Architecture des ordinateurs - CM3

Pierre Nerzic

automne 2023

Contenu :

- codage des instructions
- assemblage / désassemblage
- dispositifs avancés : modes d'adressage, pile et pointeur de pile,
- traduction des structures de contrôle avancées : tests, boucles, tableaux, fonctions, etc.

Programmation d'une unité centrale

Pierre Nerzic

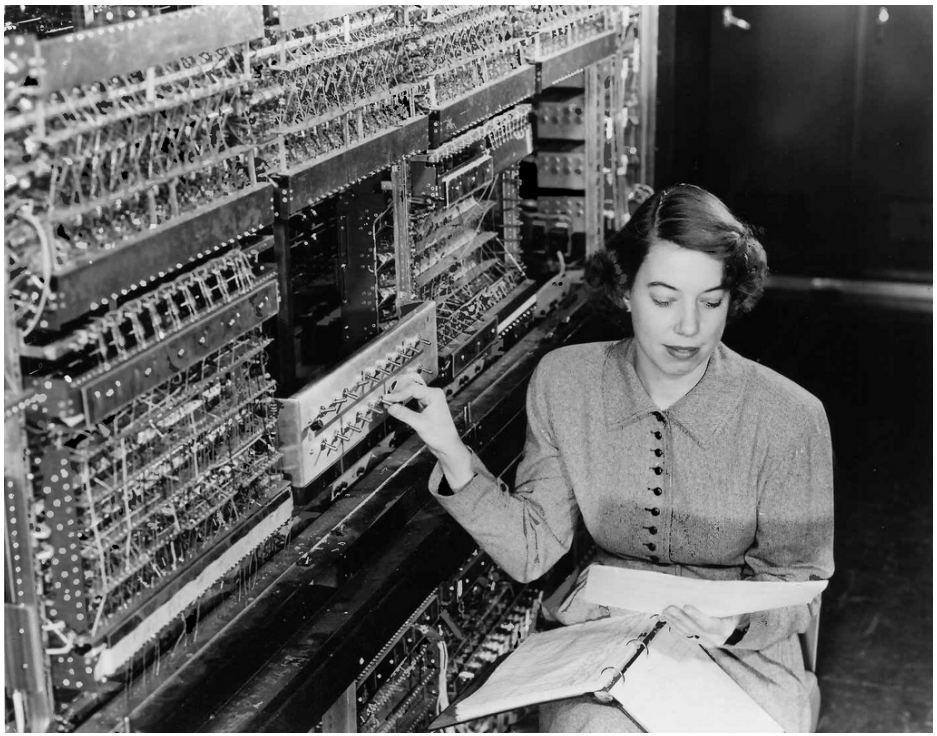
Concepts

On veut faire exécuter un programme sur un ordinateur. Pour cela, on doit :

1. Écrire le programme sous forme d'instructions : type d'instruction et paramètres (appelés *opérandes*) ; c'est ce qu'on appelle le « **langage machine** »
2. Coder ces instructions et leurs opérandes, ainsi que les données que devra traiter le programme, sous forme d'octets,
3. Saisir tous les codes dans la mémoire centrale,
4. Lancer l'exécution,
5. Récupérer les résultats dans la mémoire centrale ou sur les périphériques.

Historique de la programmation

Aux débuts de l'informatique, les programmes devaient être saisis à l'aide d'interrupteurs et de boutons poussoirs. Il n'y avait pas de notion de fichiers source/exécutable.



Pierre Nerzic

Historique, suite

Les progrès matériels ont permis d'enregistrer les programmes dans de la mémoire permanente (bandes magnétiques, puis disque durs, disquettes, puis disques SSD, etc.).

Ensuite des outils permettant de travailler plus confortablement ont été inventés :

- des traducteurs d'un langage humainement lisible (Fortran, C, Java, etc.) vers le langage machine (instruction et données codées) qu'on appelle « **compilateurs** ».
- des éditeurs de plus en plus interactifs pour ces langages.

Ce cours se situe au milieu : programmation en langage machine, mais avec un environnement moderne.

Langage machine

Le langage machine se présente ainsi :

```
02 1B 0A 23 81 72 0A 5F
```

Ce sont des octets qui spécifient les instructions pour l'unité centrale. Parfois, ces octets doivent être à des emplacements (adresses) bien précis pour pouvoir fonctionner, par exemple quand les paramètres sont des données en mémoire.

On peut parfaitement travailler de cette manière, en fournissant une suite d'octets, mais c'est très difficile à comprendre et à vérifier. De tels programmes ne peuvent pas être longs et complexes.

On a donc rapidement défini un langage plus accessible aux humains appelé « **langage d'assemblage** ».

Langage d'assemblage

Voici le même programme en langage d'assemblage :

```
LD R0, 27
LD R1, 35
ADD R0, R1
OUT R0, 10
HLT
```

Il y a une instruction par ligne. Une instruction est composée de :

- un « **mnémonique** » : le nom de l'instruction généralement abrégé sur 2 à 4 lettres. Ici, on a :
  - `LD` signifie *load*, càd affecter,
  - `ADD` signifie additionner,
  - `OUT` signifie d'envoyer une information en sortie,
  - `HLT` signifie *halt*, arrêter l'exécution.

Langage d'assemblage, suite

- des paramètres appelés « **opérandes** ». Ils dépendent de chaque instruction. On peut trouver :
  - des registres
  - des nombres
  - des adresses mémoire

On trouve aussi des « **directives** » qui dirigent le codage du programme. Par exemple :

```
ORG 20
DB 1, 2, 3
```

- `ORG` indique l'origine, c'est à dire l'adresse où mettre ce qui suit.
- `DB` (*data byte*) donne une liste de valeurs à placer en mémoire

Nommage de valeurs et adresses : labels

Pour faciliter l'écriture des programmes, on ajoute ce qui s'appelle des « **labels** » ou étiquettes. Ce sont des noms pour représenter des adresses mémoire ou des constantes. On les place dans la marge et on les fait suivre d'un « : ». Exemples :

```
debut:                      ; début du programme
        LD  R0, init        ; initialiser R0 à la valeur d'init
        OUT R0, 10          ; écrire R0 sur le terminal
fin:    HLT                 ; arrêt du processeur

init:   EQU 5               ; init vaut la constante 5
```

La directive `EQU` signifie *equals*, c'est à dire « égale ». Elle permet de définir un label avec une valeur constante.

Codage des instructions

Chaque combinaison mnémonique + types des paramètres est codée spécifiquement. Il y a toujours un premier octet appelé « **opcode** » qui définit l'instruction pour le décodeur d'instruction. Ensuite, selon l'instruction, il y a les opérandes.

Par exemple avec CimPU, l'instruction `LD` a différents codes :

- `LD reg1,reg2` avec reg1 et reg2 parmi `R0` et `R1`, est codée sur 1 octet, $(0000r00s)_2$, r étant le numéro 0 ou 1 du premier registre et s celui du second.
- `LD reg,nombre` est occupe 2 octets, $(0000r010)_2$ suivi du nombre.

Ces codes ne sont pas à apprendre, grâce au langage d'assemblage.

Assemblage et désassemblage

L'« **assembleur** » est un logiciel qui traduit un programme source en codes machine. Il y a un autre logiciel appelé « **désassembleur** » qui fait le travail inverse. Il décode les octets trouvés en mémoire et écrit les instructions correspondantes. Par contre, il ne peut pas deviner dans quelle base on préfère les nombres.

Instructions		
adr	octets	instruction
00	02 1B	LD R0, $1B
02	0A 23	LD R1, $23
04	81	ADD R0, R1
05	72 0A	OUT R0, $0A
07	5F	HLT

Pierre Nerzic

Catalogue des instructions

Voici une petite liste des instructions de CimPU. On les retrouve dans tous les processeurs.

- affectations
- calculs
- « entrées/sorties »
- contrôle d'exécution
- divers

Pierre Nerzic

Instructions d'affectation

Pierre Nerzic

Présentation

Leur rôle est de transférer une valeur d'un endroit à l'autre :

- d'un registre à un autre,
- de la mémoire à un registre.

Il y a deux instructions à connaître :

- `LD reg,source` : (*load*) la « **source** » est la valeur à placer dans le registre. Ça peut être :
  - une constante,
  - un autre registre,
  - un emplacement mémoire.
- `ST reg,destination` : (*store*) la « **destination** » indique l'endroit où placer la valeur qui est dans le registre. Dans CimPU, c'est forcément un emplacement mémoire.

Affectation de constantes

Les constantes, dans CimPU, peuvent être fournies de différentes manières :

- nombre en base 10 : rien de spécial à mettre
- nombre en base 16 : mettre un $ devant la valeur
- nombre en base 2 : mettre un % devant la valeur
- code ascii d'un caractère : l'entourer par `'c'` ou `"c"`
- expression arithmétique : `+ - * / % ()`

```
LD R1, 23           ; R1 = 23 (en base 10)
LD R0, $A0          ; R0 = 160
LD R1, %10100110    ; R1 = $A6 = 166
LD R0, 'Z'          ; R0 = code ascii de Z (90)
LD R1, (2+3)*4+1    ; R1 = 21
```


Pierre Nerzic

Emplacement mémoire

Pour désigner un emplacement mémoire, il suffit de mettre son adresse entre crochets, comme un tableau en C :

```
    LD R1, [99]      ; R1 = le contenu de la mémoire en 99
    ST R0, [100]     ; écrire R0 à l'adresse 100
```

On peut aussi utiliser un registre pour indiquer l'adresse mémoire. On appelle ça l'« **adressage indirect registre** ». On peut aussi ajouter une constante pour calculer l'adresse effective.

```
    LD R0, 106       ; R0 = 106
    LD R1, [R0]      ; R1 = l'octet situé à l'adresse R0
    ST R1, [R0+1]    ; enregistrer R1 à l'adresse R0 + 1
```

Instructions de calcul

Présentation

Tous les processeurs ont des instructions arithmétiques et logiques.

- `ADD reg, source` : additionne la source au registre. La source peut être :
  - une constante,
  - un registre,
  - un emplacement mémoire.
- `SUB reg, source` : soustrait la source du registre.

Ces instructions modifient le registre spécial appelé `CC` (*condition code*) ou registre d'état, décrit dans le CM n°2. Il est composé de 4 bits, appelés `V`, `C`, `N` et `Z`. Par exemple, `C` passe à 1 quand il y a une retenue, `N` quand le résultat est négatif, `Z` quand le résultat est nul.

Ces indicateurs permettent de réaliser des conditionnelles, page 30.

Autres instructions de calcul

L'indicateur `C` est utilisé dans les instructions suivantes pour prendre en compte une retenue précédente :

- `ADC reg, source` : additionne la source + la retenue au registre,
- `SBC reg, source` : soustrait la source + la retenue du registre.

On trouve aussi toutes les instructions logiques :

- `AND reg, source` : fait un « et » entre le registre et la source
- `OR`, `XOR`, `NOT`...

D'autres instructions seront vues en TD et TP.

Entrées/sorties

Pierre Nerzic

Ports d'entrées/sorties

C'est par ces termes qu'on désigne les mécanismes qui font communiquer l'unité centrale avec les périphériques. Dans CimPU, c'est extrêmement simplifié. Les périphériques sont sur des « **ports** ». Ce sont comme des adresses mais qui, lues ou écrites, déclenchent des échanges avec l'extérieur.

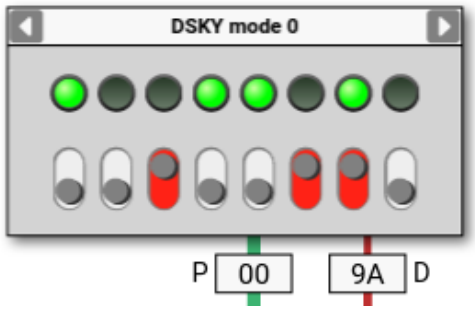
Par exemple, écrire un octet sur le port 10 provoque l'affichage de l'octet en base 10, 16 et 2, ainsi que le caractère ASCII associé.

- `IN reg,port` : affecte le registre avec ce qu'il y a sur le port,
- `OUT reg,source` : émet le registre sur le port.

Dans certaines unités centrales, les périphériques sont vus comme de la mémoire. Il suffit de lire ou d'écrire à certaines adresses pour échanger des données avec les périphériques. C'est le cas par exemple d'une carte graphique dans un PC moderne.

Exemple de port d'entrées/sorties

Sur CimPU, le port 0 est relié à des leds et des inverseurs.



Lors d'un accès, le numéro de port `P` est sur le bus vert, et la donnée sur le bus de données en rouge.

- L'instruction `OUT R0,0` fait allumer et éteindre les leds selon l'écriture binaire de `R0`. Ici, $(10011010)_2 = (9A)_{16}$.
- L'instruction `IN R0,0` affecte `R0` avec l'état des interrupteurs, ici: $(00100110)_2$.

Remarques

Les ordinateurs sont tous différents, même s'ils ont un processeur identique. C'est le rôle des « **pilotes** » (*drivers*) d'uniformiser l'accès aux périphériques, sous la forme de fonctionnalités uniformes d'un ordinateur à l'autre.

En langage C, on appelle une fonction, par exemple `putchar`, qui est identique quel que soit l'ordinateur, et le résultat, c'est un caractère affiché à l'écran, quel que soit l'écran.

Les pilotes sont une partie variable du système d'exploitation. Chaque fabricant de périphérique fournit les pilotes permettant de faire fonctionner et d'utiliser les périphériques installés sur l'ordinateur.

CimPU n'offre pas de pilotes et vous montre une réalité matérielle simplifiée, mais réaliste, comme sur un micro-contrôleur.

Contrôle d'exécution

Présentation

Avec les instructions précédentes, on peut écrire des programmes uniquement linéaires : une suite de calculs sans aucune conditionnelle ni répétition. Pour programmer des alternatives et des boucles, il faut des instructions capables de faire avancer l'exécution ou revenir en arrière, selon une condition.

Il y a une astuce. Ce sont des instructions capables de modifier le registre `IP`. Ce registre contient l'adresse de la prochaine instruction à exécuter. Si on l'affecte en cours d'exécution, alors la prochaine instruction ne sera pas la suivante, mais celle dont on a mis l'adresse dans `IP`. L'idée est évidemment d'aller sur une instruction voulue.

On appelle ça un « **branchement** » ou un « **saut** ».

Branchements

Soit ce programme abstrait. Les adresses sont mises dans la marge.

```
A1: calcul n°1
A2: si un test est vrai, alors mettre A4 dans IP
A3: calcul n°2
A4: suite du programme...
```

L'instruction en A2 modifie `IP` seulement si un test est vrai.

- Lorsque le test est faux, alors il ne se passe rien de spécial : l'exécution continue à l'adresse A3 avec le calcul n°2.
- Lorsque le test est vrai, alors l'instruction de l'adresse A2 modifie `IP`, y met A4, donc la prochaine instruction exécutée est directement à l'adresse A4, sans passer par l'adresse A3. Le calcul n°2 n'est pas fait.

Branchements, suite

Le programme précédent consiste en une structure conditionnelle de type `if (condition) { calcul n°2 }`. Le calcul n°2 est sauté à l'aide d'un branchement (ou saut) si la condition est fausse.

Il y a deux types de branchements :

- inconditionnels, ils sont toujours effectués.
- conditionnels : ils sont effectués si une condition est vraie. Cette condition est une combinaison des indicateurs d'état VCNZ. Par exemple, on peut effectuer le branchement si Z ou N sont à 1 (résultat du calcul négatif ou nul).

Et il y a deux manières de calculer l'adresse d'un branchement :

- absolu : l'opérande est directement la future valeur de `IP`,
- relatif: l'opérande est un nombre relatif qui est additionné à `IP`.

Pierre Nerzic

Instructions utiles pour les structures de contrôle

Pour coder une alternative ou une boucle, on a besoin de :

- instruction de comparaison : `CMP registre,valeur`. Elle compare le registre à la valeur et affecte les indicateurs d'état :
  - `Z` devient 1 si registre $==$ valeur
  - `C` devient 1 si registre $<$ valeur en les considérant en tant qu'entiers non-signés
  - `N` devient 1 si registre $<$ valeur en les considérant en tant qu'entiers signés
- nombreuses instructions de branchement conditionnels, ex :
  - `BEQ` saute si `Z`$==$1, `BNE` saute si `Z`$==$0
  - `BCS` saute si `C`$==$1, `BCC` saute si `C`$==$0
  - `BMI` saute si `N`$==$1, `BPL` saute si `N`$==$0

Il y en a beaucoup. Les plus utiles seront étudiées en TD et TP.

Programmation des alternatives

Langage C

```
if (condition)


{
    instructions1;
}
```

Assembleur

```
si:     ; calcul de la condition
        ; => indicateurs VCNZ
        Bxx finsi   ; saut si faux
alors:
        ; instructions1
finsi:
```

Le branchement Bxx doit sauter quand la condition est **fausse** :

```
if (i == 3)


{
    instructions1;
}
```

```
si:     ; condition: [i] == 3
        LD  R0, [i]
        CMP R0, 3   ; => Z=1 si R0==3
        BNE finsi   ; saut si Z==0
alors:
        ; instructions1
finsi:
```


Pierre Nerzic

Alternative complète (*if* + *else*)

Langage C

```
if (condition)

{
    instructions1;
}
else {
    instructions2;
}
```

Assembleur

```
si:     ; calcul de la condition
        Bxx sinon   ; saut si faux
alors:  ; instructions1
        BRA finsi
sinon:  ; instructions2
finsi:
```

- Quand la condition est vraie, on ne fait pas le saut `Bxx`, donc l'exécution continue dans le *alors*. À la fin du *alors*, il y a un saut inconditionnel, `BRA` (*branch always*), sur le *finsi*.
- Quand la condition est fausse, il y a un saut `Bxx` sur le *sinon*, donc la partie *alors* n'est pas faite.

Boucle non bornée (*while*)

Langage C

```
while (condition)
{
    instructions1;
}
```

Exemple :

```
while (i < 3)



{
    i = i + 1;
}
```

Assembleur

```
while:  ; calcul de la condition
        Bxx endwhile    ; saut si faux
do:     ; instructions1
        BRA while
endwhile:
```

```
while:  ; condition: [i] < 3
        LD  R0, [i]
        CMP R0, 3    ; => N=1 si R0<3
        BPL endwhile    ; saut si N==0
do:     LD  R0, [i]
        INC R0
        ST  R0, [i]
        BRA while
endwhile:
```


Pierre Nerzic

Boucle bornée (*for*)

Langage C

```
for (init; cond; increm)

{
    instructions1;

}
```

Assembleur

```
for:     ; initialisation
forloop: ; calcul de la condition
         Bxx endfor   ; saut si faux
         ; instructions1
         ; incrémentation
         BRA forloop
i:       DB  0 ; variable de la boucle
endfor:
```

En fait, une boucle *for* du langage C est une sorte de boucle *while*.

Avec CimPU, on peut placer des variables au milieu des instructions, à condition que l'exécution n'arrive jamais sur ces variables.
Dans un processeur actuel, les variables et les instructions sont totalement séparées par la « **segmentation** ». Les instructions sont une zone en lecture seule, et les variables en lecture et modification.

Exemple de boucle *for*

```
for (int i=0; i<5; i++)




{
    instructions1;

}
```

```
for:     LD  R0, 0
         ST  R0, [i]
forloop: ; condition: [i] < 5
         LD  R0, [i]
         CMP R0, 5
         BPL endfor    ; saut si faux
         ; instructions1
         ; incrémentation
         LD  R0, [i]
         INC R0
         ST  R0, [i]
         BRA forloop
i:       DB  0    ; variable i
endfor:
```


Pierre Nerzic

Remarques

Il est parfois possible d'optimiser le programme en assembleur. Le précédent programme peut être simplifié ainsi :

```
for:     LD  R0, 0
forloop: ST  R0, [i]
         ; condition: [i] < 5
         CMP R0, 5
         BPL endfor   ; saut si faux
         ; instructions1
         ; incrémentation
         LD  R0, [i]
         INC R0
         BRA forloop
i:       DB  0   ; variable i
endfor:
```

On économise quelques instructions, celles où R0 ne change pas de valeur. Attention à bien vérifier tous les chemins d'exécution.

Programmation d'une condition

Les exemples précédents montrent le calcul d'une condition simple. Cela se fait avec une comparaison suivie d'un branchement basé sur les indicateurs.

Dans le cas d'une conjonction ou disjonction, il faut traiter chaque condition séparément et placer des branchements à chaque condition élémentaire :

- conjonction : saut au *finsi*, *sinon* ou *endwhile* à chaque condition élémentaire fausse,
- disjonction : saut au *alors* ou *do* à chaque condition élémentaire vraie, puis saut inconditionnel au *finsi*, *sinon* ou *endwhile* tout à la fin.

Le cas général (mélange de conjonctions, disjonctions et négations) est très complexe et ne sera pas étudié en TD.

Traduction des structures de données

Introduction

Voyons maintenant comment on travaille avec des structures de données en assembleur.

On a déjà vu comment définir et accéder à des variables simples. Elles sont copiées dans les registres pour les calculs. Il est possible d'avoir besoin de variables supplémentaires, temporaires, pour les calculs compliqués. Voici un exemple :

```
;; définition des variables
a:      DB  17   ; valeur initiale = 17
b:      RB  1    ; réservation de 1 octet non initialisé

;; utilisation des variables
        LD  R0,[a]      ; R0 = a
        ADD R0, R0      ; multiplie R0 par 2
        ST  R0,[b]      ; b = R0, càd a*2
```

Tableaux

Les tableaux sont des suites d'octets auxquels on accède par un indice. Cet indice est placé dans un registre et on utilise l'adressage indirect avec décalage pour accéder à la case, voir page 17 :

```
;; définition du tableau
t:      RB  10  ; réservation de 10 octets non initialisés

;; utilisation du tableau
        LD  R0,5        ; R0 = indice de la 6e case
        LD  R1,[R0+t]   ; R1 = contenu de la case
        INC R0          ; R0 = indice de la case suivante
        ST  R1,[R0+t]   ; R1 va dans la case suivante
```

Comme en langage C, la première case du tableau est à l'indice 0. Comme en C, il n'y a aucune vérification sur les indices ; ils peuvent même être négatifs (le calcul `R0+t` est en convention $C2^8$).

Structures

Les structures sont des associations de plusieurs variables formant des *champs*. Dans CimPU, il n'y a que le type octet, alors les structures sont assez limitées. On accède aux champs à l'aide de l'adressage indirect avec décalage. Dans ce cas, le registre contient l'adresse de la structure et le décalage est celui du champ.

```
;; définition de deux structures
s1:     DB  2, 3    ; deux champs de 1 octet
s2:     DB  7, 6    ; deux champs de 1 octet

;; utilisation des structures
        LD  R0,s1       ; R0 = adresse de s1
        LD  R1,[R0+0]   ; R1 = s1.premier champ
        ADD R1,[R0+1]   ; ajout de s1.second champ
        LD  R0,s2       ; R0 = adresse de s2
        ST  R1,[R0+0]   ; s2.premier champ = R1
```

Appels de fonctions

Pierre Nerzic

Présentation

On arrive à quelque chose d'assez compliqué. Comment définir et appeler des fonctions ?

Le principe est :

- de définir une partie du programme indépendante du reste et terminée par une instruction de type *return*.
- d'appeler cette partie du programme et son instruction *return* fait revenir à l'appel.

Cela fonctionne parce qu'il y a une partie de la mémoire qui sert à mémoriser où il faut revenir au retour de la fonction. Cette partie de la mémoire s'appelle la « **pile** ».

Pierre Nerzic

Exemple de fonction très simple

```
;; partie principale
        LD   R0, 7        ; paramètre de la fonction
        CALL fois2        ; appel de la fonction fois2
        OUT  R0, 10       ; affichage sur le terminal
        CALL fois2        ; 2e appel de la fonction
        OUT  R0, 10       ; affichage sur le terminal
        HLT               ; fin du programme

;; définition de la fonction
fois2:  ADD  R0, R0       ; R0 = R0 + R0
        RET               ; retour au CALL correspondant
```

- L'instruction `CALL adresse` enregistre la valeur de `IP` sur la pile, puis affecte `IP` avec l'adresse de la fonction.
- L'instruction `RET` restitue la valeur de `IP` au moment du `CALL`.

Pierre Nerzic

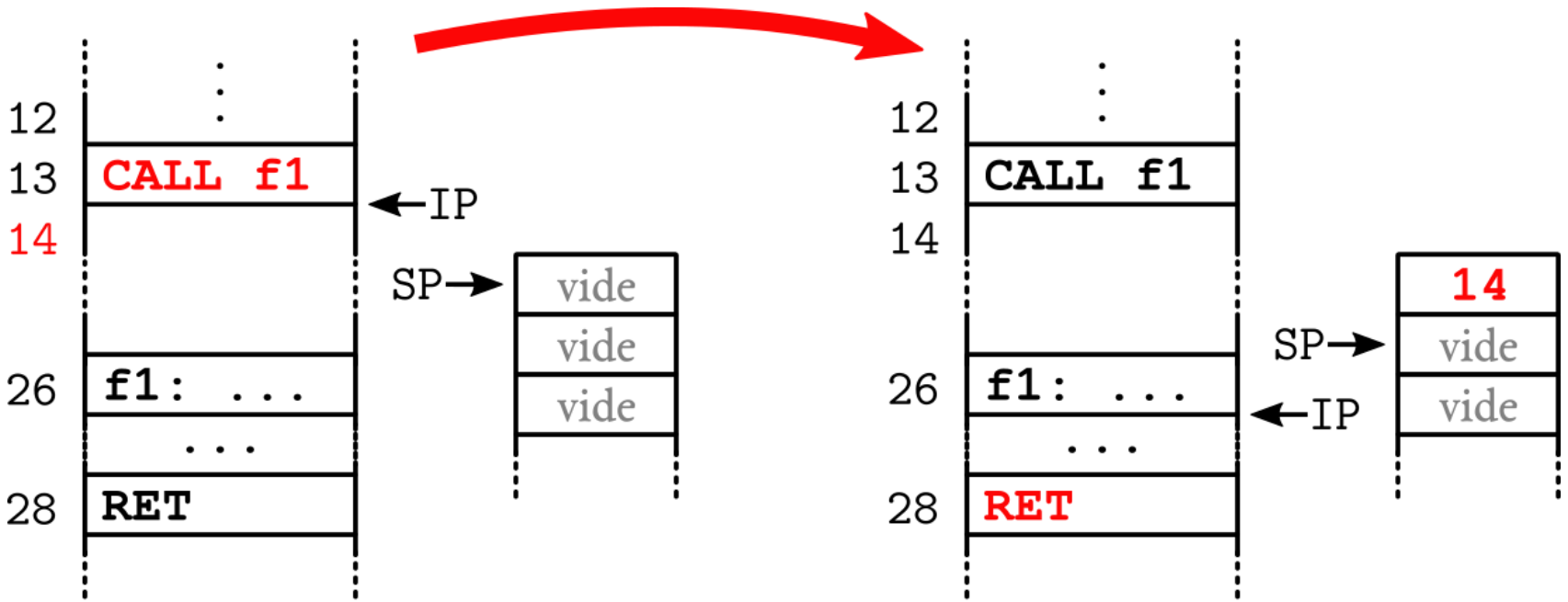
Pile

La pile est une sorte de tableau qui se remplit en descendant dans la mémoire. Il y a un registre spécial, `SP` qui indique où en est la pile. Il est initialisé à l'adresse $(\texttt{FF})_{16}$, à la fin de la mémoire.

- L'instruction `CALL` place la valeur de `IP` dans `[SP]`, puis décrémente `SP`, et enfin affecte `IP` avec l'adresse en opérande,
- L'instruction `RET` incrémente `SP`, puis affecte `IP` avec `[SP]`.

Ça fait comme une pile d'assiettes, chaque assiette contient une adresse, celle d'où on vient = celle où on a fait un `CALL`.
L'instruction `RET` enlève l'assiette du dessus et remet `IP` comme il était au moment du `CALL`, donc l'exécution revient là où la fonction a été appelée.

Appel d'une fonction



Lorsqu'un `CALL` est rencontré, la valeur de IP est empilée. On l'appelle *adresse de retour*. Elle est dépilée lors d'un `RET`.

NB: la valeur de `IP` qui est empilée est l'adresse de l'instruction qui suit le `CALL`, car `IP` est toujours en avance. Ainsi, au retour, c'est par cette instruction que l'exécution continue.

Paramètres et résultats d'une fonction

Dans l'exemple précédent, paramètre et résultat étaient dans le registre `R0`, mais ça ne permet pas de faire des fonctions complexes. Normalement, on place les paramètres et le futur résultat sur la pile.

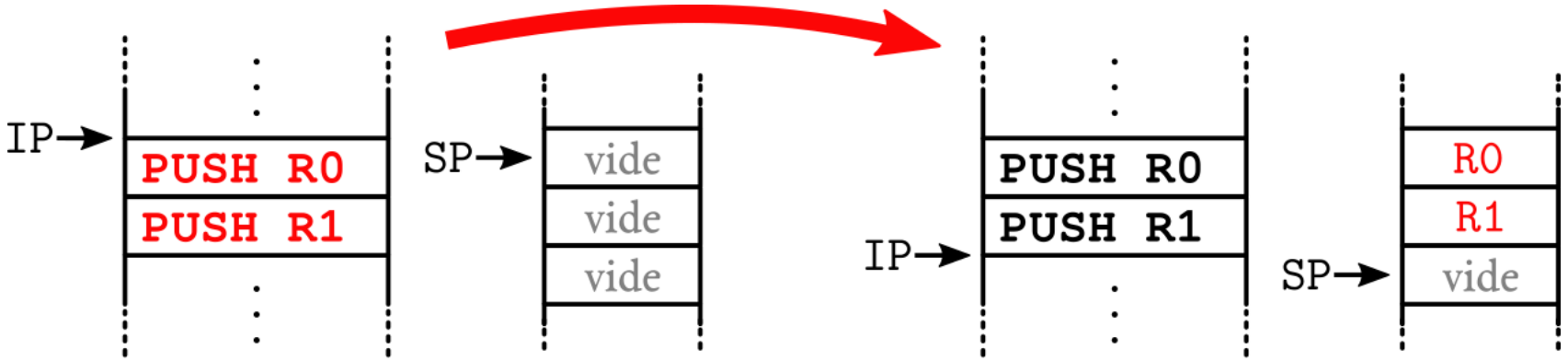
1. On empile les octets du futur résultat,
2. On empile les paramètres de la fonction,
3. On appelle la fonction,
4. On dépile les paramètres de la fonction,
5. On dépile les octets du résultat.

Il y a deux instructions :

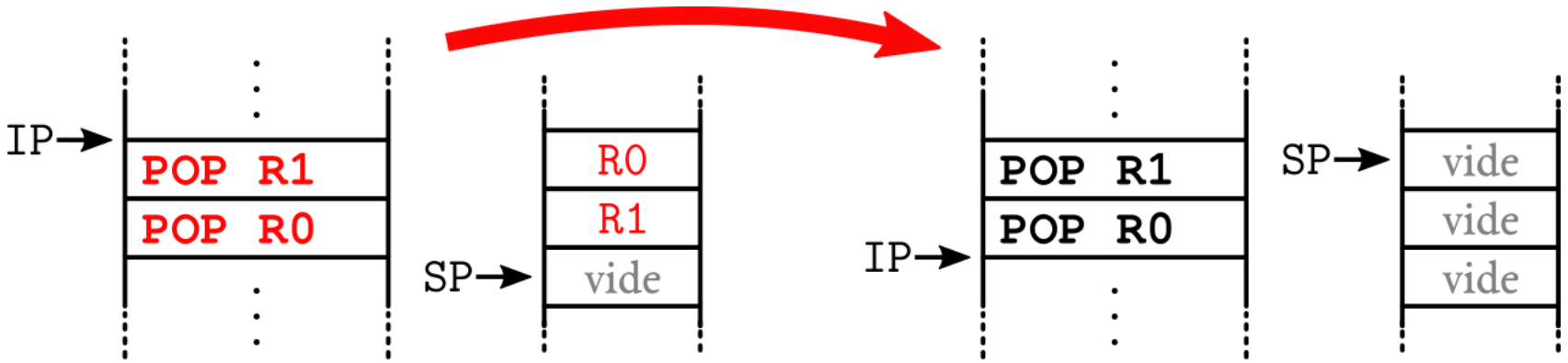
- `PUSH registre(s)` : pour chaque registre, ça le place en `[SP]` puis décrémente `SP`,
- `POP registre(s)` : pour chaque registre, ça incrémente `SP` puis récupère le registre de `[SP]`.

Pierre Nerzic

Empiler/dépiler des registres



Attention à dépiler autant de valeurs qu'empilées, avant un `RET`.



Pierre Nerzic

Exemple d'appel de fonction

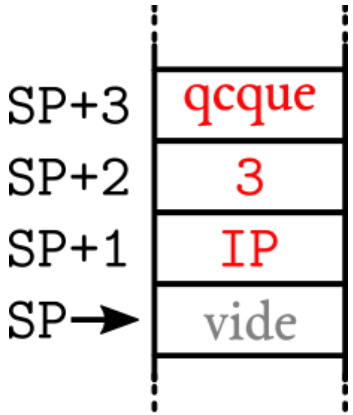
```
        PUSH R0          ; valeur quelconque pour le résultat
        LD   R0,3        ; paramètre : 3
        PUSH R0
        CALL fois2       ; appel de la fonction fois2
        POP  R0          ; enlever le paramètre
        POP  R0          ; récupérer le résultat
        OUT  R0,10       ; affichage sur le terminal

        PUSH R0          ; valeur quelconque pour le résultat
        PUSH R0          ; paramètre : valeur retournée préc.
        CALL fois2       ; 2e appel de la fonction
        POP  R0          ; enlever le paramètre
        POP  R0          ; récupérer le résultat
        OUT  R0,10       ; affichage sur le terminal
        HLT              ; fin du programme
```


Pierre Nerzic

Empilement des informations

Voici la situation de la pile quand l'exécution arrive au début de la fonction. On trouve les deux empilements de R0, le premier pour le futur résultat, le second pour le paramètre 3. Ensuite, on trouve l'adresse de retour :

```
PUSH R0 ; valeur quelconque
LD   R0,3
PUSH R0
CALL fois2
```



Rappel: la pile se remplit en allant vers le bas (adresses décroissantes), et SP désigne le prochain emplacement libre. Ainsi, les paramètres et le résultat sont au dessus de SP.

Paramètres et résultats d'une fonction, suite

Dans la fonction, il faut utiliser l'adressage indirect avec décalage basé sur SP pour récupérer les paramètres et affecter le résultat.

- Les paramètres sont en [SP+2], [SP+3]... [SP+N+1] selon le nombre N de paramètres. Le premier paramètre est celui qui a le plus grand décalage, et le dernier est [SP+2].
- La valeur du retour est en [SP+N+2]...

```
fois2:  LD   R0,[SP+2]   ; premier parametre
        ADD  R0,R0
        ST   R0,[SP+3]   ; valeur de retour
        RET              ; retour au CALL
```

Il faut remarquer que la fonction modifie la valeur de R0. Il faut toujours empiler les registres utilisés avant d'appeler une fonction.

Pierre Nerzic

Variables locales

Il est possible de réserver de la place sur la pile pour y placer des variables locales. Sur CimPU, on utilise ces deux instructions (variantes avec des nombres au lieu de registres) :

- `PUSH n` : effectue `SP = SP - n`
- `POP  n` : effectue `SP = SP + n`

Après avoir fait cela, il faut ajouter le nombre n aux adresses des paramètres et résultats. Les variables locales sont disponibles en `[SP+1]`, `[SP+2]`... `[SP+n]`

C'est relativement complexe et sera étudié dans le dernier TP de la matière.