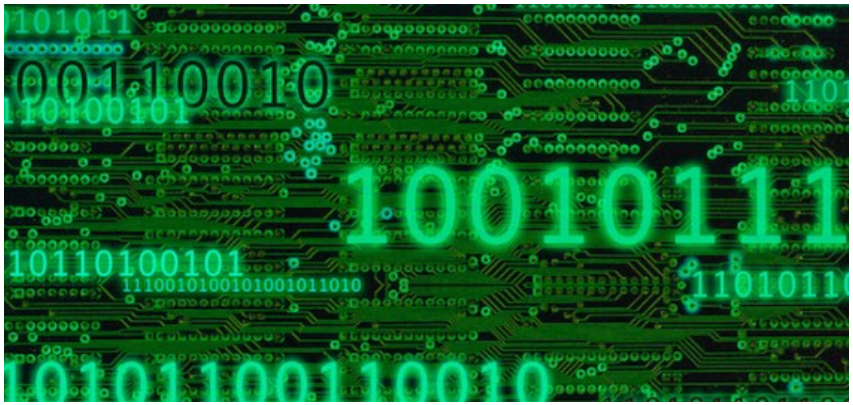
Langages de programmation

- booléens, entiers, flottants, chaînes de caractères

Représentation des données

Pour le microprocesseur

- mots de 8, 16, 32 ou 64 bits
- trois interprétations :
  - code d'une instruction
  - adresse
  - donnée



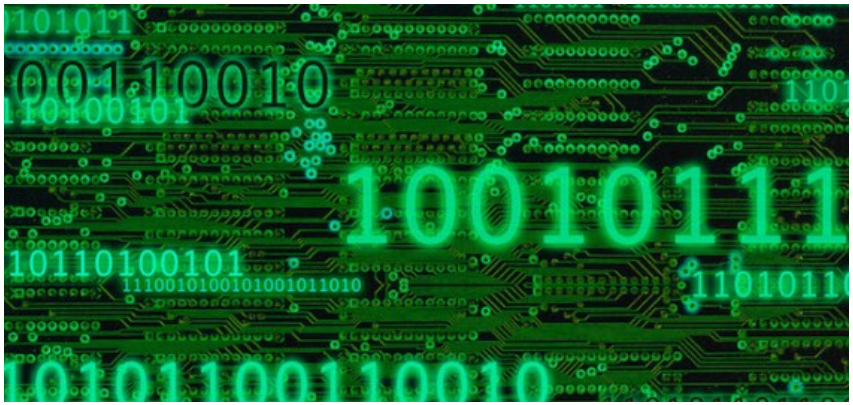
Langages de programmation

- booléens, entiers, flottants, chaînes de caractères
- types structurés : tableaux, classes, enregistrements

Représentation des données

Pour le microprocesseur

- mots de 8, 16, 32 ou 64 bits
- trois interprétations :
  - code d'une instruction
  - adresse
  - donnée



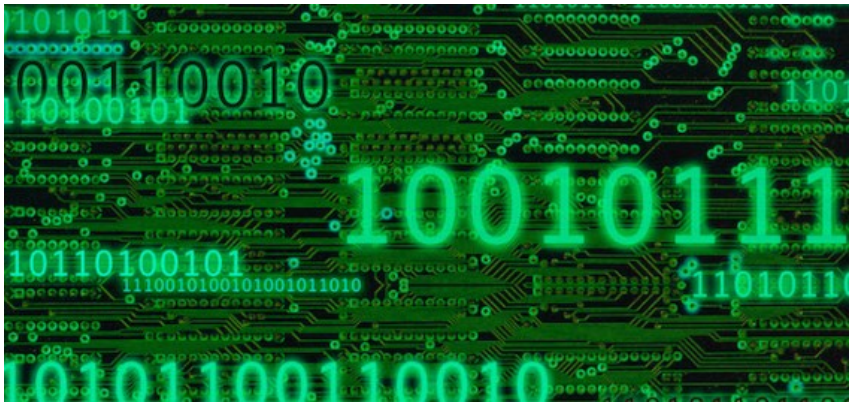
Langages de programmation

- booléens, entiers, flottants, chaînes de caractères
- types structurés : tableaux, classes, enregistrements
- Représentation :

Représentation des données

Pour le microprocesseur

- mots de 8, 16, 32 ou 64 bits
- trois interprétations :
  - code d'une instruction
  - adresse
  - donnée



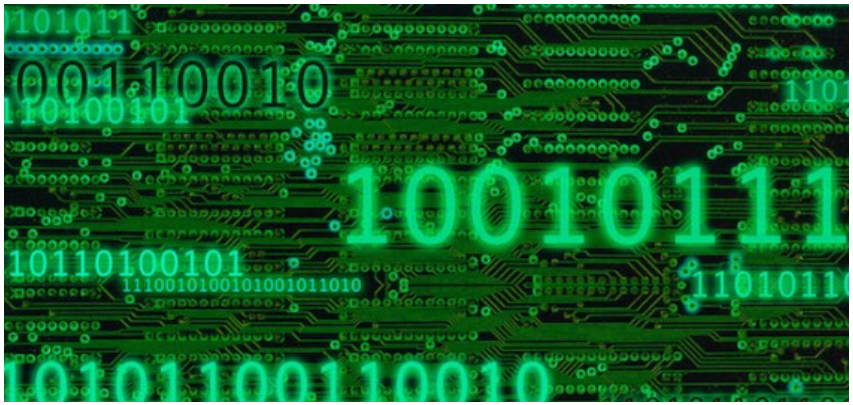
Langages de programmation

- booléens, entiers, flottants, chaînes de caractères
- types structurés : tableaux, classes, enregistrements
- Représentation :
  - booléens : faux = 0, vrai = 1

Représentation des données

Pour le microprocesseur

- mots de 8, 16, 32 ou 64 bits
- trois interprétations :
  - code d'une instruction
  - adresse
  - donnée



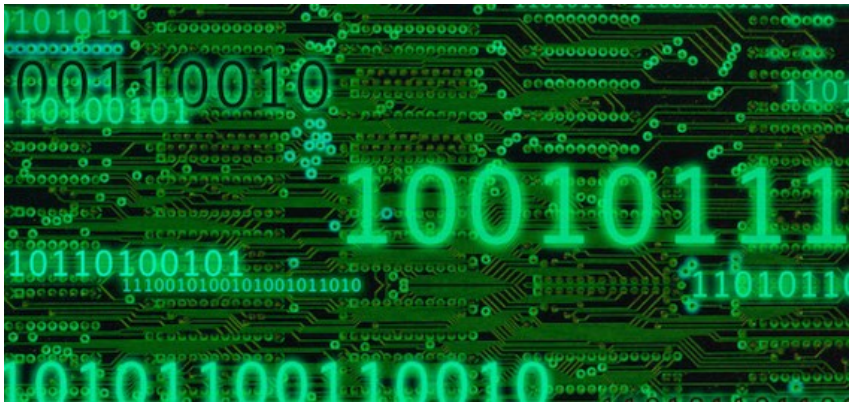
Langages de programmation

- booléens, entiers, flottants, chaînes de caractères
- types structurés : tableaux, classes, enregistrements
- Représentation :
  - booléens : faux = 0, vrai = 1
  - entiers : complément à 2

Représentation des données

Pour le microprocesseur

- mots de 8, 16, 32 ou 64 bits
- trois interprétations :
  - code d'une instruction
  - adresse
  - donnée



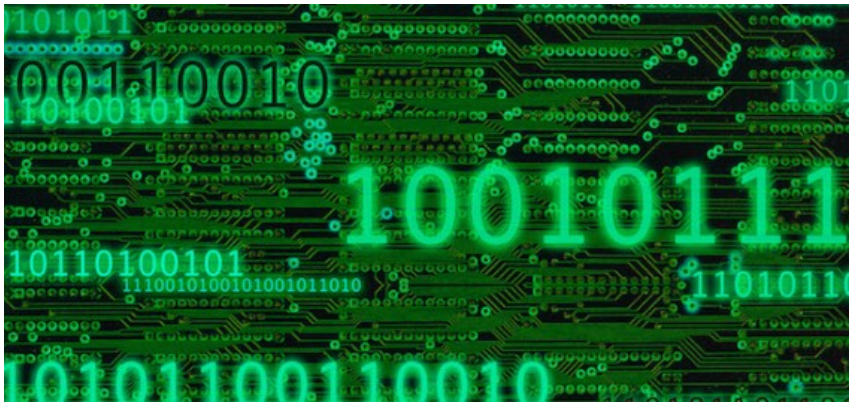
Langages de programmation

- booléens, entiers, flottants, chaînes de caractères
- types structurés : tableaux, classes, enregistrements
- Représentation :
  - booléens : faux = 0, vrai = 1
  - entiers : complément à 2
  - flottants : norme IEEE 754

Représentation des données

Pour le microprocesseur

- mots de 8, 16, 32 ou 64 bits
- trois interprétations :
  - code d'une instruction
  - adresse
  - donnée



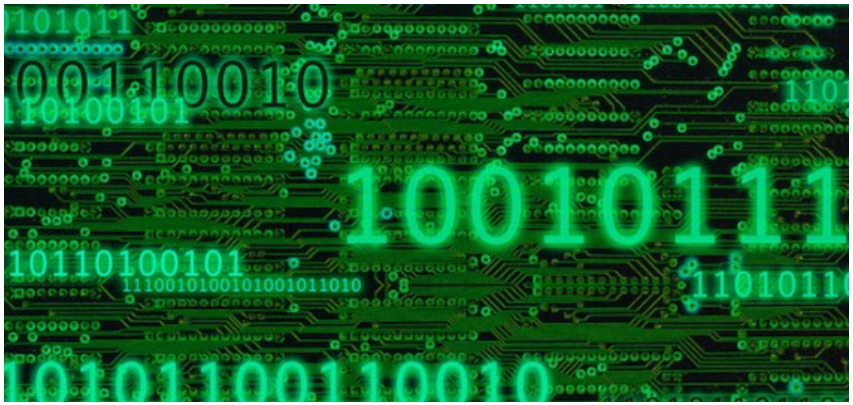
Langages de programmation

- booléens, entiers, flottants, chaînes de caractères
- types structurés : tableaux, classes, enregistrements
- Représentation :
  - booléens : faux = 0, vrai = 1
  - entiers : complément à 2
  - flottants : norme IEEE 754
  - caractères : Unicode, UTF-8

Représentation des données

Pour le microprocesseur

- mots de 8, 16, 32 ou 64 bits
- trois interprétations :
  - code d'une instruction
  - adresse
  - donnée



Langages de programmation

- booléens, entiers, flottants, chaînes de caractères
- types structurés : tableaux, classes, enregistrements
- Représentation :
  - booléens : faux = 0, vrai = 1
  - entiers : complément à 2
  - flottants : norme IEEE 754
  - caractères : Unicode, UTF-8
  - tableaux et structures : mots consécutifs en mémoire

Structures de contrôle

Pour le microprocesseur

- `cmp`, `b`, `beq` et `blt`

Structures de contrôle

Pour le microprocesseur

- `cmp`, `b`, `beq` et `blt`

Langages de programmation

Structures de contrôle

Pour le microprocesseur

- `cmp`, `b`, `beq` et `blt`

Langages de programmation

- comparaisons variées : $<=, <, ==, >, >=$

Structures de contrôle

Pour le microprocesseur

- `cmp`, `b`, `beq` et `blt`

Langages de programmation

- comparaisons variées : $<=$, $<$, $==$, $>$, $>=$
- boucles `while`

Structures de contrôle

Pour le microprocesseur

- `cmp`, `b`, `beq` et `blt`

Langages de programmation

- comparaisons variées : $<=, <, ==, >, >=$
- boucles `while`
- définition et appel de fonctions, récursion

Structures de contrôle

Pour le microprocesseur

- `cmp`, `b`, `beq` et `blt`

Langages de programmation

- comparaisons variées : $<=, <, ==, >, >=$
- boucles `while`
- définition et appel de fonctions, récursion

Comment coder ces structures de contrôle ?

Conditions

- `a<b` : `cmp a, b; blt vrai`

Conditions

- a < b : `cmp a, b; blt vrai`
- a <= b : idem que `not b < a` $\Rightarrow$ `cmp b, a; blt faux`

Conditions

- a < b : `cmp a, b; blt vrai`
- a <= b : idem que `not b < a` $\Rightarrow$ `cmp b, a; blt faux`
- a == b : `cmp a, b; beq vrai`

Conditions

- a < b : `cmp a, b; blt vrai`
- a <= b : idem que `not b < a` $\Rightarrow$ `cmp b, a; blt faux`
- a == b : `cmp a, b; beq vrai`
- a != b : `cmp a, b; beq faux`

Conditions

- a < b : `cmp a, b; blt vrai`
- a <= b : idem que `not b < a` $\Rightarrow$ `cmp b, a; blt faux`
- a == b : `cmp a, b; beq vrai`
- a != b : `cmp a, b; beq faux`
- a > b : `cmp b, a; blt vrai`

Conditions

- a < b : `cmp a, b; blt vrai`
- a <= b : idem que `not b < a` ⇒ `cmp b, a; blt faux`
- a == b : `cmp a, b; beq vrai`
- a != b : `cmp a, b; beq faux`
- a > b : `cmp b, a; blt vrai`
- a >= b : idem que `not a < b` ⇒ `cmp a, b; blt faux`

Conditions

Exemple

```
if r0 < r1 :
  r0 = r1 - r0
else :
  r1 = r0 - r1
```

```
        cmp r0, r1
        blt vrai
        sub r1, r0, r1
        b   fin
@vrai   sub r0, r1, r0
@fin
```

Conditions

Exemple

```
if r0 <= r1 :
  r0 = r1 - r0
else :
  r1 = r0 - r1
```

```
        cmp r1, r0
        blt faux
@vrai   sub r0, r1, r0
        b   fin
@faux   sub r1, r0, r1
@fin
```

Boucles

Se codent comme un `if` avec un branchement à la fin pour revenir au début de la boucle.

```
r2 = 0
while r0 >= r1 :
    r0 = r1 - r0
    r2 = r2 + 1
```

```
          mov r2, #0
@boucle   cmp r0, r1
          blt fin
          sub r0, r1, r0
          add r2, r2, #1
          b   boucle
@fin
```

Boucles

Autre exemple :

```
while r0 < r1 :
    r1 = r1 - 1
```

```
@boucle   cmp r0, r1
          blt ok
          b fin
@ok       sub r1, r1, #1
          b   boucle
@fin
```

Fonctions

Définition

- Une fonction peut être identifiée à l'adresse de sa première instruction

Fonctions

Définition

- Une fonction peut être identifiée à l'adresse de sa première instruction
- ... mais que fait-on à la fin ?

Fonctions

Définition

- Une fonction peut être identifiée à l'adresse de sa première instruction
- ... mais que fait-on à la fin ?
- Les arguments peuvent être passés dans des registres

Fonctions

Définition

- Une fonction peut être identifiée à l'adresse de sa première instruction
- ... mais que fait-on à la fin ?
- Les arguments peuvent être passés dans des registres
- ... mais s'il y en a plus de 8 ?

Fonctions

Définition

- Une fonction peut être identifiée à l'adresse de sa première instruction
- ... mais que fait-on à la fin ?
- Les arguments peuvent être passés dans des registres
- ... mais s'il y en a plus de 8 ?

Appel

Fonctions

Définition

- Une fonction peut être identifiée à l'adresse de sa première instruction
- ... mais que fait-on à la fin ?
- Les arguments peuvent être passés dans des registres
- ... mais s'il y en a plus de 8 ?

Appel

- On place les arguments dans des registres

Fonctions

Définition

- Une fonction peut être identifiée à l'adresse de sa première instruction
- ... mais que fait-on à la fin ?
- Les arguments peuvent être passés dans des registres
- ... mais s'il y en a plus de 8 ?

Appel

- On place les arguments dans des registres
- et on saute à l'adresse de la fonction (`b func`)

Fonctions

Définition

- Une fonction peut être identifiée à l'adresse de sa première instruction
- ... mais que fait-on à la fin ?
- Les arguments peuvent être passés dans des registres
- ... mais s'il y en a plus de 8 ?

Appel

- On place les arguments dans des registres
- et on saute à l'adresse de la fonction (`b func`)
- ... mais comment revient-on après l'appel ?

Fonctions

Solution

- On place les arguments dans des registres

Fonctions

Solution

- On place les arguments dans des registres
- on mémorise dans un registre (`r7`) l'adresse de l'instruction qui suit l'appel

Fonctions

Solution

- On place les arguments dans des registres
- on mémorise dans un registre (`r7`) l'adresse de l'instruction qui suit l'appel
- et on saute à l'adresse de la fonction (`b func`)

Fonctions

Solution

- On place les arguments dans des registres
- on mémorise dans un registre (`r7`) l'adresse de l'instruction qui suit l'appel
- et on saute à l'adresse de la fonction (`b func`)
- la fonction place le résultat dans `r0`

Fonctions

Solution

- On place les arguments dans des registres
- on mémorise dans un registre (`r7`) l'adresse de l'instruction qui suit l'appel
- et on saute à l'adresse de la fonction (`b func`)
- la fonction place le résultat dans `r0`
- et saute à l'adresse contenue dans `r7`

Fonctions

Exemple

```
% r0 <- r0 * r1
@mult   mov r2, #0
@loop   cmp r0, #0
        beq fin
        add r2, r2, r1
        sub r0, r0, #1
        b loop
@fin    mov r0, r2
        b r7
```

```
% programme principal
@main   mov r0, #3
        mov r1, #4
        mov r7, #next
        b mult
@next   mov r1, #2
        mov r7, #nexxt
        b mult
@nexxt  mov r2, r0
        ...
```

Fonctions

Problèmes

- la fonction `mult` modifie `r2`

Fonctions

Problèmes

- la fonction `mult` modifie `r2`
- que se passe-t-il si `mult` appelle une autre fonction ?

Fonctions

Problèmes

- la fonction `mult` modifie `r2`
- que se passe-t-il si `mult` appelle une autre fonction ?
- comment préserver la valeur de `r7` ?

Fonctions

Problèmes

- la fonction `mult` modifie `r2`
- que se passe-t-il si `mult` appelle une autre fonction ?
- comment préserver la valeur de `r7` ?
- fonctions récursives ?

Fonctions

Problèmes

- la fonction `mult` modifie `r2`
- que se passe-t-il si `mult` appelle une autre fonction ?
- comment préserver la valeur de `r7` ?
- fonctions récursives ?

Solution

Fonctions

Problèmes

- la fonction `mult` modifie `r2`
- que se passe-t-il si `mult` appelle une autre fonction ?
- comment préserver la valeur de `r7` ?
- fonctions récursives ?

Solution

- on utilise une zone de mémoire pour sauvegarder les registres

Fonctions

Problèmes

- la fonction `mult` modifie `r2`
- que se passe-t-il si `mult` appelle une autre fonction ?
- comment préserver la valeur de `r7` ?
- fonctions récursives ?

Solution

- on utilise une zone de mémoire pour sauvegarder les registres
- structure LIFO (Last In First Out) = pile

La pile

Principe

- zone de mémoire de taille variable

La pile

Principe

- zone de mémoire de taille variable
- les données sont ajoutées au sommet (push)

La pile

Principe

- zone de mémoire de taille variable
- les données sont ajoutées au sommet (push)
- les données sont enlevées du sommet (pop)

La pile

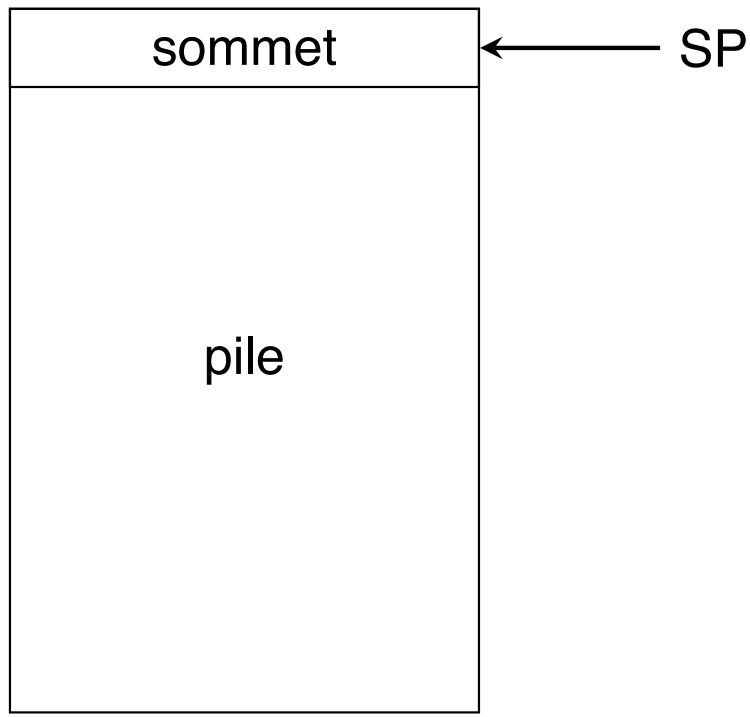
Principe

- zone de mémoire de taille variable
- les données sont ajoutées au sommet (push)
- les données sont enlevées du sommet (pop)
- le sommet est indiqué par le pointeur de pile (SP = *Stack Pointer*)

La pile

Principe

- zone de mémoire de taille variable
- les données sont ajoutées au sommet (push)
- les données sont enlevées du sommet (pop)
- le sommet est indiqué par le pointeur de pile (SP = *Stack Pointer*)



La pile

Opérations

Push (empiler)

La pile

Opérations

Push (empiler)

- on incrémente le pointeur de pile

La pile

Opérations

Push (empiler)

- on incrémente le pointeur de pile
- on écrit la nouvelle donnée au sommet

La pile

Opérations

Push (empiler)

- on incrémente le pointeur de pile
- on écrit la nouvelle donnée au sommet

Pop (dépiler)

La pile

Opérations

Push (empiler)

- on incrémente le pointeur de pile
- on écrit la nouvelle donnée au sommet

Pop (dépiler)

- on lit la donnée au sommet de la pile

La pile

Opérations

Push (empiler)

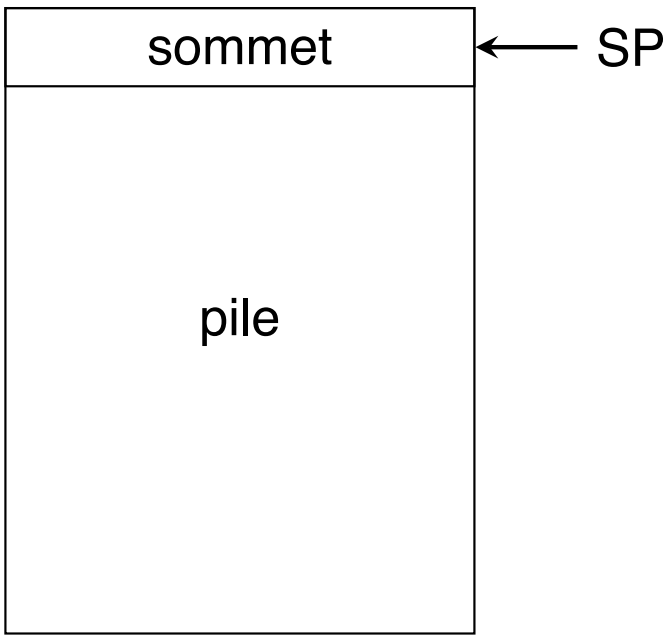
- on incrémente le pointeur de pile
- on écrit la nouvelle donnée au sommet

Pop (dépiler)

- on lit la donnée au sommet de la pile
- on décrémente le pointeur de pile

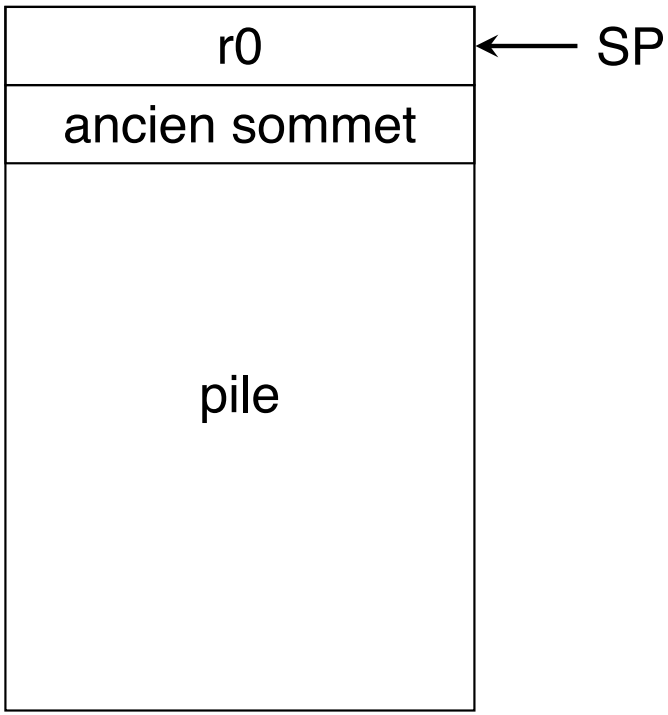
La pile

Push r0



La pile

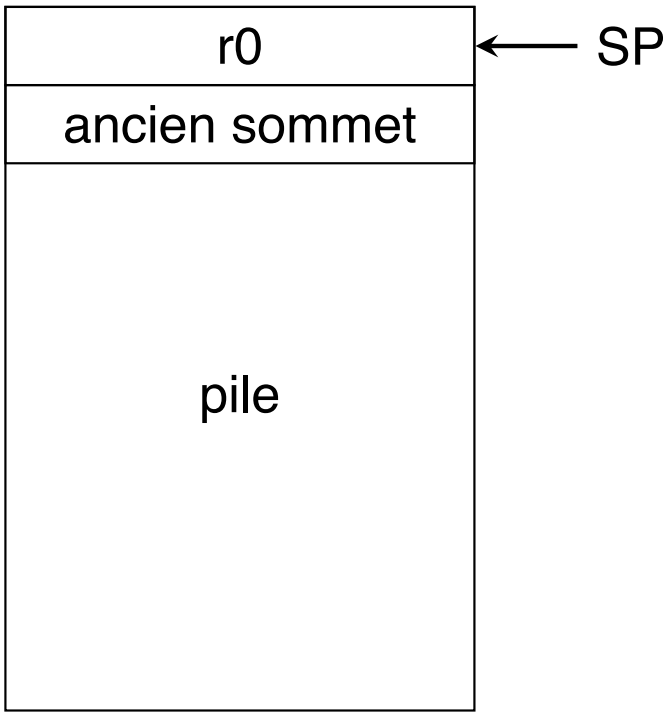
Push r0



La pile

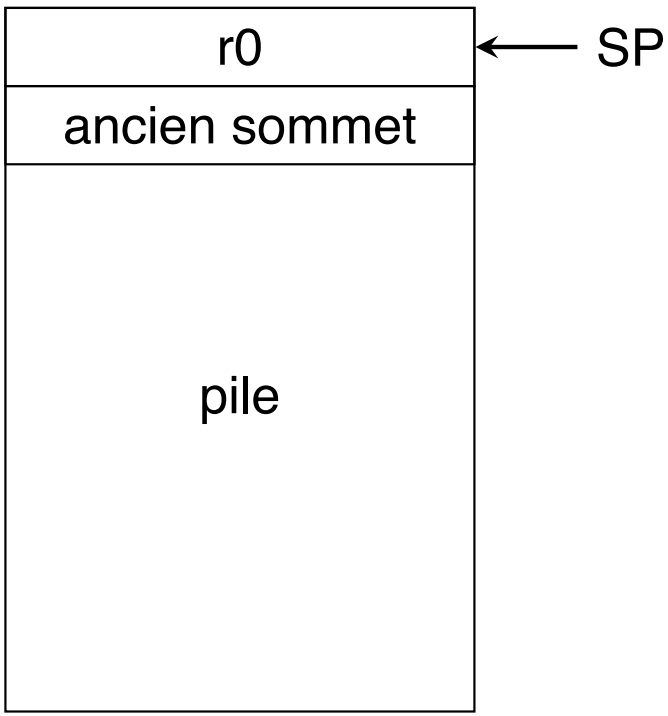
Push r0

Pop r0

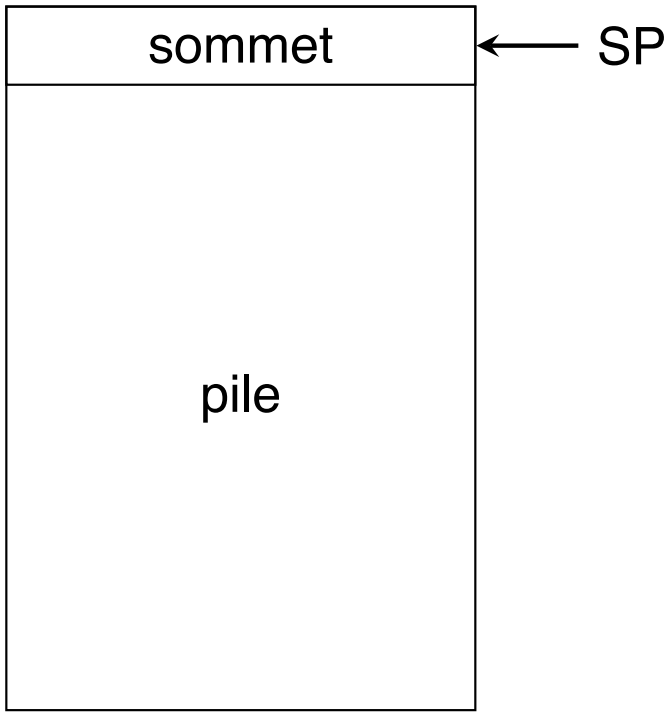


La pile

Push r0

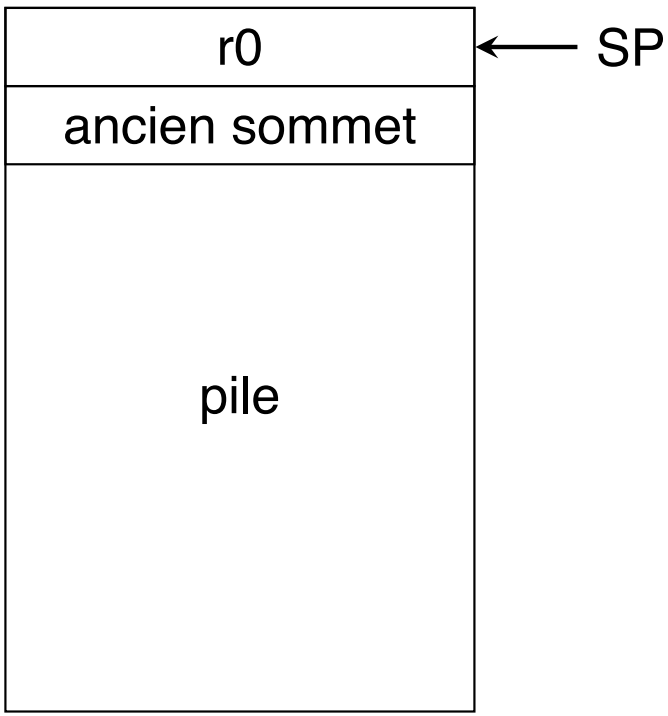


Pop r0

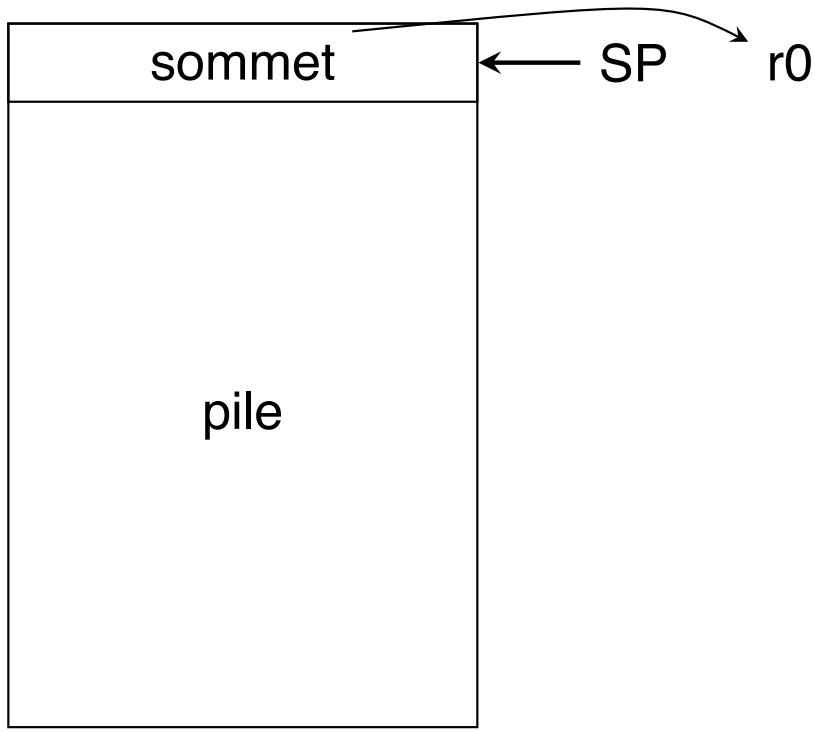


La pile

Push r0

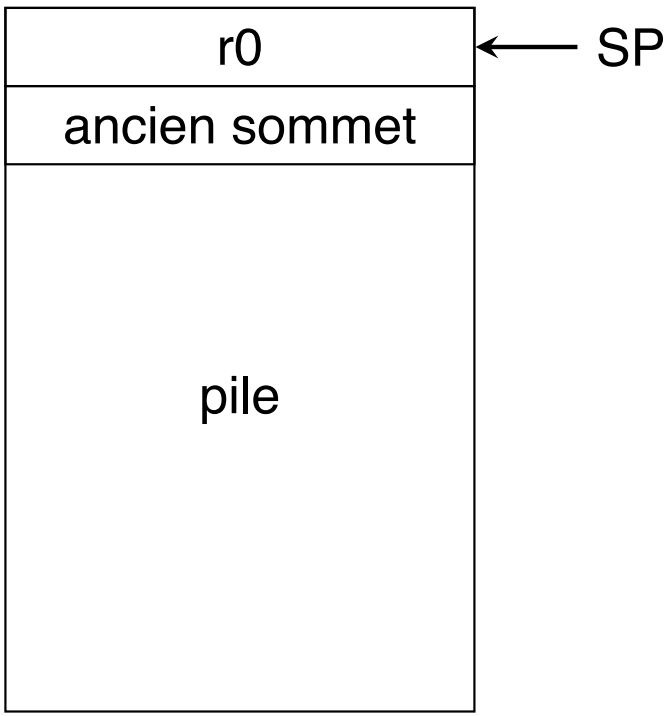


Pop r0

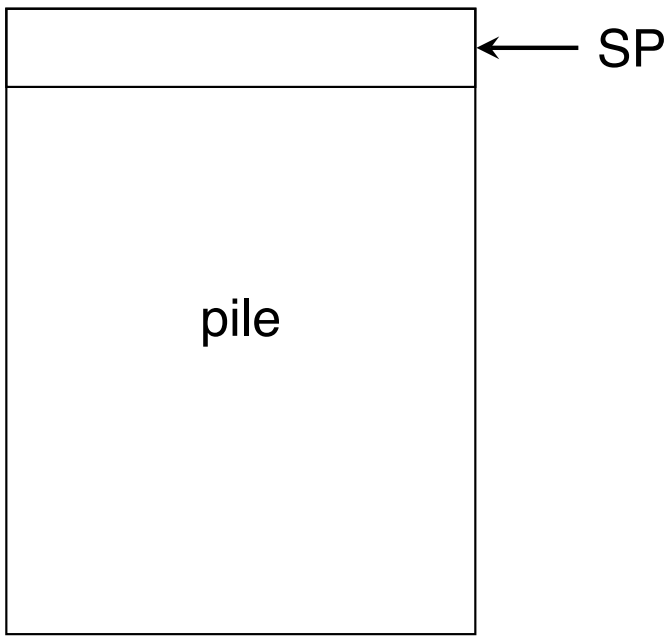


La pile

Push r0



Pop r0



Fonctions

Avec pile

```
% r0 <- r0 * r1
@mult   push r2
        mov r2, #0
@loop   cmp r0, #0
        beq fin
        add r2, r2, r1
        sub r0, r0, #1
        b loop
@fin    mov r0, r2
        pop r2
        b r7
```

```
% programme principal
@main   mov sp, #stack
        mov r0, #3
        mov r1, #4
        mov r7, #next
        b mult
@next   mov r1, #2
        mov r7, #nexxt
        b mult
@nexxt  mov r2, r0
        ...
@stack  rmw 1
```

Fonctions récursives

Avec pile

```
% r0*r1 =
%   0 si r0 = 0
%   r1+(r0-1)*r1 sinon
@mult   push r2
        mov  r2, #0
@loop   cmp  r0, #0
        beq  fin
        sub  r0, r0, #1
        push r7
        mov  r7, #mlnxt
        b    mult
@mlnxt  pop  r7
        add  r2, r0, r1
@fin    mov  r0, r2
        pop  r2
        b    r7
```

```
% programme principal
@main   mov  sp, #stack
        mov  r0, #3
        mov  r1, #4
        mov  r7, #next
        b    mult
@next   mov  r1, #2
        mov  r7, #nexxt
        b    mult
@nexxt  mov  r2, r0
        ...
@stack  rmw  1
```

Fonctions

Instruction `bl`

`bl rx, adresse` :

- copie le PC dans `rx`
- saute à `adresse`
- `bl` = *Branch And Link*

Fonctions

Instruction `bl`

`bl rx, adresse` :

- copie le PC dans `rx`
- saute à `adresse`
- `bl` = *Branch And Link*

Utilisation

Au lieu de :

```
        mov r7, #next
        b func
@next   ...
```

on écrit :

```
        bl r7 func
```

Pile et tas

Pile

La pile contient :

Pile et tas

Pile

La pile contient :

- les paramètres des fonctions

Pile et tas

Pile

La pile contient :

- les paramètres des fonctions
- les adresses de retour

Pile et tas

Pile

La pile contient :

- les paramètres des fonctions
- les adresses de retour
- les registres sauvegardés

Pile et tas

Pile

La pile contient :

- les paramètres des fonctions
- les adresses de retour
- les registres sauvegardés
- les variables locales

Pile et tas

Pile

La pile contient :

- les paramètres des fonctions
- les adresses de retour
- les registres sauvegardés
- les variables locales

Tas

La tas contient :

Pile et tas

Pile

La pile contient :

- les paramètres des fonctions
- les adresses de retour
- les registres sauvegardés
- les variables locales

Tas

La tas contient :

- les données allouées dynamiquement :

Pile et tas

Pile

La pile contient :

- les paramètres des fonctions
- les adresses de retour
- les registres sauvegardés
- les variables locales

Tas

La tas contient :

- les données allouées dynamiquement :
  - tableaux

Pile et tas

Pile

La pile contient :

- les paramètres des fonctions
- les adresses de retour
- les registres sauvegardés
- les variables locales

Tas

La tas contient :

- les données allouées dynamiquement :
  - tableaux
  - chaînes de caractères

Pile et tas

Pile

La pile contient :

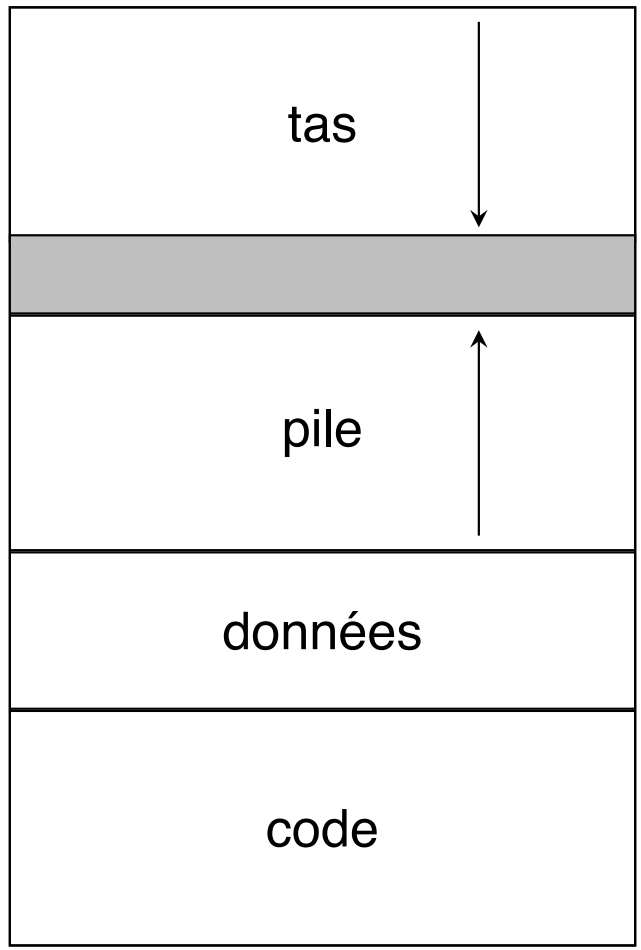
- les paramètres des fonctions
- les adresses de retour
- les registres sauvegardés
- les variables locales

Tas

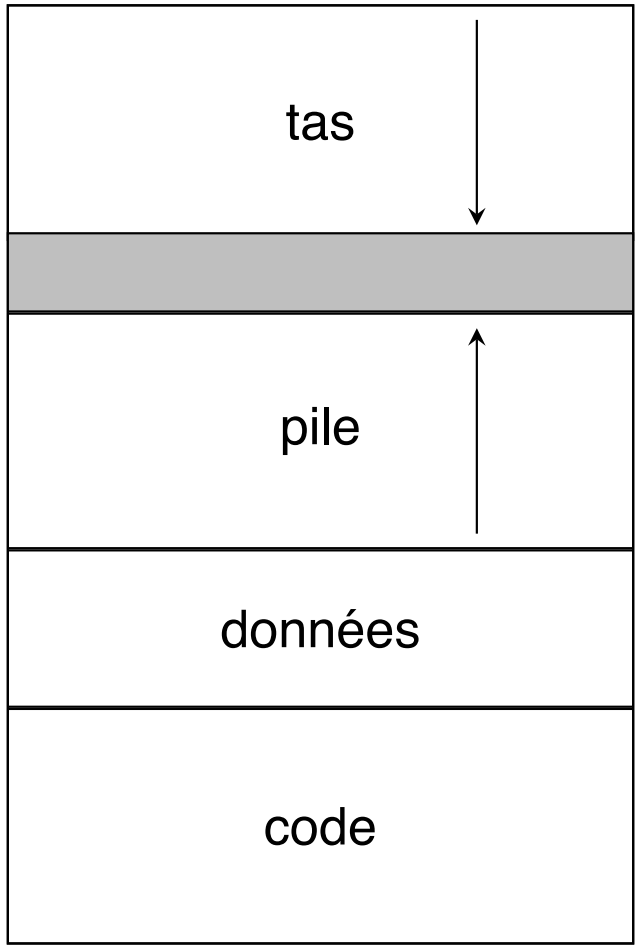
La tas contient :

- les données allouées dynamiquement :
  - tableaux
  - chaînes de caractères
  - objets

Topographie mémoire



Topographie mémoire



Si la pile rencontre le tas $\Rightarrow$ stack overflow !

Suite...

Mémoires