UE23 — Architecture des ordinateurs et système d'exploitation

C. Nguyen, J. Razik

2024–2025

Le processeur

Statistiques d'utilisation du jeu d'instruction

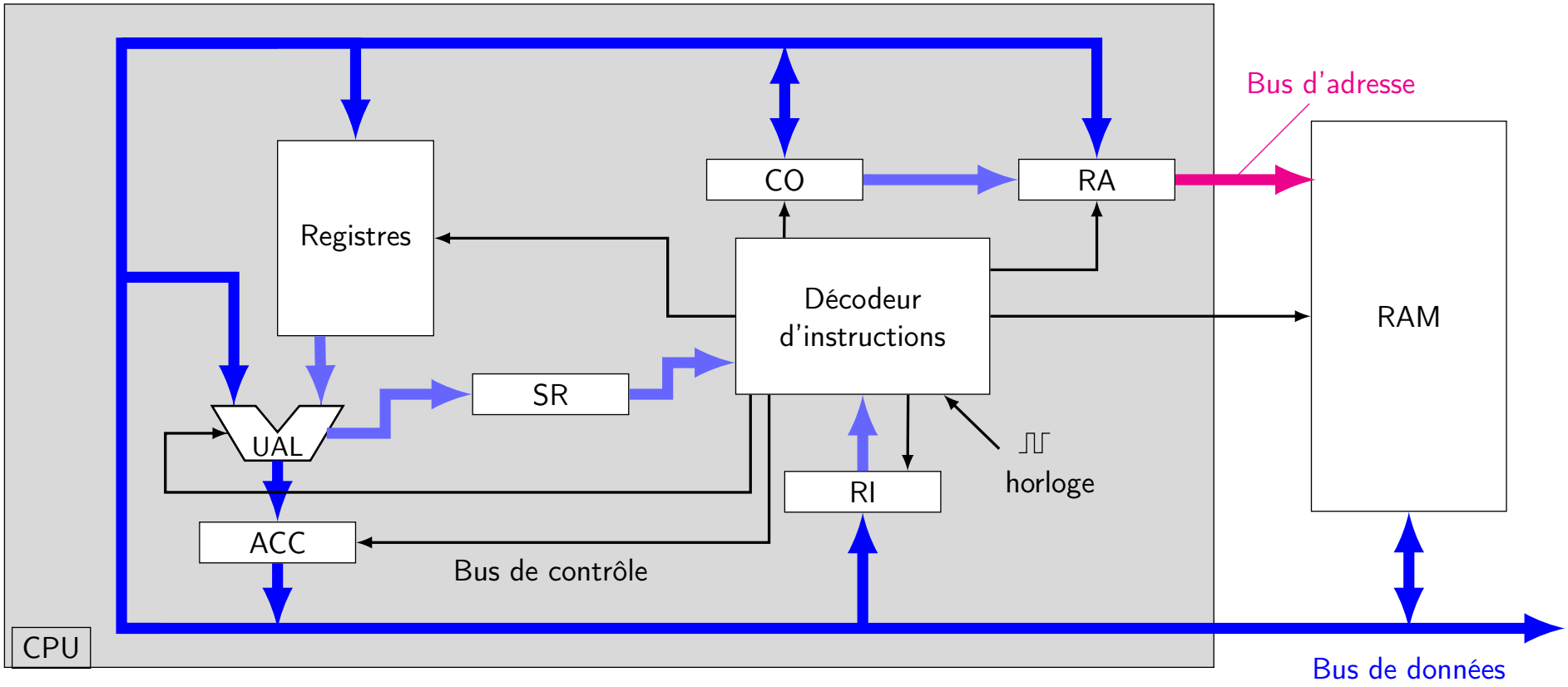
Type d'instruction	Taux d'utilisation
Mouvement de données	45,5 %
Branchement	27,5 %
Comparaison	12 %
Arithmétique	8 %
Logique	3,5 %
Décalage	1,5 %
Autre	2%

Cycle d'exécution d'une instruction

- Recherche de l'instruction en mémoire (*Instruction Fetch*),
  - récupération de l'instruction à l'adresse donnée par le Compteur Ordinal,
  - placement du l'instruction dans le registre d'instruction pour le décodeur d'instructions,
  - incrémentation du Compteur Ordinal,
- Décodage,
  - récupération des opérandes (calcul des adresses effectives),
  - découpage en séquence de microinstructions,
  - transfert de chaque microinstruction sur le bus de contrôle,
  - compteur de microinstructions (équivalent du Compteur Ordinal mais pour les microinstructions),
- Exécution
  - opération (UAL)
  - sauvegarde du résultat

Éléments nécessaires à l'exécution

- Le compteur ordinal (CO),
- Le registre de code condition (*SR – status register*),
- Le registre d'instruction (RI),
- Le registre d'adresse (RA),
- L'Unité Arithmétique et Logique (UAL),
- Des registres généraux,
- Le décodeur d'instruction,
- Une horloge,
- Le bus d'adresse,
- Le bus de données,
- Le bus de contrôle.


Bus d'adresse
CO
RA
Registres
Décodeur
d'instructions
RAM
SR
UAL
RI
horloge
ACC
Bus de contrôle
CPU
Bus de données

Décodeur d'instructions

Découpage de l'instruction processeur (assembleur) séquence de microinstructions contrôlant les organes du CPU.

- Utilisation du bus de contrôle,
- Cadencé par l'horloge (cycle),
- Présence d'un contrôleur de séquence (quelle étapes dans la séquence de microinstructions),

Réalisation du décodeur d'instructions

- Câblée
  - Réalisation d'un circuit séquentiel complexe pour chaque instruction,
  - Rapide,
  - Non évolutif.
- microprogrammée
  - Proposé par Wilkes dès 1951,
  - Utilisation d'un microprogramme (*firmware*),
  - Le code opération est l'adresse de la première microinstruction associée,
  - Peut être reprogrammable, évolutif.

Optimisations

- lecture anticipée de plusieurs instructions,
- utilisation d'un cache.
- branchement prédictif,
- réordonnement des instructions ou microsinstructions.

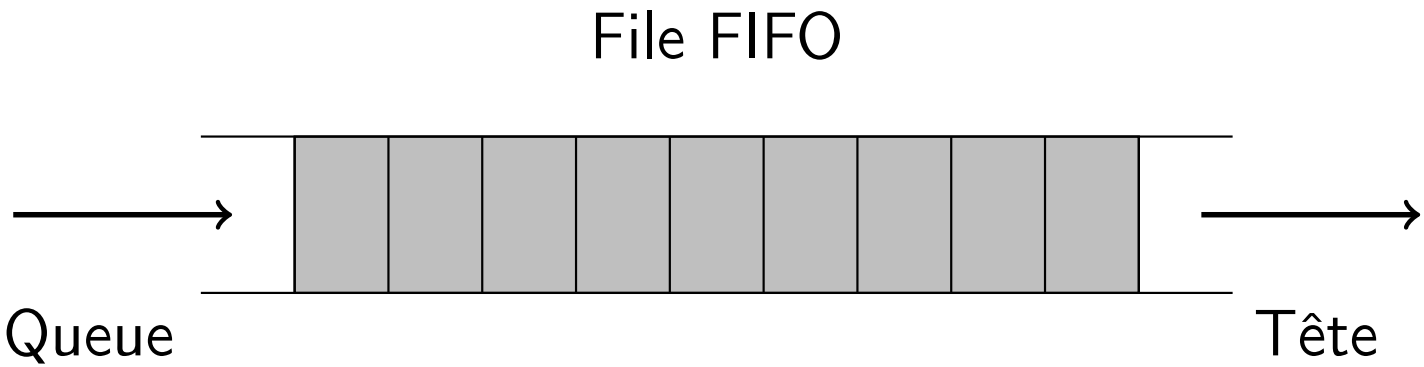
File FIFO – *First In First Out*

Structure de données définie par 2 éléments :

- la tête de file,
- la queue de file.

Fonctionnement

- on ajoute un élément en queue,
- on prélève un élément en tête.



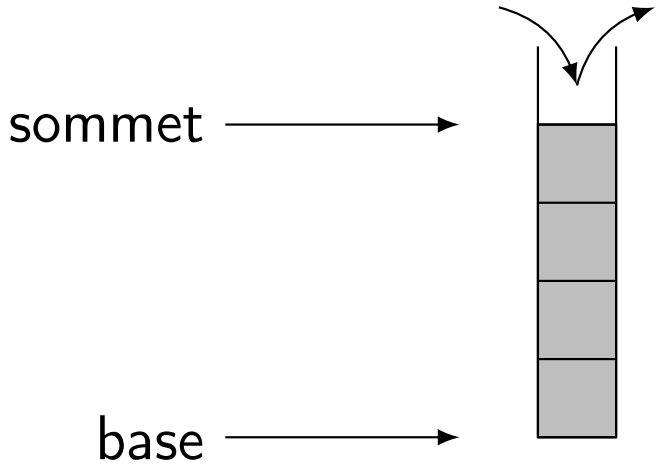
La Pile (*stack*)

Structure de données définie par 2 éléments :

- la base de la pile,
- le sommet de la pile.

Fonctionnement

- on ajoute un élément au sommet,
- on prélève un élément au sommet,
- LIFO – *Last In First Out*.



La pile en mémoire

La pile évolue

- du bas vers le haut,
- des adresses fortes au adresses faibles.

Manipulation via le registre SP – Stack Pointer

- pointeur/flotteur indiquant le sommet de la pile,
- adresse du dernier élément sur la pile.

Mode d'adressage spécifique, par exemple en 68000

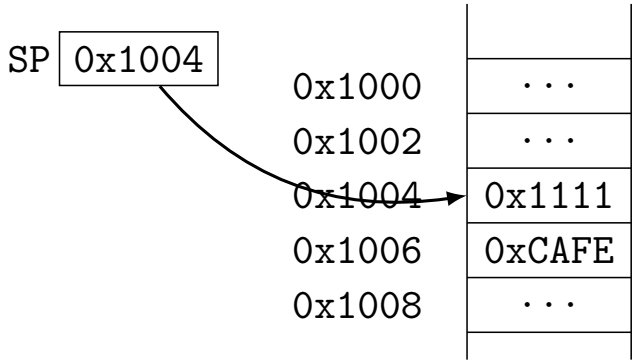
- pour l'ajout au sommet : `-(A7)` ou `-(SP)`,
- pour le prélèvement au sommet : `(A7)+` ou `(SP)+`
- **ATTENTION** l'adresse de la pile doit toujours être paire !

Instructions spécifiques, par exemple en 68000

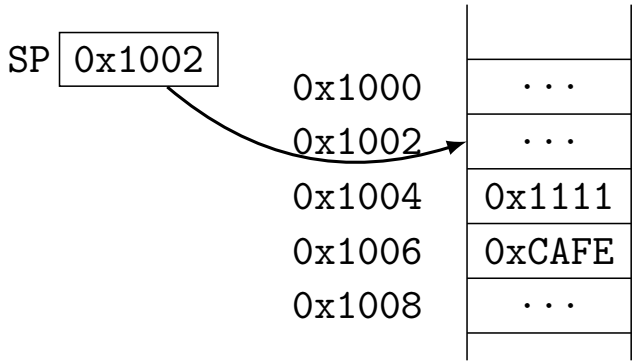
- LINK, PEA, UNLK,
- BSR, JSR, RTD, RTR, RTS,

La pile en mémoire, ajout de la valeur 0x1234 au sommet

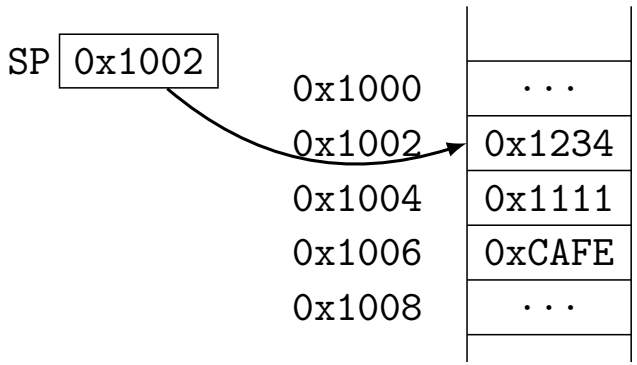
- État de départ de SP et de la mémoire



- Décalage de SP vers le haut du bon nombre d'octets (réservation d'espace), dans notre cas 2 octets

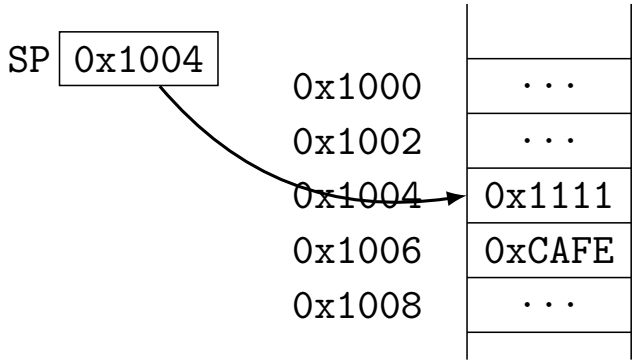


- Placement en mémoire des 2 octets 0x1234.

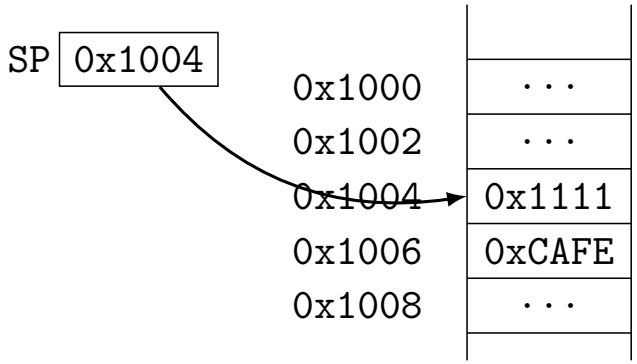


La pile en mémoire, prélèvement de 2 octets au sommet

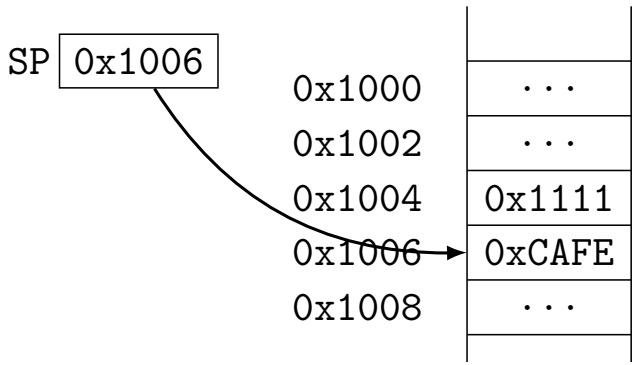
- État de départ de SP et de la mémoire



- Lecture des 2 octets en mémoire (copie) : `0x1111`



- Décalage du pointeur SP vers le bas du nombre d'octets lus, 2 dans cet exemple



Les variables

Dans un programme assembleur les variables peuvent utiliser

- des registres,
  - \+ rapide
  - \- peu de registres disponibles
  - ▸ utilisation de la directive `.equ` en 68000 : `.equ a, d0`
- des espaces mémoire.
  - \- plus lent,
  - \+ autant de variables possibles que l'espace mémoire
  - ▸ utilisation de la directive `.ds` ou `.byte` et des labels
  - ▸ `a: .ds.W 1` ou `a: .word`
- un mélange des deux.

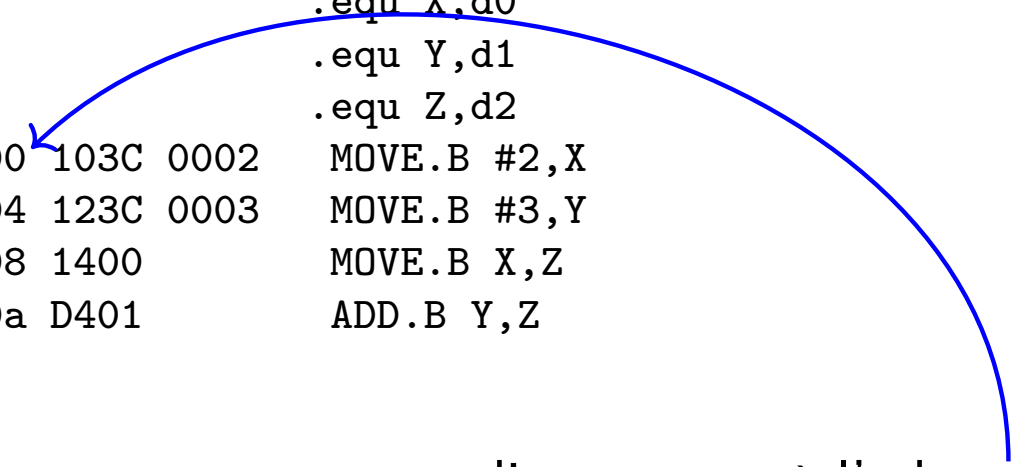
Les variables en registres

Programme en langage de haut niveau : `X = 2; Y = 3; Z = X + Y;`
Traduction en assembleur 68000 en utilisant des registres :

```
.equ X,d0
.equ Y,d1
.equ Z,d2
MOVE.B #2, X   | initialisation de X
MOVE.B #3, Y   | initialisation de Y
MOVE.B X, Z    | copie de X dans Z
ADD.B Y,Z      | ajout de Y a Z
```

Après assemblage

```
1                       .equ X,d0
2                       .equ Y,d1
3                       .equ Z,d2
4 0000 103C 0002        MOVE.B #2,X
5 0004 123C 0003        MOVE.B #3,Y
6 0008 1400             MOVE.B X,Z
7 000a D401             ADD.B Y,Z
```



Le programme commence directement à l'adresse 0, sans réservation pour les variables.

Les variables en mémoire

On ne peut pas mélanger données et instructions

- l'exécution est « linéaire » : l'adresse suivant une instruction est supposée être une instruction (hormis les sauts).
- exemple **incorrect**

```
1 a: .ds.w 1
2     move.b #2,a
3 b: .ds.w 1
4     move.b #3,b
```

2 façon de constituer un programme

- placer volontairement dès le départ toutes les données avant ou après les instructions,

```
1 a: .ds.w 1
2 b: .ds.w 1
3     move.b #2,a
4     move.b #3,b
```

- mélanger instructions et données mais indiquer le type et laisser l'assembleur faire le regroupement.

Les variables en mémoire

Programme en langage de haut niveau : `X = 2; Y = 3; Z = X + Y;`
Traduction en utilisant la mémoire sans segmentation :

```
X: .byte 2    | reservation et initialisation de X
Y: .byte 3    | reservation et initialisation de Y
Z: .ds.B 1    | reservation de Z
.align 2
MOVE.B X, d0  | copie de X dans d0 car ADD ne fait pas EA->EA
ADD.B Y,d0    | ajout de Y a d0
MOVE.B d0,Z   | copie du resultat dans Z
```

Après assemblage

```
  1 0000 02          X:.byte 2
  2 0001 03          Y:.byte 3
  3 0002 00          Z:.ds.B 1
  4 0003 00          .align 2
  5 0004 103A FFFA   MOVE.B X,d0
  6 0008 D03A FFF7   ADD.B Y,d0
  7 000c 13C0 0000   MOVE.B d0,Z
  7      0000
DEFINED SYMBOLS
  {entree standard}:1      .text:00000000 X
  {entree standard}:2      .text:00000001 Y
  {entree standard}:3      .text:00000002 Z
```

Le programme commence à l'adresse 4, les octets précédents sont les variables.

Les variables en mémoire : les détails

- la première instruction `0004 103A FFFC | MOVE.B X,d0` est traduit par l'assembleur en `MOVE.B (d16,PC),d0`
- la donnée est atteinte par déplacement relatif au registre `PC`
  - déplacement de -6 (`0xFFFA`) par rapport au registre PC
  - **MAIS** PC vaut 6 et non 2 :
    - ★ dès le *fetch* de l'instruction, le compteur ordinal est incrémenté de 2.
  - l'adresse effective est `0x0000`.
- la seconde instruction `D03A FFF7 ADD.B Y,d0` est aussi traduite en `ADD.B (d16,PC),d0`
  - déplacement relatif de -9 (`0xFFF7`) par rapport à `PC` qui vaut 8+2=10
  - l'adresse effective est 10-9 = 1 qui est bien l'adresse de Y.
- la troisième instruction utilise cette fois un adressage absolu (direct)
  - l'assembleur utilise temporairement l'adresse `0x00000000`
  - la vraie adresse est conservée dans la table des symboles
    `entrée standard:3 .text:00000002 Z`
  - l'éditeur de lien fera la substitution.

Les appels de fonctions

Un appel de fonction est une rupture d'exécution

- on saute à la première instruction de la fonction
- on revient à l'instruction suivant l'appel

Problème :

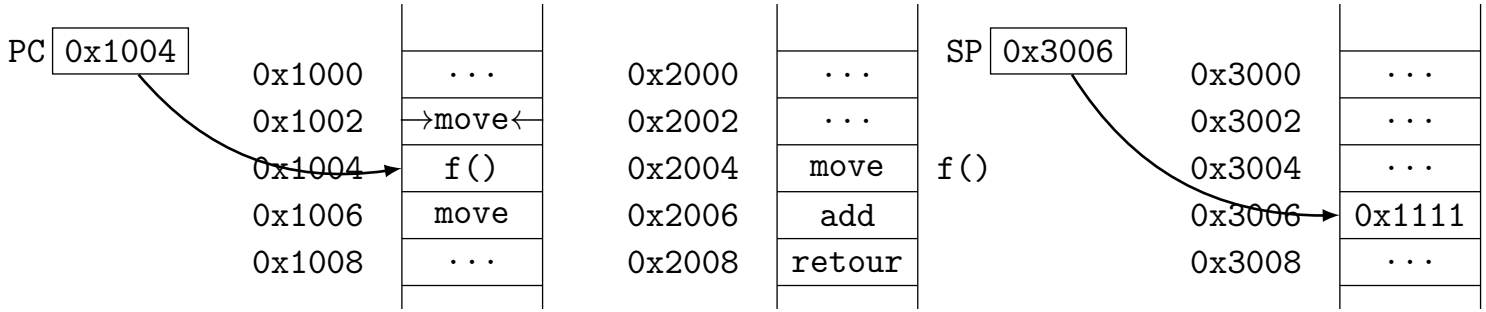
- la fonction peut-être appelée à plusieurs endroits dans le code,
- la fonction peut s'appeler elle-même (récursivité),
- comment revenir à la bonne instruction ?

Solution : mémoriser l'adresse

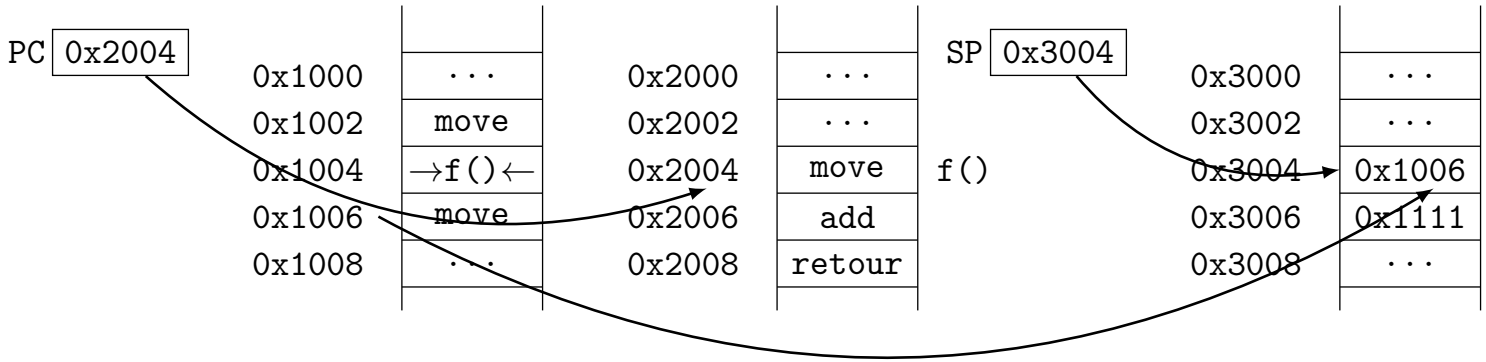
- mémoriser `PC` avant l'appel,
- restaurer `PC` à la fin de la fonction,
- où mémoriser ?
  - un registre $\Rightarrow$ récursivité impossible,
  - une variable mémoire $\Rightarrow$ récursivité impossible,
  - une pile : espace mémoire variable, parfait

Les appels de fonctions

- État initial : exécution de l'instruction `move` d'adresse `0x1002`
  - Le registre `PC` contient l'adresse de la prochaine instruction, `0x1004`
  - La fonction `f` se trouve à l'adresse `0x2004`,
  - Le sommet de la pile est à l'adresse `0x3006`

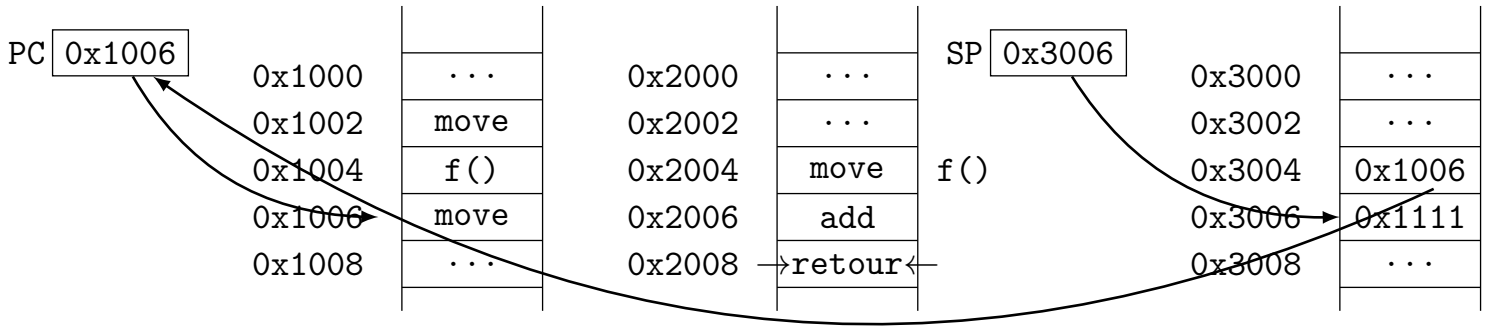


- Exécution de l'appel de `f()`
  - Décalage du sommet de pile,
  - Sauvegarde de `PC` sur la pile,
  - `PC` ← adresse de la fonction,



Les appels de fonctions

- Exécution du retour
  - Copie de l'adresse en sommet de pile dans `PC`,
  - Décalage du sommet de pile,



- La prochaine instruction exécutée sera le `move` d'adresse `0x1006`

Les appels de fonctions

Utilisation d'instructions dédiées, en 68000 :

- Pour l'appel
  - `BSR` – Branch to Sub-Routine :
    - ★ décalage de la pile : `SP` $\leftarrow$ `SP` - 4
    - ★ sauvegarde `PC` sur la pile,
    - ★ `PC` $\leftarrow$ `PC` + déplacement
  - `JSR` – Jump to Sub-Routine :
    - ★ décalage de la pile : `SP` $\leftarrow$ `SP` - 4
    - ★ sauvegarde `PC` sur la pile,
    - ★ `PC` $\leftarrow$ adresse absolue
- Pour le retour de la fonction
  - `RTS` – Return from Sub-routine :
    - ★ `PC` $\leftarrow$ valeur en sommet de pile,
    - ★ décalage de la pile : `SP` $\leftarrow$ `SP` + 4

Le passage des paramètres

Comment passer des paramètres à la fonction ?

- Par les registres
  - Nombre limité,
  - Peut-être déjà utilisés pour autre chose $\rightarrow$ sauvegarde,
- Par la mémoire
  - Pas moyen de faire de la récursion
- Par la pile
  - Plus lent mais pas de limites.

Le passage des paramètres par la pile

On empile

1. les paramètres
2. le compteur ordinal pour le retour

Comment accéder autre paramètres ?

- adresse relative
- nécessite une adresse de référence

Comment choisir l'adresse de référence ?

- doit être une adresse dans la pile
- doit être stable au cours de l'exécution de la fonction

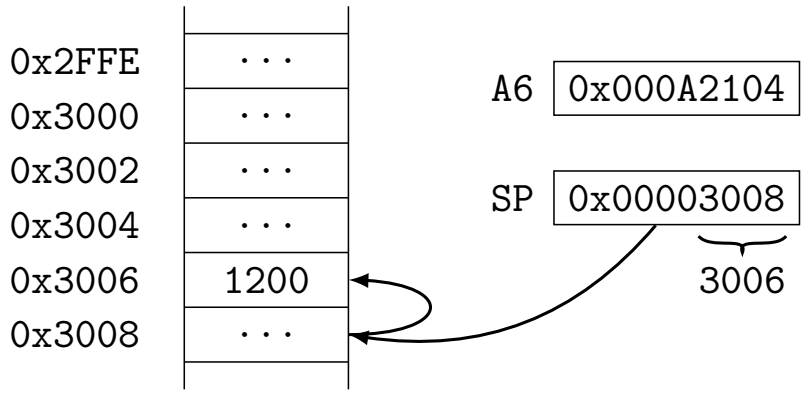
$\Rightarrow$ utilisation d'un registre

$\Rightarrow$ nécessite sa sauvegarde sur la pile avant

Le passage des paramètres par la pile en 68000

Soit une fonction `f` prenant 1 paramètre `pa` (pa).

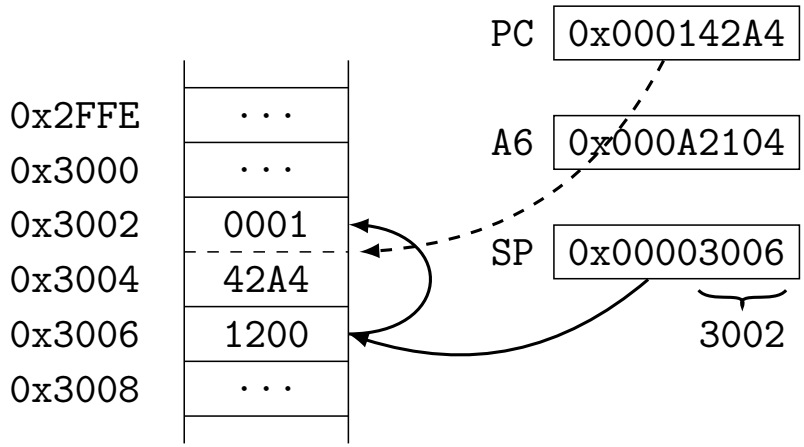
1. On empile le paramètre (de valeur `0x12`)



Le passage des paramètres par la pile en 68000

Soit une fonction `f` prenant 1 paramètre `pa` (`pa`).

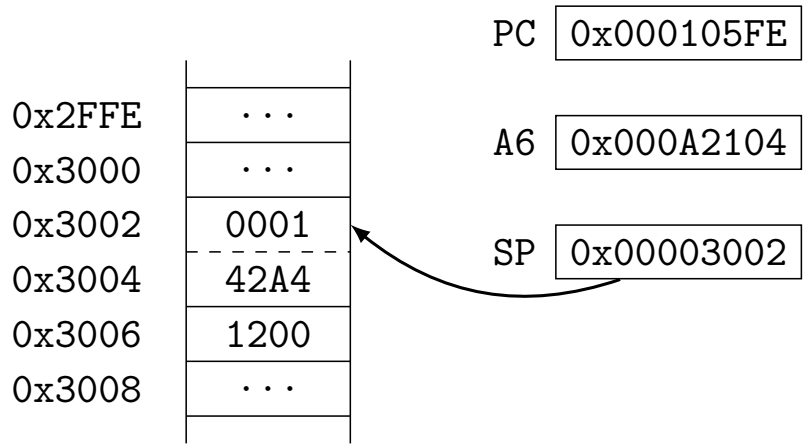
1. On empile le paramètre (de valeur `0x12`)
2. On empile le compteur ordinal



Le passage des paramètres par la pile en 68000

Soit une fonction `f` prenant 1 paramètre `pa` (`pa`).

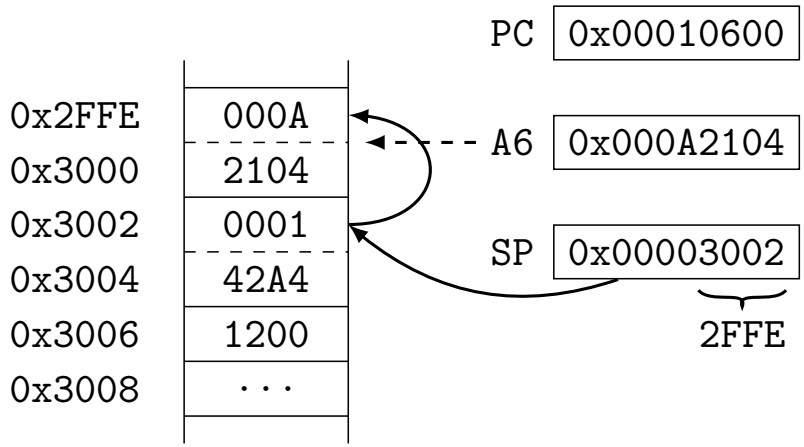
1. On empile le paramètre (de valeur `0x12`)
2. On empile le compteur ordinal
3. On saute à l'adresse de `f` d'adresse `0x105FE`



Le passage des paramètres par la pile en 68000

Soit une fonction `f` prenant 1 paramètre `pa` (`pa`).

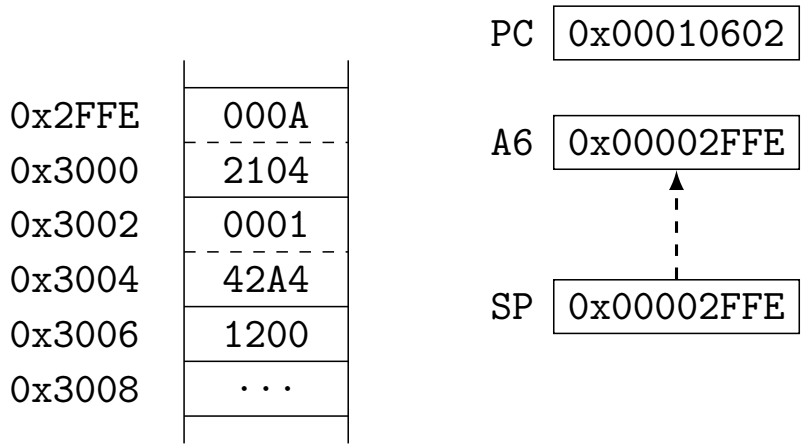
1. On empile le paramètre (de valeur `0x12`)
2. On empile le compteur ordinal
3. On saute à l'adresse de `f` d'adresse `0x105FE`
4. On empile le registre `A6`



Le passage des paramètres par la pile en 68000

Soit une fonction `f` prenant 1 paramètre `pa` (pa).

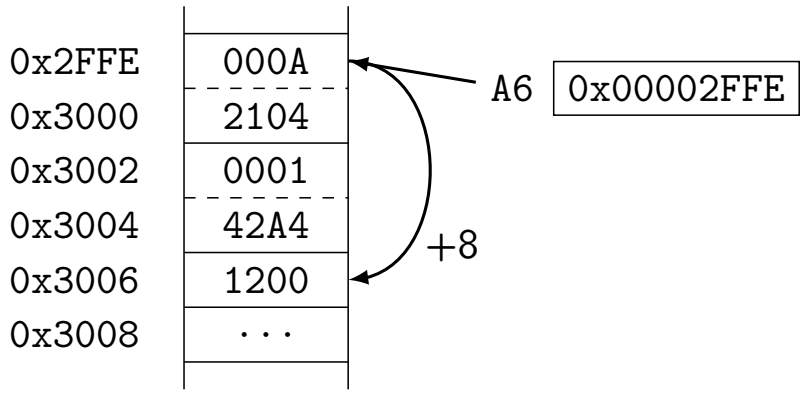
1. On empile le paramètre (de valeur `0x12`)
2. On empile le compteur ordinal
3. On saute à l'adresse de `f` d'adresse `0x105FE`
4. On empile le registre `A6`
5. On copie l'adresse du sommet de la pile dans `A6`



Le passage des paramètres par la pile en 68000

Soit une fonction `f` prenant 1 paramètre `pa` (`pa`).

1. On empile le paramètre (de valeur `0x12`)
2. On empile le compteur ordinal
3. On saute à l'adresse de `f` d'adresse `0x105FE`
4. On empile le registre `A6`
5. On copie l'adresse du sommet de la pile dans `A6`



Accès au paramètre

- par adressage relatif `(d16, A6)`.
- déplacement pour `pa` $= 4 + 4 = 8$
- peut le définir par une constante (`.equ pa, 8`)
- exemple : `move.b (pa, A6),d0`

Le passage des paramètres par la pile : le retour

Il ne suffit plus de faire un retour, il faut

1. restaurer le sommet de la pile sauvé dans A6
   - `move.l A6,SP`
2. restaurer la valeur de A6
   - `move.l (SP)+, A6`
3. faire le retour
   - `rts`
4. nettoyer la pile des paramètres
   - on a empiler 1 paramètre qui a pris 2 octets
   - `add.l #2, SP`

Les variables locales

Où placer les variables locales d'une fonction ?

- dans des registres fixes
  - peu de registres,
  - peut-être déjà utilisés $\Rightarrow$ sauvegarde des registres avant,
  - récursivité impossible.
- en mémoire
  - mêmes problèmes et limitations.
- sur la pile
  - plus lent que les registres,
  - pas d'autres limitations.

Les variables locales par la pile

On empile

1. les paramètres,
2. le compteur ordinal,
3. la sauvegarde du registre de base (A6),
4. les variables locales de la fonction
   - pas forcément de valeur à empiler
   - réservation de place suffit
   - par décalage du sommet de pile

Comment accéder aux variables locales ?

- adresse relative
- nécessite une adresse de référence

Comment choisir l'adresse de référence ?

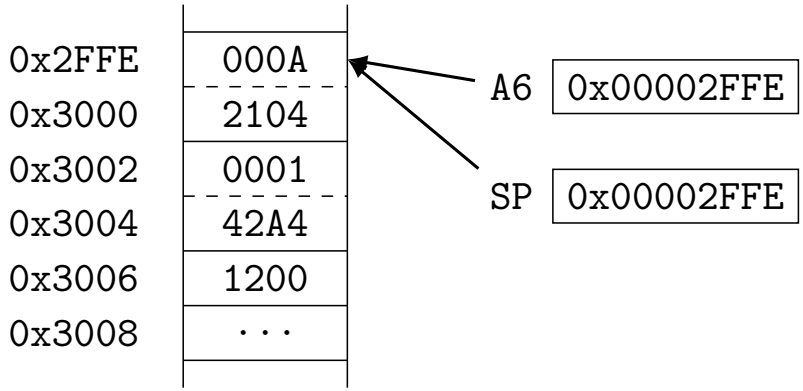
- doit être une adresse dans la pile
- doit être stable au cours de l'exécution de la fonctionnaire

$\Rightarrow$ on a déjà une : l'adresse dans le registre de base ! (A6)

Les variables locales par la pile

Soit `f` une fonction prenant 1 paramètre `a` (`byte`) et utilisant deux variables locales `b` (`byte`) et `c` (`word`)

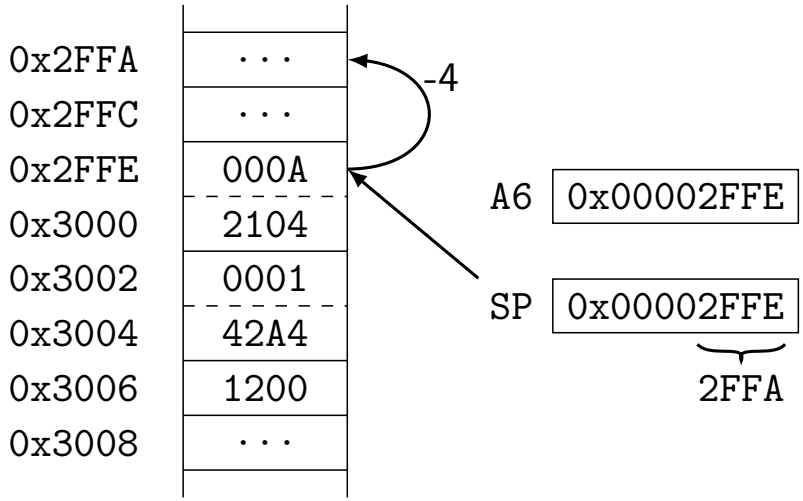
1. même chose que pour les paramètres par la pile



Les variables locales par la pile

Soit `f` une fonction prenant 1 paramètre `a` (`byte`) et utilisant deux variables locales `b` (`byte`) et `c` (`word`)

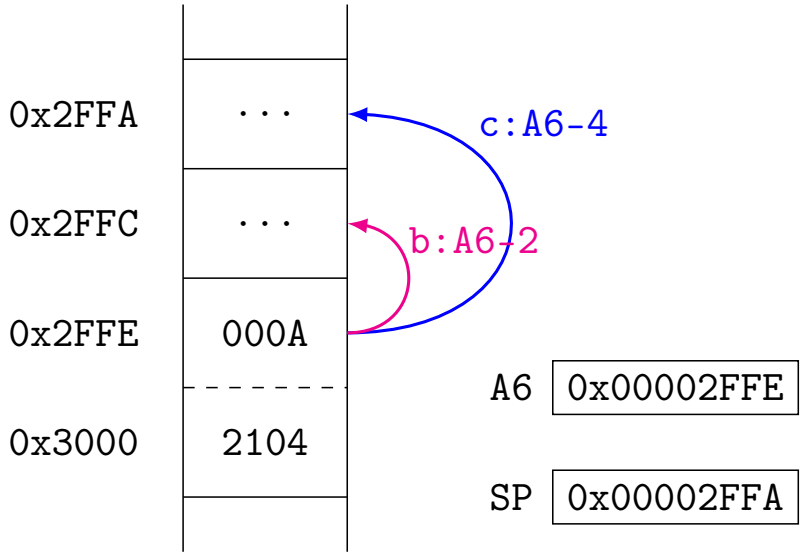
1. même chose que pour les paramètres par la pile
2. réservation de la place sur pile pour les variables :
   - 2 (byte sur pile) + 2 (word) = 4 octets
   - déplacement vers les adresses basses $\Rightarrow$ -4
   - `add.l #-4, SP`



Les variables locales par la pile

Soit `f` une fonction prenant 1 paramètre `a` (`byte`) et utilisant deux variables locales `b` (`byte`) et `c` (`word`)

1. même chose que pour les paramètres par la pile
2. réservation de la place sur pile pour les variables :
   - 2 (byte sur pile) + 2 (word) = 4 octets
   - déplacement vers les adresses basses $\Rightarrow$ -4
   - `add.l #-4, SP`



Accès aux variables

- par adressage relatif `d16,A6)`
- déplacement pour `b` : -2, pour `c` : -4
- peut les définir par une constante
  - `.equ b,-2` et `.equ c,-4`
- exemple : `move.b #0,(b,A6)`

Les valeurs de retour

Comment récupérer la ou les valeurs de retour d'une fonction ?

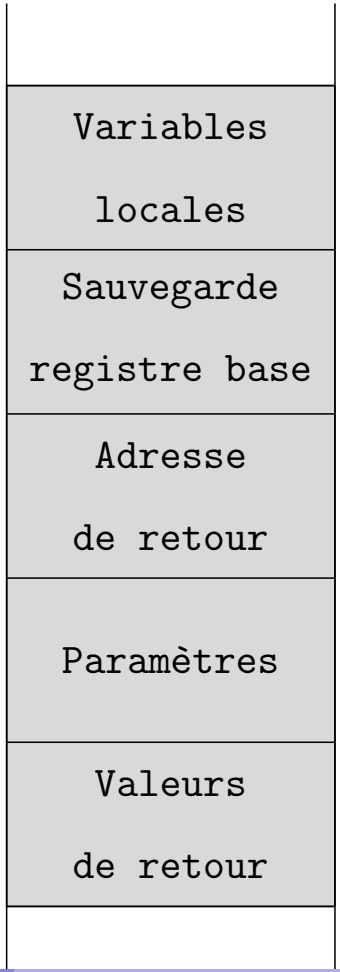
- dans des registres fixes
  - peu de registres,
  - souvent une seule valeur de retour,
  - sauvegarde des registres avant l'appel,
- par la pile
  - réservation sur la pile avant les paramètres avant l'appel,
    - si valeur de retour de type `word`
    - exemple : `add.l -#2, SP`
  - accès par adressage relatif
    - par rapport au registre de base (`A6`)
    - déplacement = base + sauvegarde base + retour de fonction + taille des paramètres
    - déplacement = 4 + 4 + 4 = 12 (si taille des paramètres = 4)
    - exemple : `move.w d1,(#12,A6)`
  - récupération de la valeur et nettoyage de la pile après le retour,
    - nettoyage de la pile d'un coup : paramètres et valeur de retour
    - exemple : `add.l #6, SP`

Pile d'exécution - *stack*

La pile – *stack* – va principalement servir à conserver les données liées à l'appel de routines (fonctions, procédures) encore actives (non terminées). Le sommet de la pile concerne la dernière routine actuellement en cours d'exécution.

Une *frame* (*stack frame*) est composée

- des valeurs de retour,
- des paramètres,
- de l'adresse de retour,
- de la sauvegarde du registre de base,
- des variables locales.



Le tas (*heap*)

Le tas est une autre portion de mémoire pour des données qui ne seront connues qu'au moment de l'exécution.
Pendant l'exécution, le programme doit :

- réserver de l'espace,
- libérer l'espace réservé.

On appelle ce principe l'*allocation dynamique*.

Exemple : dessiner une courbe passant par des points placés à la souris

- on ne connaît pas à l'avance le nombre de points que l'utilisateur placera,
- mais on doit tous les connaître pour les relier

$\Rightarrow$ ils doivent être en mémoire

C'est le système d'exploitation qui gère l'organisation du *tas*.
Régulièrement il appellera le *ramasse-miettes* (*garbage collector*) pour remettre les espaces libérés dans ceux disponibles.