Les systèmes d'exploitation

Wadoud Bousdira[1]
wadoud.bousdira@univ-orleans.fr

[1]LIFO, University of Orléans
Orléans, France

Orléans, 2023

Le langage assembleur

Langage machine

Un processeur exécute des programmes écrits en langage machine

- un programme écrit en langage machine est une suite contigüe de mots binaires (16, 32 ou 64 bits selon le processeur)
- chaque mot binaire correspond à une instruction du processeur
- cette suite de mots est placée en mémoire pour être exécutée

- l'ensemble des instructions reconnues par un processeur constitue son jeu d'instructions.
- chaque processeur (familles des x86, arm, sparc, ...) dispose de son propre jeu d'instructions.

Le langage assembleur

Jeu d'instructions

On trouve typiquement

- des instructions de calcul arithmétique ou logique (add, mul, and,...)
- des instructions d'accès à la mémoire (load, store, ...)
- des instructions de saut conditionnel ou inconditionnel (beq, jmp, ...)

Le langage machine n'est pas *structuré* :

- il ne dispose pas de structures de contrôle élaborées comme les langages de haut niveau : `while`, `repeat`, `if/then/else`
- les expressions complexes (ex. $x + 2 \times (y + z)$ doivent être décomposées

On utilise un compilateur pour transformer un programme écrit dans un langage de haut niveau en assembleur.

Jeu d'instructions

- les instructions de calcul travaillent sur des registres (en nombre limité) ou directement sur la mémoire
- les instructions d'accès à la mémoire permettent de déplacer des données entre registres et mémoire
- un registre spécial, le compteur ordinal (`PC : Program counter`) contient l'adresse de la prochaine instruction à exécuter
  - ce compteur est incrémenté de 1 après l'exécution de chaque instruction qui n'est pas un saut
  - les instructions de saut permettent de modifier la valeur de `PC` pour exécuter une autre section de code
  - il n'est pas possible d'accéder directement à `PC`.

Jeu d'instructions

Routines

Des instructions de saut spécifiques permettent d'appeler des routines

- les routines correspondent aux fonctions d'un langage de haut niveau
- l'appel d'une routine est réalisé en effectuant un branchement vers son code
- la différence avec les autres instructions de branchement est qu'il faut pouvoir revenir au point d'appel après l'exécution de la routine.

Jeu d'instructions

Une instruction contient

- un code correspondant à l'instruction à exécuter
- les arguments de l'opération :
  - valeur directe

    ```
    ADD R0, R0, #1 (R0 <- R0 +1)
    ```
  - numéro de registre

    ```
    ADD R0, R0, R1 (R0 <- R0 + R1)
    ```
  - adresse mémoire

    ```
    LDR R0, 0xAFFF, #2 (R0 <- MEMORY[0xAFFF + 0x0002])
    ```

Jeu d'instructions

On distingue deux grandes familles de processeurs en fonction des caractéristiques de leur jeu d'instructions

- `CISC (Complex Instruction Set Computer)`
  - nombre d'instructions élevé
  - peu de registres
  - les instructions peuvent combiner mémoire et calcul

  ex. Intel x86, Motorola 68000
- `RISC (Reduced Instruction Set Computer)`
  - peu d'instructions
  - beaucoup de registres
  - les instructions d'accès à la mémoire et de calcul sont séparées
  - les instructions de calcul travaillent sur les registres

  ex. ARM, SPARC, MIPS

Les processeurs CISC

- leur conception a été favorisée par le manque de mémoire et sa lenteur. Nécessite de disposer d'instructions réalisant des tâches complexes
- le faible nombre de registres conduisait à utiliser des instructions de calcul capables d'opérer directement sur la mémoire
- les instructions complexes présentent l'inconvénient de ralentir la fréquence d'horloge 
- En pratique, peu d'instructions complexes sont utilisées par les compilateurs.

Les processeurs RISC

- conçus pour restreindre le nombre d'instructions et accroître la fréquence d'horloge
- le calcul nécessite de déplacer les opérandes dans des registres. Par conséquent, il faut un nombre de registres important 👎
  Rendu possible par la densification des circuits et la réduction du coût de fabrication.

Le langage assembleur

Un exemple de programme (Wikipedia)

```
0000000 0000 0001 0001 1010 0010 0001 0004 0128
0000010 0000 0016 0000 0028 0000 0010 0000 0020
0000020 0000 0001 0004 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0010 0000 0000 0000 0204
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
0000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
0000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
0000080 8888 8888 8888 8888 288e be88 8888 8888
0000090 3b83 5788 8888 8888 7667 778e 8828 8888
00000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
00000f0 8888 8888 8888 8888 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```

Jeu d'instructions

Mnémoniques

Pour rendre plus lisbible les programmes écrits en binaire.

instruction	code binaire	mnémonique
addition (`r2 <- r2+r1`)	00000110	add r1 r2
soustraction (`r3 <- r2-r3`)	00011011	sub r2 r3
multiplication (`r2 <- r1*r2`)	00100110	mul r1 r2

Assembleur

Le langage obtenu à partir des mnémoniques est appelé assembleur.
On appelle aussi assembleur l'outil qui traduit ce langage en langage machine (binaire).

Le langage assembleur

Le processeur LC3

à but pédagogique développé par Yale N. Patt et Sanjay J. Patel.

- pas de réalisation physique du processeur
- des simulateurs

Référence : Introduction to Computing Systems : from bits and gates to C/C^{++} & beyond
third edition, McGraw Hill

Le processeur LC3

La mémoire

organisée en mots de 16 bits adressables sur 16 bits (adresses 0000-FFFF)

Les registres généraux

8 registres généraux, nommés R0, R1, ..., R7.
Par convention, R5, R6 et R7 sont réservés à la gestion des routines.

Les registres spécifiques

- `PC` : Program Counter (adresse de la prochaine instruction)
- `IR` : Instruction Register (instruction courante)
- `PSR` : Program Status Register (information sur l'état du processeur)

Le processeur LC3

Le programme

C'est un fichier texte constitué d'une suite de lignes. Chaque ligne peut être :

- une instruction
- une macro
- une directive d'assemblage

Une ligne peut être précédée par une étiquette pour référencer son adresse.

Les constantes

On distingue deux types de constantes :

- les chaînes de caractères, qui ne peuvent apparaître qu'après la directive `.STRINGZ`. Elles sont délimitées par deux caractères "
- les entiers relatifs, précédés d'un x pour une notation héxadécimale et d'un # pour une notation décimale. Ils peuvent être utilisés
  - comme opérande des instructions
  - comme paramètre des directives

Le processeur LC3

Directives d'assemblage

- `.ORIG adresse` : spécifie l'adresse à laquelle doit se trouver le bloc qui suit
- `.END` : termine un bloc d'instructions
- `.FILL` valeur : réserve un mot de 16 bits et le remplit avec la valeur donnée
- `.STRINGZ chaine` : réserve un nombre de mots égal à la longueur de la chaîne de caractères et le remplit avec la chaîne
- `.BLKW nombre` : réserve le nombre de mots de 16 bits passé en paramètre.

Le processeur LC3

Macros

- `HALT` : termine le programme
- `GETC` : lit au clavier un caractère et le place dans l'octet de poids faible de R0
- `OUT` : écrit à l'écran le caractère contenu dans l'octet de poids faible de R0
- `PUTS` : écrit à l'écran la chaîne pointée par R0
- `IN` : lit au clavier un caractère, l'écrit à l'écran et le place dans l'octet de poids faible de R0.

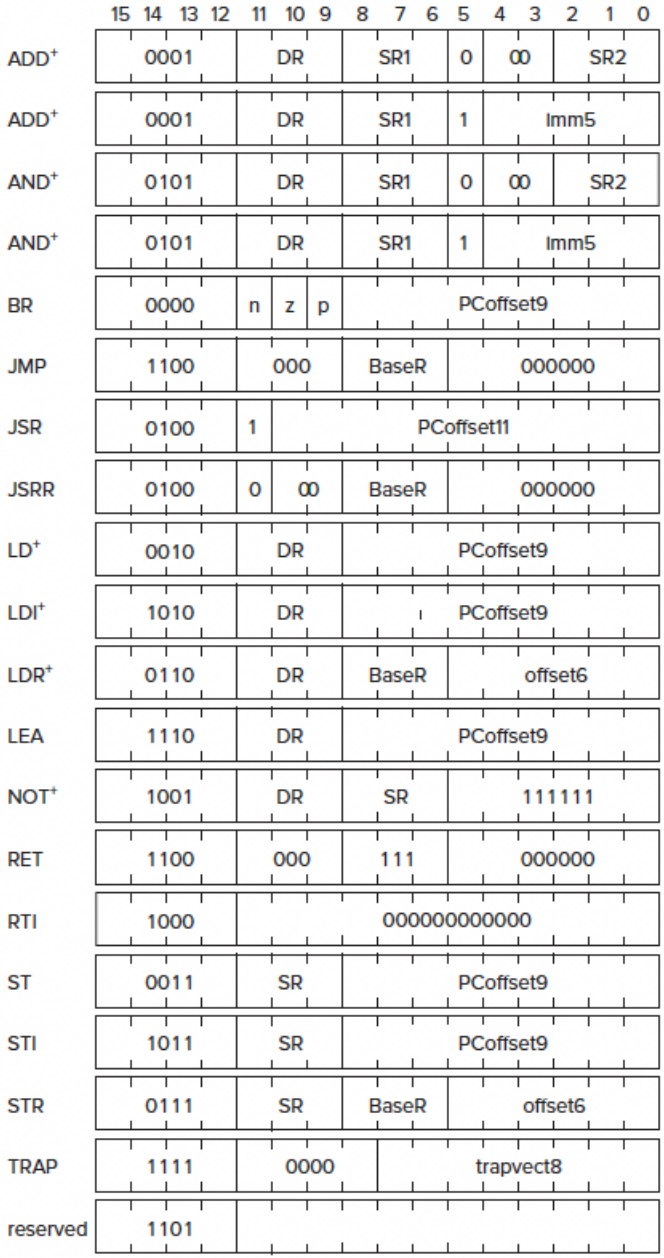
Le processeur LC3

Instructions

Les instructions du LC3 se répartissent en 3 catégories

- instructions arithmétiques et logiques : `ADD, AND, NOT`
- instructions de chargement et déchargement
  - `LD, ST` : load et store
  - `LDR, STR` : load et store avec adressage relatif
  - `LEA` : load effective address
- instructions de branchement
  - `BR` : branch
  - `JSR` : jump subroutine
  - `RET` : retour (de routine)

Le processeur LC3



Le processeur LC3

Note : + indique les instructions qui modifient les codes condition.

Addition

- Syntaxe : `ADD RD, RA, RB`
- `RD <- RA + RB`
- `PC <- PC + 1`

Sur le même principe :

- AND RD, RA, RB `RD <- RA && RB`
- OR RD, RA, RB `RD <- RA || RB`
- NOT RD, RA `RD <- !RA`

Le processeur LC3

LD (Load)

- Syntaxe : `LD DR, label9`
- l'étiquette `label`, codée sur 9 bits, désigne une adresse mémoire `adr` à laquelle se situe une valeur `val`
- charge dans le registre `DR` la valeur `val` : `DR <- MEM[adr]`
- `PC <- PC + 1`

Les étiquettes permettent de désigner une adresse par un nom.

Exemple :

```
    LD R1, a
    LD R2, b
    ADD R3, R1, R2
a   .FILL #10
b   .FILL #15
```

ST (Store)

- Syntaxe : `ST SR, label9`
- l'étiquette `label`, codée sur 9 bits, désigne une adresse mémoire `adr`
- charge à l'adresse `adr` la valeur `val` du registre `SR` : `MEM[adr] <- SR`
- `PC <- PC + 1`

Exemple :

```
    LD R1, a
    LD R2, b
    ADD R3, R1, R2
    ST R3, b
a   .FILL #10
b   .FILL #15
```

Le processeur LC3

Limitation de LD et ST

- Les instructions du L3C sont codées sur 16 bits (pour tenir dans le registre IR)
- pour les instructions LD et ST, le label est contenu dans ces 16 bits
- ce mode d'adressage, dit direct, impose que l'adresse mémoire manipulée soit "à proximité" du code exécuté
- on ne peut pas désigner toutes les adresses (codées sur 16 bits).

LDR et STR

Ont les mêmes fonctions que `LD` et `ST` mais l'opérande (l'adresse) est placé dans un registre (sur 16 bits) et peut donc adresser toute la mémoire

- `LDR` charge dans un des registres généraux le mot mémoire à l'adresse égale à la somme du registre de base et d'un offset codé dans l'instruction
- `STR` range le contenu du registre à cette même adresse.

Ce mode d'adressage est appelé adressage relatif. Ces instructions permettent de manipuler

- des données sur la pile (ex. les variables des fonctions)
- des structures de données complexes (ex. tableaux, structures, listes,...)

L'offset permet d'accéder facilement aux différents éléments d'une structure de données.

Le processeur LC3

LDR (Load Register)

- syntaxe : `LDR DR, baseR, Offset6`
- le registre de base `baseR` contient une adresse
- l'offset est codé sur 6 bits
- charge dans le registre `DR` la valeur située à l'adresse obtenue en faisant la somme de `baseR` et de l'offset
- `PC <- PC + 1`

STR (Store Register)

- syntaxe : `STR SR, baseR, Offset6`
- le registre de base `baseR` contient une adresse
- l'offset est codé sur 6 bits
- charge le contenu de `SR` à l'adresse obtenue en faisant la somme de `baseR` et de l'offset
- `PC <- PC + 1`

LEA (Load Effective Address)

L'instruction LEA permet de charger une adresse dans un registre général (utile pour LDR et STR)

- syntaxe : LEA DR, label9
- l'étiquette label codée sur 9 bits, désigne une adresse mémoire adr
- charge dans DR l'adresse mémoire correspondant à l'étiquette
- PC <- PC + 1

Exemple : charger dans R0 le contenu de la mémoire à l'adresse 0xBFFF

```
      LEA R0, a
      LDR R0, R0, #0
.a    .FILL 0xBFFF
```

Le processeur LC3

BR

permet de réaliser des branchements inconditionnels ou conditionnels. Met à jour la valeur de PC pour modifier le flot d'exécution du programme. Trois drapeaux N, Z et P du registre PSR sont mis à jour dès qu'une valeur est chargée dans un registre

- N passe à 1 si cette valeur est strictement négative
- Z passe à 1 si cette valeur est nulle
- P passe à 1 si cette valeur est strictement positive

BR détermine en fonction de ces drapeaux si le saut doit être réalisé.

BR

- `BR label` réalise un saut inconditionnel vers l'étiquette `label`
- en ajoutant au moins un des indicateurs `N, Z` et `P` au nom de l'instruction, le saut devient conditionnel
- dans le cas d'un branchement condiionnel
  - le saut est réalisé si au moins un des drapeaux correspondant aux indicateurs est à 1
  - dans le cas contraire, on passe à l'instruction suivante (`PC <- PC+1`)

Le processeur LC3

Les routines

Une routine est une portion de code situé à une adresse marquée par une étiquette

- on peut effectuer un saut vers cette étiquette (`JSR`), on réalise un appel à la routine
- à la fin de l'exécution de la routine, une instruction de retour (`RET`) remet `PC` à l'adresse qui suit l'instruction d'appel

Les routines sont similaires aux fonctions des langages de plus haut niveau, mais le programmeur doit gérer beaucoup plus de choses

- `JSR` stocke l'adresse de retour dans R7, puis effectue un saut à l'adresse passée en opérande (label)
- `RET` permet de retourner à la routine appelante en faisant un saut à l'adresse contenue dans R7.

Les routines

Pour réaliser un appel de fonction, on ne dispose que des instructions `JSR` et `RET`. Plusieurs questions se posent :

- comment passer des paramètres à la fonction appelée
- comment la fonction appelante récupère-t-elle la valeur de retour
- comment savoir où reprendre l'exécution après l'appel lors d'appels imbriqués
- comment sauvegarder l'état des registres
- comment utiliser des variables locales

La réponse à toutes ces questions est : en utilisant une pile d'exécution.

Les routines

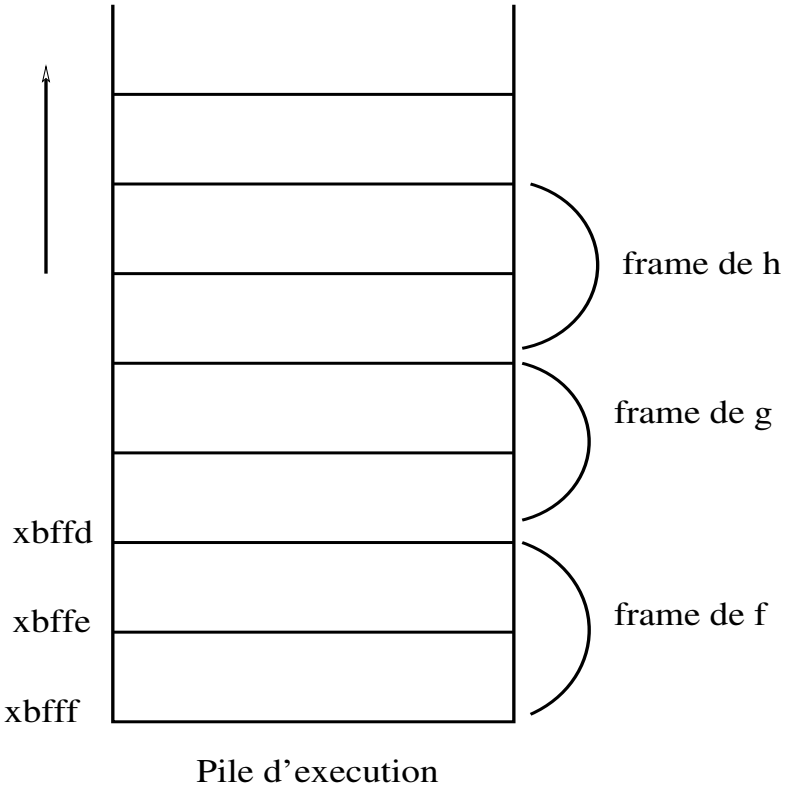
Pile d'exécution

- par convention, la pile d'exécution croît des adresses hautes vers les adresses basses
- elle est découpée en *frames*, chaque frame contient les informations liées à un appel de fonction.

Pendant l'exécution du code d'une méthode, le pointeur `FP` permet d'accéder aux paramètres et aux variables.
Par convention, sur LC3, il pointe vers la première variable locale.

Pile d'exécution

Ex. f appelle g qui appelle h,



Pile d'exécution

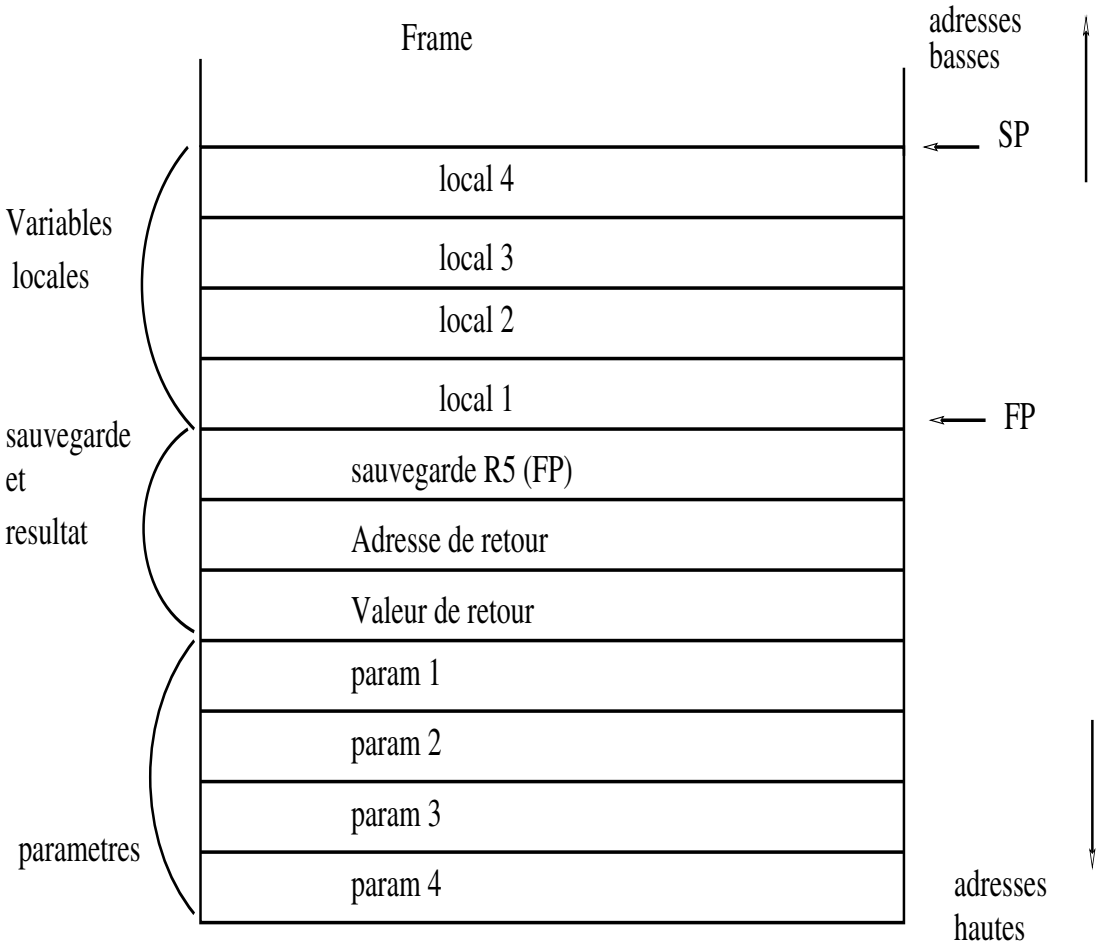
Chaque frame est délimitée par deux registres :

- stack pointer : adresse du sommet de la pile - R6 dans LC3
- frame pointer : adresse permettant d'accéder aux éléments de la frame courante - R5 dans LC3.

Les informations contenues dans une frame sont

- les paramètres de la fonction
- la valeur de retour de la fonction
- une sauvegarde de R5 pour rétablir la frame de l'appelant au retour de la fonction courante
- une sauvegarde de R7 (adresse de retour) pour ne pas la perdre lors de l'appel à d'autres fonctions.

Pile d'exécution



Passage de paramètres

Les paramètres d'une fonction peuvent être passés sur la pile.
Cependant, on utilise plus simplement les registres R0 à R4 pour les 5 premiers paramètres. Les suivants seront effectivement placés sur la pile.

Tout comme l'adresse de retour contenue dans R7, les valeurs des registres R0 à R4 peuvent être modifiées par la fonction appelée.
Si la fonction appelante a besoin du contenu de ces registres après l'appel, il faut également les sauvegarder. Pour cela, on utilisera des variables locales dans la pile.

Les routines

La gestion de la pile d'exécution repose sur une collaboration entre l'appelant et l'appelé.

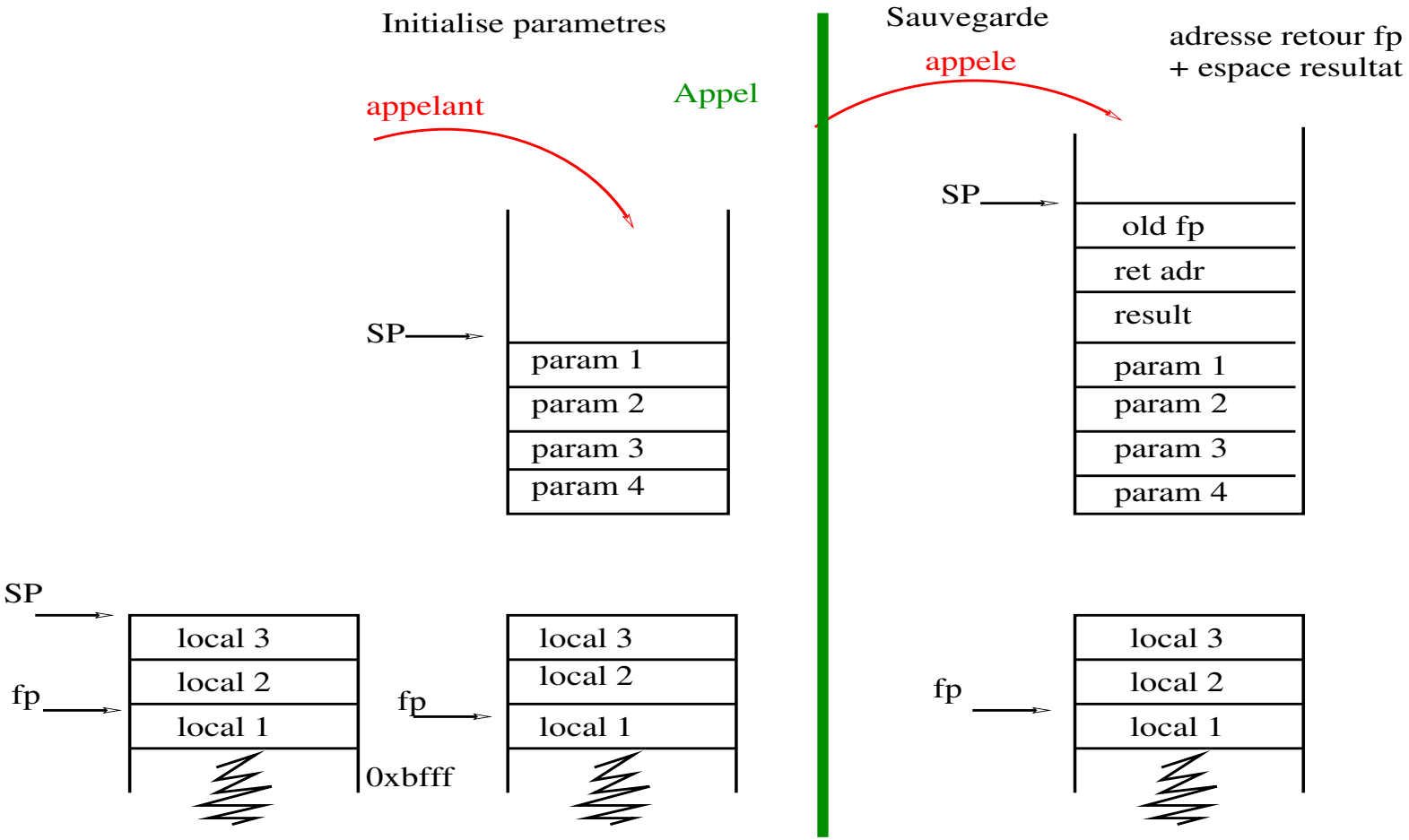
- Les tâches à réaliser par chacun dépendent de l'architecture et du système d'exploitation
- la répartition est définie par *les conventions d'appel*
- Ici, on utilisera la convention d'appel suivante pour la gestion de la frame de la routine appelée
  - l'appelant place les paramètres de l'appelé sur la pile avant l'appel et les retire après l'appel
  - l'appelé fait le reste

  L'appelant sauvegarde et restaure également les registres dont il a encore besoin après l'appel, mais il le fait dans sa propre frame (variables locales).

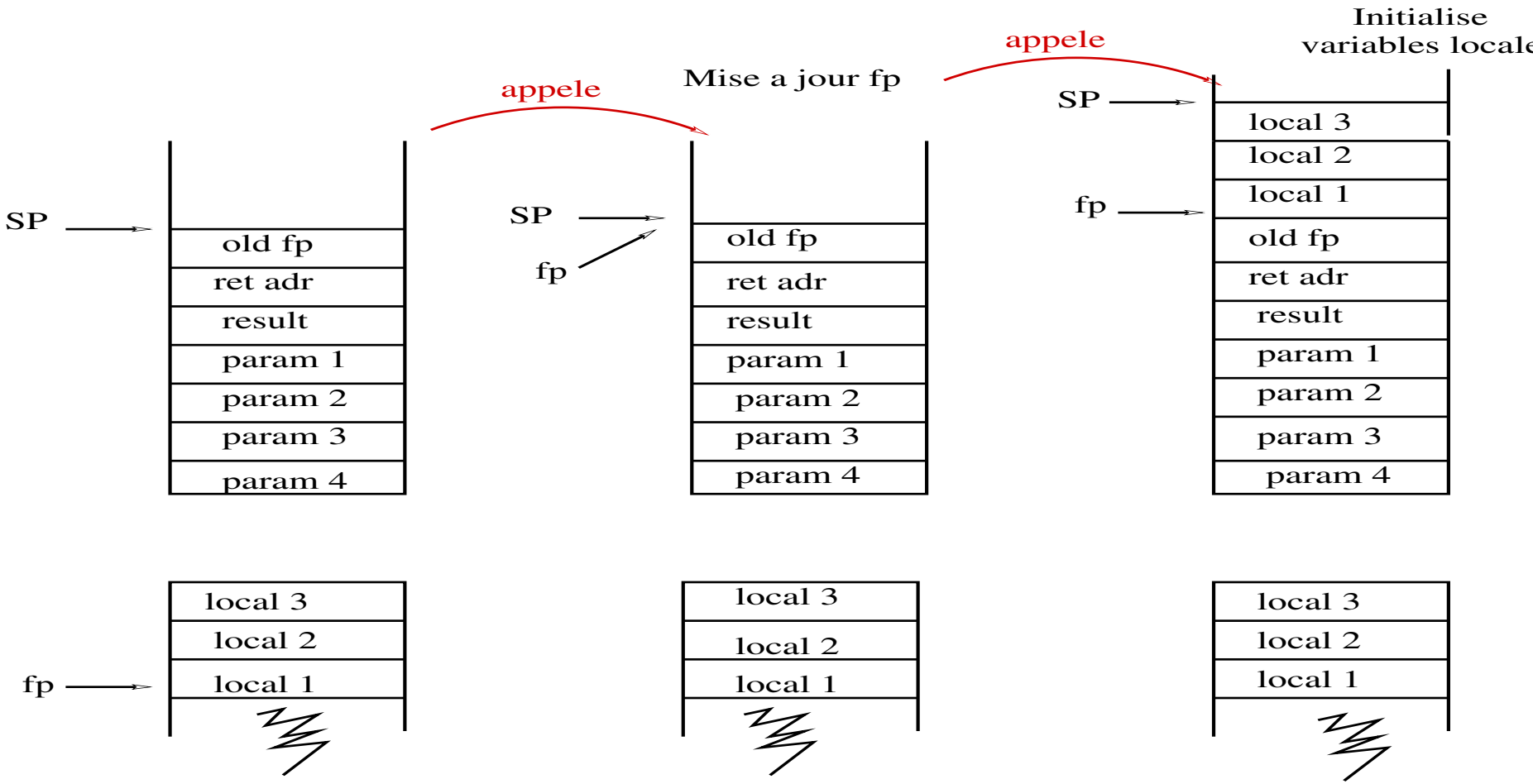
Les routines

- L'appelant empile les paramètres puis exécute `JSR`
- l'appelé réserve un espace sur la pile pour le résultat
- l'appelé sauvegarde sur la pile son adresse de retour (R7) et le fp (R5) courant
- l'appelé configure le nouveau fp
- l'appelé alloue de l'espace sur la pile pour ses variables locales
- l'appelé exécute son code
- l'appelé libère l'espace de ses variables locales
- l'appelé restaure l'ancien fp et son adresse de retour
- l'appelé exécute `RET`
- l'appelant dépile la valeur de retour et libère l'espace des paramètres
- l'appelant continue son exécution.

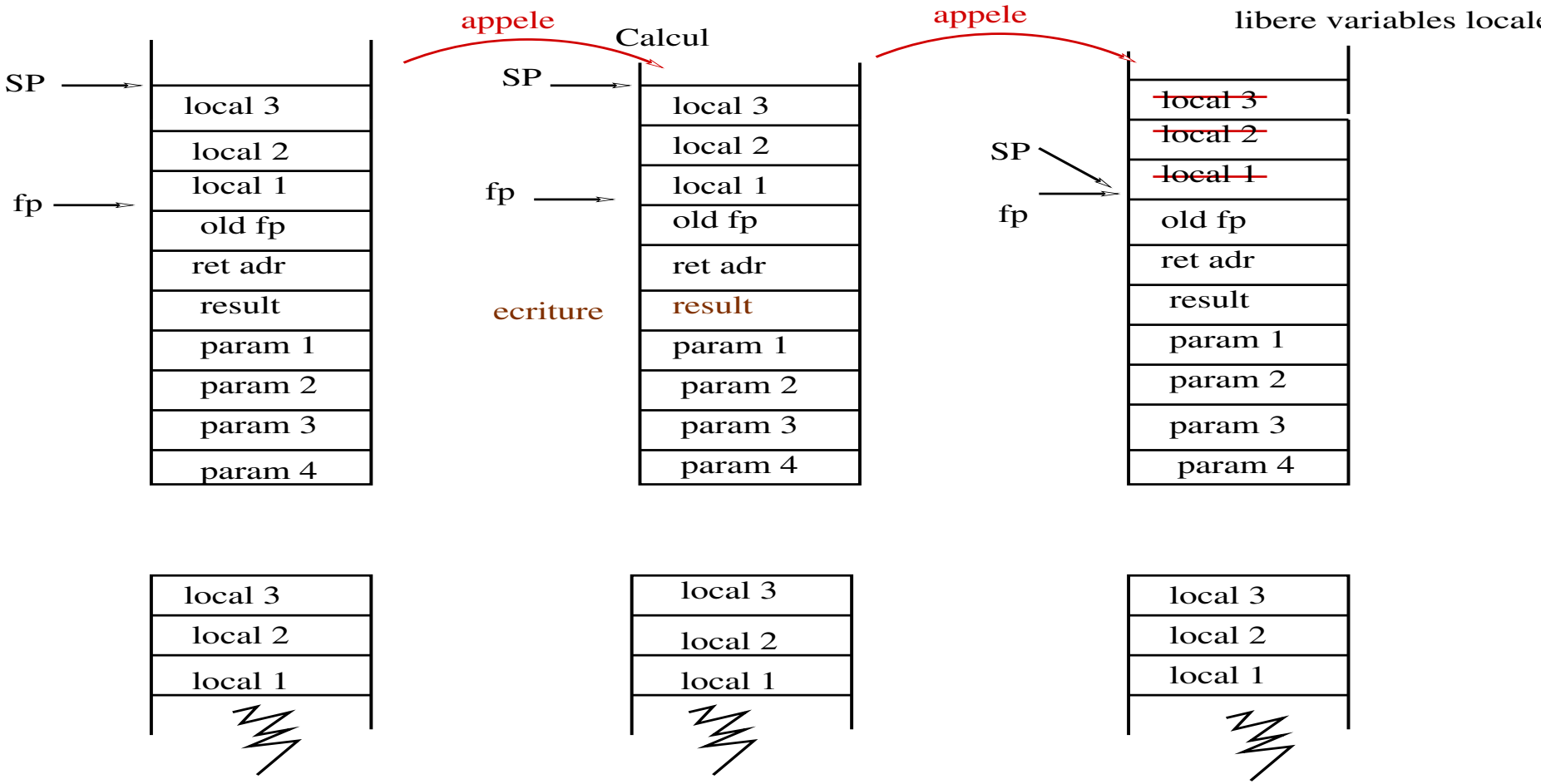
Les routines



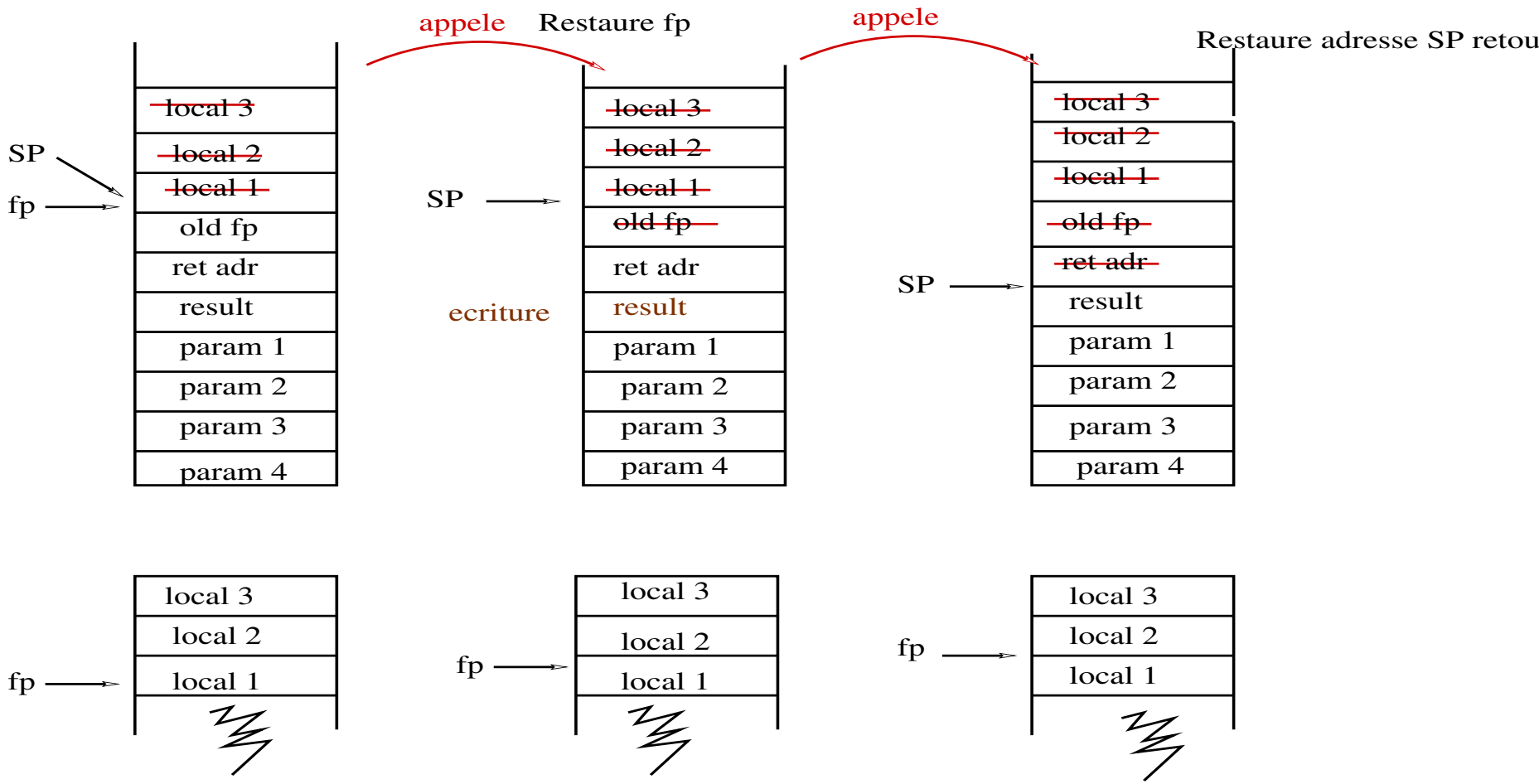
Les routines



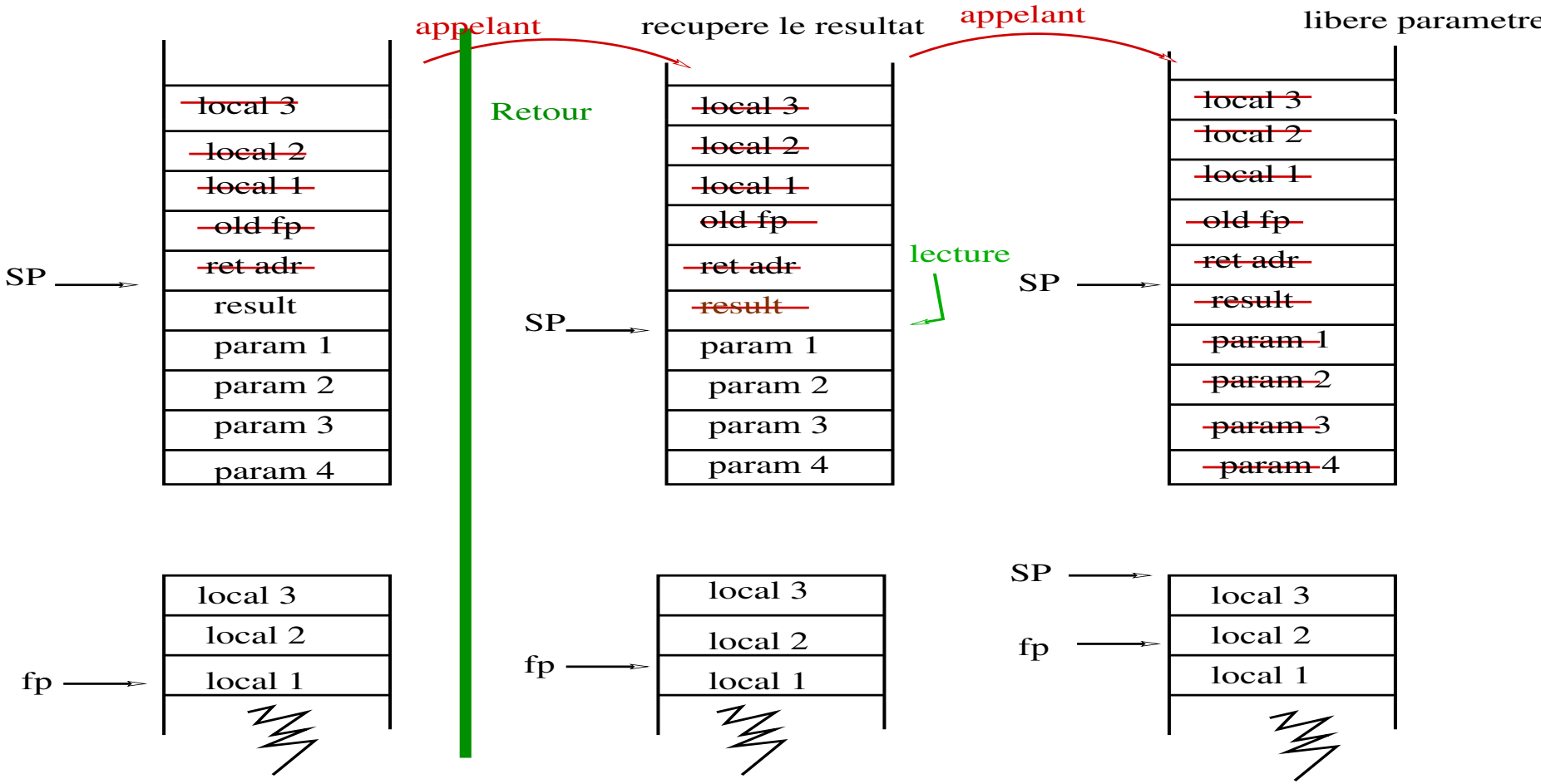
Les routines



Les routines



Les routines



Le processeur LC3

Exemple

- Passage de paramètres par les registres `sub.asm`
- Passage de paramètres par la pile `substack.asm`

Constructions usuelles

Déclaration de variables globales

code haut niveau	code LC3
int a ;	a .BLKW 1
int b = 1000 ;	b .FILL #1000
int c[10] ;	c .BLKW 10

Constructions usuelles

Affectation

code haut niveau	code LC3
b = a ;	LD R0, a ; ST R0, b ;

b = a + 1 ;	LD R0, a
	ADD R0, R0, #1
	ST R0, b

Constructions usuelles

Conditionnelle

code haut niveau	expression testée	branchement	négation
if (a < b)	(a-b) < 0	BRn	BRpz
if (a <= b)	(a-b) <= 0	BRnz	BRp
if (a == b)	(a-b) == 0	BRz	BRnp
if (a >= b)	(a-b) >= 0	BRzp	BRn
if (a > b)	(a-b) > 0	BRp	BRnz
if (a != b)	(a-b) != 0	BRnp	BRz
if (a)	a != 0	BRnp	BRz
if (!a)	a == 0	BRz	BRnp

Constructions usuelles

If simple

code haut niveau	code LC3

if (a < b) {
  // do something
}
	LD R0, a
	LD R1, b
	NOT R1, R1
	ADD R1, R1, #1
	ADD R0, R0, R1
	BRzp SKIP
	// do something
	SKIP
	// rest of the program

Constructions usuelles

Conditionnelles complexes

code haut niveau	code LC3
if (a || (b &&!c)) {	LD R0, a ;
// do something	BRnp DO
}	LD R0, b
	BRz SKIP
	LD R0, c
	BRnp SKIP
	DO
	// do something
	SKIP
	// rest of the program

Assembleur

Constructions usuelles

Boucle for : for (int i = 0 ; i < limit ; i++) { body }

- initialisation (int i = 0)
- condition de terminaison (i < limit)
- increment (i++)

Assembleur

Étape	code LC3
i = 0	AND R0, R0, #0 ST R0, i

i < limit	TEST LD R0, i
	LD R1, limit
	NOT R1, R1
	ADD R1, R1, #1
	ADD R0, R0, R1
	BRzp END

// body	// body

i++	LD R0, i
	ADD R0, R0, #1
	ST R0, i

	BR TEST END

Constructions usuelles

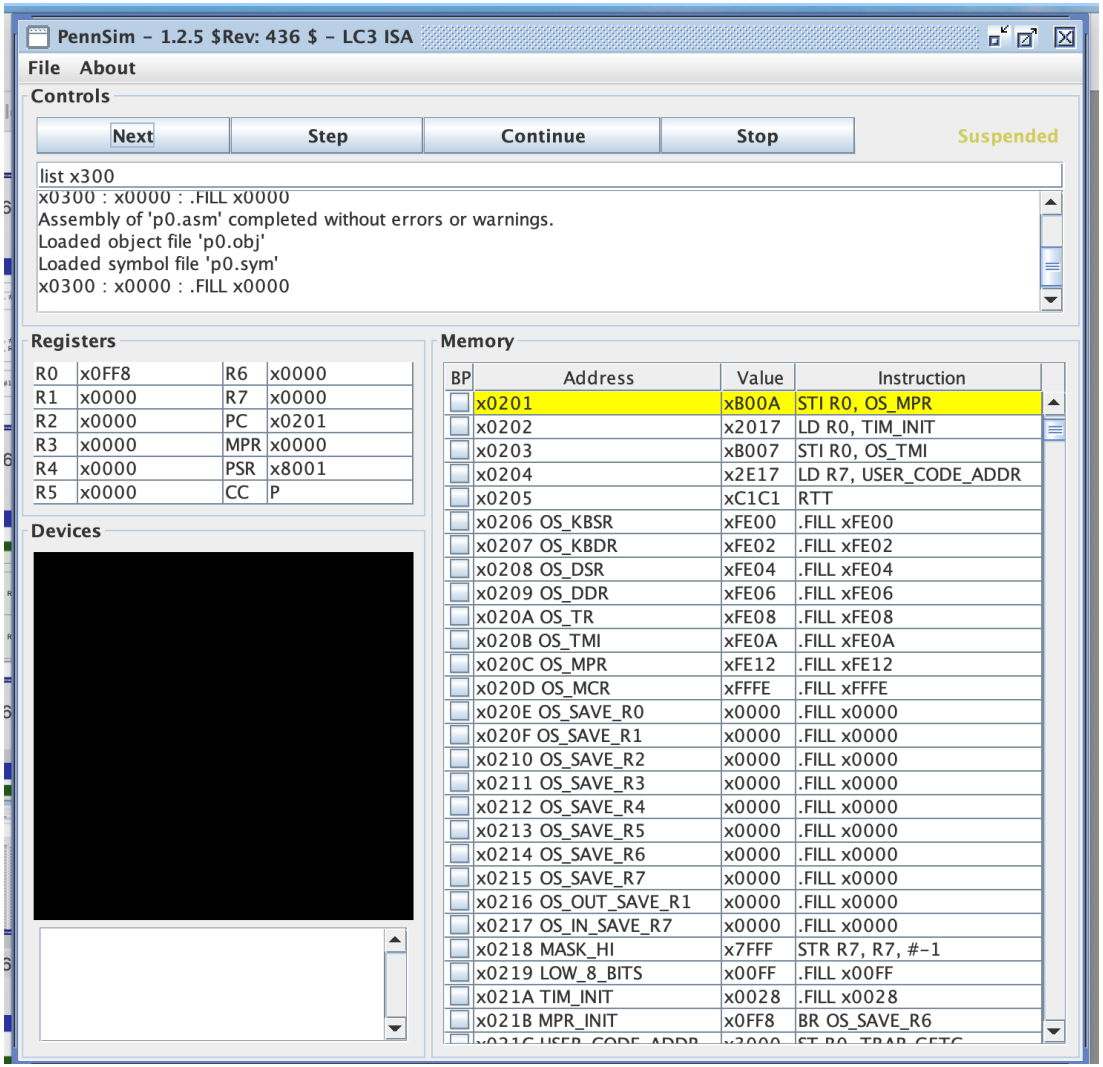
Tableaux

code haut niveau	code LC3

b = c[i]	LEA R0, c
	LD R1, i
	ADD R0, R0, R1
	LDR R0, R0
	ST R0, b

c[i] = b	LEA R0, c
	LD R1, i
	ADD R0, R0, R1
	LD R1, b
	STR R1, R0

Le simulateur PennSIm (LC3)



java -jar PennSim.jar : doit être exécuté dans les répertoires où sont les programmes.

Assembleur

PennSIm

Le simulateur exécute des fichiers objets (fichiers binaires en langage machine) dont l'extension est `.obj`

- les programmes sont écrits en assembleur, l'extension de ces fichiers est `.asm`
- les fichiers `.asm` doivent être traduits en fichiers `.obj` pour être exécutés
  réalisé par la commande `as filename.asm`
- le fichier objet est ensuite chargé par la commande `load filename.obj`

Assembleur

PennSim

Avant de charger le programme, il faut charger un petit système d'exploitation qui gèrera les E/S

```
as lc3os.asm
load lc3os.obj
as myProg.asm
load myProg.obj
```

Par convention, les programmes commencent à l'adresse `0x3000`

Pour exécuter le programme, on clique sur `Continue`. Le bouton `Stop` permet de l'arrêter. L'exécution pas à pas se fait avec `Next` et `Step`.

Le simulateur PennSim (LC3)

```
;; Set R0 to 10*R1
          .orig x3000
mul10     ADD R0,R1,R1              ; R0 <- 2*R1
          ADD R0,R0,R0              ; R0 <- 4*R1
          ADD R0,R0,R1              ; R0 <- 5*R1
          ADD R0,R0,R0              ; R0 <- 10*R1
          HALT                      ;
          .end
```

Le simulateur PennSim (LC3)

PennSylvaniaAssembler — nano example2CM.asm — 84×27

```
  GNU nano 2.0.6                File: example2CM.asm

;; Set R3 to R1 xor R2
;; i.e. ( (!R1 && R2) || (R1 && !R2) )
;; Rappel: x || y = ! (!x && !y)

        .orig x3000
START   LD R4,R1                ; R4 <- R1
        NOT R4, R4              ; R4 <- !R1
        AND R4, R4, R2          ; R4 <- !R1 && R2
        LD R0, R2               ; R0 <- R2
        NOT R0, R0              ; R0 <- !R2
        AND R0, R1, R0          ; R0 <- R1 && !R2
        NOT R4, R4              ; R4 <- !(!R1 && R2)
        NOT R0, R0              ; R0 <- !(R1 && !R2)
        AND R3, R4, R0          ;
        NOT R3, R3
        HALT                    ;
        .end
```

Application :

Exemple 1

Écriture à l'écran : l'instruction PUTS

```
        .ORIG x3000
        LEA R0, chaine
        PUTS                          ; appel systè
        HALT
chaine: .STRINGZ "hello\n"
.END
```

Application :

Exemple 2

Afficher les entiers de 9 à 0

```
          .ORIG x3000
; partie dédiée au code
          LEA R0, msg0              ; R0 <- @ effective désignée par msg0
          PUTS                      ; affiche la chaîne de car (TRAP x22)
          LD R2, carzero            ; R2 <- '0'
          AND R1, R1, 0             ;
          ADD R1, R1, 9             ; R1 <- 9
loop:     BRn endloop               ; si R1 < 0 aller à endloop
          ADD R0, R1, R2            ; R0 <- R1+'0' = code ascii du chiffre R1
          OUT                       ; affiche le car contenu dans R0 (TRAP x21)
          ADD R1, R1, -1            ; R1 <- R1-1
          BR loop
endloop:  LEA R0, msg1              ; charge la chaîne de car désignée par msg1 dans R0
          PUTS                      ; l'affiche
          HALT
; partie dédiée aux données
carzero:  .FILL x30                 ; code ASCII du caractère '9' (48 en décimal)
msg0:     .STRINGZ "Affichage des entiers de 9 à 0 : \n"
msg1:     .STRINGZ "\nFin de l'affichage !\n"
          .END
```