Programmation de bas niveau
le langage de la machine

S2 - dept info

18 mars 2013

Table des matières

1 Structure d'un ordinateur

Un ordinateur est un appareil qui comporte essentiellement un *processeur*, de la *mémoire*, des *dispositifs d'entrée-sortie*.

Ces éléments interagissent :
- le processeur exécute les *instructions* qui sont dans la mémoire ;
- ces instructions effectuent des calculs, prennent et placent des données qui sont aussi en mémoire, les envoient ou les lisent sur les dispositifs d'entrée-sortie.

2 Les processeurs

De nos jours les processeurs sont très complexes, ils intègrent plusieurs coeurs, des lignes de caches, des coprocesseurs etc.

Pour comprendre ce qu'ils font, on peut regarder les premiers ordinateurs. Construits en technologies discrètes (lampes, puis transistors et circuits intégrés), ils étaient par force relativement simples. Le premier prototype d'ordinateur [1], ne comportait que 550 tubes électroniques.

Depuis, les choses ont légèrement évolué

1. le calculateur expérimental Small Scale Experimental Machine, développé en 1948 à l'Université de Manchester, était la première machine à programme enregistré en mémoire vive

année	transistors	processeur
1971	2,300	Intel 4004, premier microprocesseur
1978	29,000	Intel 8086, premiers PC
1979	68,000	Motorola 68000
1989	1,180,000	Intel 80486
1993	3,100,000	Pentium
1997	9,500,000	Pentium III
2000	42,000,000	Pentium 4
2012	1,400,000,000	Quad-Core + GPU Core i7
2012	5,000,000,000	62-Core Xeon Phi

Source : `http://en.wikipedia.org/wiki/Transistor_count`

Pour comprendre le fonctionnement des ordinateurs, nous allons donc nous ramener à des machines très simples, du niveau de celles qui existaient au début des années 60.

Fondamentalement, dans un processeur il y a des registres (circuits capables de mémoriser quelques bits d'information), des circuits combinatoires (additionneur, comparateurs, ...), et des circuits qui assurent le déroulement des différentes phases d'exécution des instructions.

Le programmmeur n'a pas forcément connaissance de tous ces éléments. Pour programmer, il se réfère au *modèle du programmeur*, c'est à dire la partie qui lui est directement accessible.

En général, il s'agit
- du registre *compteur de programme*, qui indique ou se trouve la prochaine instruction à exécuter
- de registres de travail, où sont stockés les résultats intermédiaires,
- d'indicateurs de condition, qui permettent de savoir si l'addition a eu une retenue, si la comparaison demandée a réussi ou échoué, etc.

3 Un exemple très simple

Pour les besoins pédagogiques, nous allons considérer une machine extrêmement simplifiée *quoique plus élaborée que la SSEM*, avec les caractéristiques suivantes :
- machine à mots de 16 bits
- nombres en binaire complément à deux
- instructions sur 16 bits
- architecture Von Neumann (les données et les instructions sont dans le même espace mémoire)
- adresses sur 12 bits (capacité d'adressage : 4096 mots de 16 bits)
- compteur ordinal 16 bits
- accumulateur 16 bits
- opérations arithmétique : addition et soustraction.
- adressage direct et indirect.

4 Jeu d'instructions

Les instructions de cette machine sont codées sur 16 bits, dans un format unique :
- les 4 bits de poids fort indiquent le code opération
- les 12 bits de poids faible contiennent l'opérande.

code 4 bits	opérande 12 bits
`----`	`------------`

Trois types d'opérandes sont utilisés, selon les instructions. Par exemple il y a 3 variantes de l'instruction de chargement dans l'accumulateur :
- `loadi 42` place la valeur 42 dans l'accumulateur. Le nombre 42 est une *valeur immédiate*.
- `load 23` met dans l'accumulateur le contenu du mot mémoire d'adresse 23. Le nombre 23 est une *adresse directe*.
- `loadx 123` met dans l'accumulateur le contenu du mot mémoire dont l'adresse est contenue dans le mot d'adresse 123. Le nombre 123 est une `adresse indirecte`. On peut considérer que le mot d'adresse 123 est un pointeur.

4.1 Table des instructions

Par défaut, toutes les instructions ont pour effet d'incrémenter le compteur de programme (`Cp++`), à l'exception des instructions de saut.

Pour `halt`, l'opérande est ignoré.

Code	Mnémonique	Description	Action	`Cp =`
0	`loadi` *imm12*	chargement immediat	`Acc = ext(imm12)`	`Cp + 1`
1	`load` *adr12*	chargement direct	`Acc = M[adr12]`	`Cp + 1`
2	`loadx` *adr12*	chargement indirect	`Acc = M[M[adr12]]`	`Cp + 1`
3	`store` *adr12*	rangement direct	`M[adr12] = Acc`	`Cp + 1`
4	`storex` *adr12*	rangement indirect	`M[M[adr12]] = Acc`	`Cp + 1`
5	`add` *adr12*	addition	`Acc += M[adr12]`	`Cp + 1`
6	`sub` *adr12*	soustraction	`Acc -= M[adr12]`	`Cp + 1`
7	`jmp` *adr12*	saut inconditionnel		`adr12`
8	`jneg` *adr12*	saut si négatif		`Acc<0 ? adr12 : Cp+1`
9	`jzero` *adr12*	saut si zero		`Acc==0 ? adr12 : Cp+1`
A	`jmpx` *adr12*	saut indirect		`M[adr12]`
B	`call` *adr12*	appel	`M[adr12] = Cp+1`	`M[adr12]+1`
C	`halt 0`	arrêt		
D		op. illégale	erreur	
E		op. illégale	erreur	
F		op. illégale	erreur	

Commentaires

- *adr12* (resp. *imm12*) désigne l'adresse (resp. la valeur immédiate) encodée sur les 12 bits de l'instruction
- l'instruction `loadi` procède à une *extension de signe* de *imm12* : le bit de poids fort de la valeur immédiate est copiée dans 4 bits de poids fort de l'accumulateur. Par exemple l'intruction `loadi -1` est codée `0000 1111 1111 1111` en binaire. Lors de l'affectation dans l'accumulateur 16 bits, le bit de signe de la valeur immédiate est propagé de façon à obtenir la valeur `1111 1111 1111 1111` (qui représente -1 sur 16 bits) dans l'accumulateur.
- lors de l'exécution des opérations indirectes (`loadx`, `storex`,`jmpx`) le contenu de `M[adr12]` qui est sur 16 bits est interprété comme une adresse sur 12 bits. Une erreur est détectée si les 4 bits de poids ne sont pas nuls, et entraîne l'arrêt du processeur.
- 3 codes ne sont pas utilisés. Ils pourront servir à définir de nouvelles opérations.

5 Notation des programmes

Le chargement d'un programme consiste à donner un contenu initial à la mémoire. Ce contenu peut être décrit en hexadécimal :

```
0009 5005 6006 3007 C000 0005 0003 0000
```

5.1 Utilisation de mnémoniques

En décodant les 5 premiers mots, on verrait qu'il s'agit d'un programme

adresse	contenu	mnémonique	opérande
0	0009	loadi	9
1	5005	add	5
2	6006	sub	6
3	3007	store	7
4	C000	halt	0

qui charge la valeur 9 dans l'accumulateur, lui ajoute le contenu du mot d'adresse 5, retranche celui de l'adresse 6 et range le résultat à l'adresse 7.

A ces adresses on trouve initialement les valeurs 5, 3 et 0, ce qu'on peut noter

adresse	contenu	directive	opérande
5	0005	word	5
6	0003	word	3
7	0000	word	0

La *directive* `word` indique qu'un mot mémoire est réservé, avec une certaine valeur initiale sur 16 bits.

5.2 Utilisation d'étiquettes

Écrire des programmes avec la notation ci-dessus serait fastidieux : ajouter/enlever des instructions décale les variables qui sont située plus loin, et oblige à modifier le codage des instructions qui y font référence.

Pour cela on utilise des noms symboliques, les *étiquettes*, pour désigner les adresses, et un programme prend l'allure suivante

```
#
# premier programme
#
          load   9
          add    premier
          sub    second
          store  resultat
          halt   0
#
# réservation des variables
#
premier   word   5
second    word   3
resultat  word   0
```

La traduction de ce texte source est faite par un programme appelé *assembleur* : il assemble entr'elles les lignes rédigées par le programmeur, écrites en *langage d'assemblage*[2], et comportant un mélange d'instructions sous forme mnémonique et de directives de réservation.

Remarques

- L'étiquette est facultative et commence obligatoirement en colonne 1 si elle est présente.
- Si elle est absente, il doit y avoir au moins un espace avant le mnémonique ou la directive `word`.
- l'étiquette peut aussi être seule sur la ligne. Elle se réfère alors au prochain mot, donnée ou instruction.
- un texte source contient aussi des commentaires, d'autant plus utiles que la programmation est laborieuse.

Exercice : traduire les affectations

- A = B
- A = A + 1
- A = B + C - 1

6 Tests et décisions

Le processeur ne comporte que deux instructions pour tester des conditions : regarder si le contenu de l'accumulateur est nul (`jzero`) ou si il est négatif (`jneg`). Ces instructions sont des *sauts conditionnels* : si la condition indiquée est vraie, le déroulement se poursuit à l'adresse désignée, sinon on passe à l'instruction suivante.

En pseudo-code, cela correspondrait à

si `Acc` est nul, aller à *adresse*
si `Acc` est négatif, aller à *adresse*

2. par un abus de langage courant, on parle souvent de "programmer en assembleur"

C'est rudimentaire, mais suffisant pour exprimer des si-alors-sinon. Voyons par exemple un algorithme qui calcule la valeur absolue d'un nombre.

Sous la forme habituelle d'algorithme structuré :

```
Donnée:
    X nombre
Résultat
    R nombre
début
    si X >= 0
    alors
      |   R = X
    sinon
      |   R = -X
fin
```

quand la condition est fausse, on va exécuter le bloc "sinon". Quand elle est vraie, on exécute le bloc "alors", et on contourne le bloc "alors" Ceci peut être exprimé sous forme de séquence d'instructions avec quelques sauts et étiquettes :

```
#
# Séquence de pseudo-instructions pour
# calcul de  R = abs(X)
#
     si X < 0 aller à OPPOSE
     calculer R = X
     aller à SUITE
OPPOSE:
     calculer R = - X
SUITE:
```

Important : l'étape d'écriture en séquence de pseudo-instructions comme ci-dessus est `indispensable` pour espérer arriver à un programme à moindre effort.

Les programmeurs expérimentés, qui donnent l'impression d'écrire directement en langage d'assemblage, font figurer le pseudo-code en commentaires. En réalité, ils écrivent les commentaires d'abord, qui leur servent de guide pour aboutir aux instructions machine.

Une première version

```
1  # Programme pour calcul de la valeur absolue
2  # version non optimisée
3
4         load  X        # si X<0 aller à OPPOSE
5         jneg OPPOSE
6
7         load  X        # R = X
8         store R
9
10        jmp   SUITE    # aller à SUITE
11 OPPOSE
12        loadi 0         # R = 0-X
13        sub   X
14        store R
15 SUITE
16        halt  0
17 #
18 # données
19 #
20 X       word  -3
21 R       word  0
```

Exercice. Remarquez que
- lors du second chargement de X, sa valeur est déjà dans l'accumulateur.
- les deux alternatives se finissent par la même instruction.

et profitez-en pour *optimiser* ce programme. Précisez le gain effectué en espace mémoire, et en nombre d'instructions effectuées.

Exercice. Ecrire la séquence d'instructions en pseudo-code qui qui calcule le maximum de deux nombres A et B. Traduire ensuite en instructions machine. Remarque : comparer, c'est étudier la différence.

Exercice. Pogramme qui ordonne deux nombres A et B. (Après l'exécution A et B contiendront respectivement le max et le min des deux valeurs initiales).

7 Faire des boucles

Soit à écrire un programme qui calcule la somme S des entiers de 1 à N.
Sour forme d'algorithme classique

```
donnée     N nombre
résultat   S nombre
variable   K nombre
début
    S = O
    K = 1
    tant que K <= N
     faire
       |    S = S + K
       |    K = K + 1
fin
```

L'exécution de la boucle commence par un test. Si la condition est vraie, le corps de boucle est exécuté, et on revient au test. Quand la condition du test est fausse, le corps de boucle est contourné.
Sous forme de séquence de pseudo-instructions

```
BOUCLE
     S = 0
     K = 1
     si K > N aller à SUITE
     S = S + K
     K = K + 1
     aller à BOUCLE
SUITE
     ...
```

Le test revient à étudier le signe de la différence N-K.
D'où le programme

```
1  #
2  # Calcul de la somme des entiers de 1 à N
3  #
4      loadi 0     # S=0
5      store S
6
7      loadi 1     # K=1
8      store K
9
10 BOUCLE          # si K>N aller à suite
11     load  N     # - calcul N-K
12     sub   K
13     jneg  SUITE
```

```
14
15      load   S      # S = S+K
16      add    K
17      store  S
18
19      loadi   1     # K = K+1
20      add     K
21      store   K
22      jmp    BOUCLE
23
24  SUITE
25      halt  0
26  #
27  # variables
28  #
29  N    word   5
30  K    word   0
31  S    word   0
```

Exercice : quand la borne de la boucle est fixée, par exemple "pour K de 0 à 4", il y a diverses façons de traduire la boucle selon qu'on fait le test de fin de boucle en premier ou après la première exécution du corps, et qu'on effectue la comparaison avec 4 ou avec 5. Comparez-les.

Exercice : programme qui multiplie deux valeurs (additions successives)

Exercice : programme qui divise deux valeurs (soustractions successives) et fournit le quotient et le reste.

Exercice : programme qui calcule la factorielle d'un nombre.

Exercice : programme qui trouve le plus petit diviseur non trivial d'un nombre (plus grand que 1).

8 Utiliser des tableaux

Les instructions `loadx` et `storex` chargent ou rangent un mot mémoire à une adresse qui est définie par une variable en mémoire. Exemple, si le mot d'adresse 23 contient 42, l'instruction `loadx 23` aura le même effet que `load 42`. Ceci revient à utiliser le mot d'adresse 23 comme un `pointeur` vers la donnée effectivement chargée.

Ceci permet de travailler avec des *tableaux*, qui sont constitués de cases consécutives :

```
T       word  0          # T[0]
        word  0          # T[1]
        word  0          # T[2]
        word  0          # T[3]
        ...
```

Un tableau de mots (indicé à partir de 0) commence à l´adresse T, sa K-ième case est à l´adresse T+K. Il suffit donc d´additionner l´`adresse de base` T et la valeur de l´indice K pour obtenir l´adresse de la case

```
        loadi   T
        add     K
```

que l´on peut ranger dans un pointeur

```
        store   PTR
```

qui donne accès à la case `T[K]`, par `loadx PTR` ou `storex PTR`.

Exemple : remplir un tableau de 5 cases avec les nombres de 0 à 4.

```
résultat: tableau T[5]
variables:
début
   pour K de 0 à 4
   faire
     |  T[K] = K
fin
```

L'instruction `T[K] = K` peut se traduire ainsi

```
      loadi  T       # PTR = & T[K]
      add    K
      store  PTR

      load   K       # *PTR = K
      storex PTR
```

Exercice Ecrire complètement le programme qui remplit le tableau.

Exercice Ecrire un programme qui calcule la somme des éléments d'un tableau.

Solution

```
1  #
2  #  Somme des éléments d'un tableau
3  #
4  #  données  : N, T[N]
5  #  résultat : S
6
7          loadi   0       # S = 0
8          store   S
9          store   K       # K = 0
10
11         # pour K de 0 à N-1
12
13 Boucle
14         load    N       # si K==N aller à Fin
15         sub     K
16         jzero   Fin
17
18         loadi   T       # P = & T[K]
19         add     K
20         store   P
21
22         loadx   P       #  S += *P
23         add     S
24         store   S
25
26         loadi   1       # K++
27         add     K
28         store   K
29
30         jmp     Boucle
31
32 Fin
33         halt    0
34 #
35 # tableau de 5 cases
```

```
#
N       word    5
T       word    10000
        word    2000
        word    300
        word    40
        word    5
#
# variables
#
K       word    0       # indice
P       word    0       # pointeur
S       word    0       # somme
```

Exercice Ecrire un programme qui détermine le minimum des éléments d'un tableau.

Exercice Ecrire un programme qui trie les des éléments d'un tableau dans l'ordre croissant (algorithme de tri par sélection).

Solution :

```
#
# Tri d'un tableau  T[N] par sélection
#
#
#       pour i de 0 à n-1
#               imin = i
#               min = t[i]
#               pour j = i+1 à n-1
#                   si t[j] <= min
#                       imin = j
#                       min  = t[j]
#               echanger t[i] t[imin]

        loadi   0
        store   i
bcle_i
        load    i       # si i == n  fin boucle sur i
        sub     n
        jzero   fin_i

        load    i       # imin = i
        store   imin

        loadi   t       # min = t[i]
        add     i
        store   ptr
        loadx   ptr
        store   min

        loadi   1       # j = i+1
        add     i
        store   j
bcle_j
        load    j
        sub     n
        jzero   fin_j

        loadi   t       # tj = t[j]
```

```
39          add     j
40          store   ptr
41          loadx   ptr
42          store   tj
43
44          load    min     # si tj > min  avancer j
45          sub     tj
46          jneg    incr_j
47
48          load    j       # imin = j
49          store   imin
50
51          load    tj      # min = tj
52          store   min
53  incr_j
54          loadi   1       # j++
55          add     j
56          store   j
57          jmp     bcle_j
58  fin_j
59          # echange
60          loadi   t       # ti = t[i]
61          add     i
62          store   ptr
63          loadx   ptr
64          store   ti
65          load    min     # t[i] = min
66          storex  ptr
67
68          loadi   t       # t[imin] = ti
69          add     imin
70          store   ptr
71          load    ti
72          storex  ptr
73
74  incr_i
75          loadi   1       # i++
76          add     i
77          store   i
78          jmp     bcle_i
79  fin_i
80          halt    0
81
82  # ------------------------------
83
84  i       word    0
85  j       word    0
86  min     word    0
87  imin    word    0
88  ti      word    0
89  tj      word    0
90  ptr     word    0
91
92  # ------------------------------
93  n       word    10
94
95  t       word    1111
96          word    4444
```

```
97          word    2222
98          word    8888
99          word    5555
100
101         word    7777
102         word    0
103         word    9999
104         word    6666
105         word    3333
106 # ------------------------------------
```

Remarque : Une autre approche consiste à incrémenter à chaque étape le pointeur pour qu'à chaque étape il avance sur l'élément suivant

```
    loadi T
    store PTR       # PTR = T
boucle:
    ...
    loadi  1        #  PTR++
    add    PTR
    store  PTR
    ....
    jmp   boucle
```

9 Sous-programmes

Les instructions `call` et `jmpx` servent à réaliser des appels et retours de sous-programme.

`call adr12` sauve l'adresse de l'instruction suivante (`Cp+1`) à l'emplacement indiqué par *adr12*, et continue à l'adresse (*adr12*+1). [3].

Le premier mot d'un sous-programme est donc réservé, et contiendra l'adresse à laquelle le sous-programme devra revenir, par l'intermédiaire de l'instruction `jumpx`

Exemple : un sous-programme qui calcule le plus grand de deux nombres

```
#
# sous programme de calcul du max de 2 nombres
# données dans MaxA et MaxB
# résultat dans MaxRes
#
Max       word  0         # res. adresse de retour
          load  MaxA      # si MaxA < MaxB, aller à MaxL2
          sub   MaxB
          jneg  MaxL1

          load  MaxA      # Acc =  MaxA
          jmp   MaxL2

MaxL1     load  MaxB      # Acc =  MaxB
MaxL2     store MaxRes    # résultat
          jmpx  max       # retour
#
# paramètres / résultats
#
MaxA      word  0         # paramètre
MaxB      word  0
MaxRes    word  0
```

3. Ce type d'instruction d'appel existait sur les machines PDP/1 et PDP/4 de DEC, et HP 1000 de Hewlett-Packard

Les conventions d'appel sont les suivantes
– placer les valeurs des deux paramètres dans MaxA et MaxB
– effectuer l'appel `call Max`
– le résultat est disponible dans MaxRes
Exemple : séquence de calcul pour `R = max(max(X,Y),Z)`

```
load    X
store   MaxA
load    Y
store   MaxB
call    Max    # calcul temp = max(X,Y)
load    MaxRes
store   MaxA
load    Z
store   MaxB
call    Max    # calcul max(temp,Z)
load    MaxRes
store   R
```

Exercice Ecrire un sous programme de multiplication par addition successives

Exercice Ecrire un sous programme de calcul de factorielle.

Exercice Ecrire un sous programme de division par soustractions successives

Exercice Ecrire un sous programme de calcul de coefficients binomiaux $C_n^p = \frac{n!}{p!(n-p)!}$

10 Passage de pointeurs

Une action qui doit modifier ses paramètres (par exemple échanger le contenu de deux variables) sera écrite sous forme d'un sous-programme qui reçoit les *adresses* des données à échanger.
Exemple

```
Swap    word   0
        loadx  SwapA     # tmp = *A
        store  SwapTmp

        loadx  SwapB     # *A = *B
        storex SwapA

        load   SwapTmp   # *B = tmp
        storex SwapB
        callx  Swap
#
# paramètres reçus
#
SwapA   word 0
SwapB   word 0
#
# variable locale
#
SwapTmp word 0
```

11 Utilisation d'une pile

La technique précédente ne convient pas aux fonctions qui s'appellent elles-mêmes, directement ou indirectement. Exemple, la fonction récursive définie par

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1)+fib(n-2)
```

en effet, avec la technique précédente, on ne réserve en mémoire que des emplacements (adresse de retour, paramètres, variables locales) pour une seule invocation de chaque fonction.

Une solution classique est d'utiliser une pile, composée de "cadres" contenant les données propres à chaque invocation de la fonction.

Voici un exemple de programme utilisant cette technique. Les conventions d'appels du sous-programme FIB(N) sont

- le paramètre d'entrée N est dans l'accumulateur
- la variable SP pointe sur un espace libre assez grand (environ 2N mots) qui sert de pile d'exécution
- au retour le résultat sera dans l'accumulateur

Le déroulement de la fonction est schématiquement le suivant

```
FIB(N)
début
    si N < 1
       alors retourner N
    sauvegarder N et l'adresse de retour dans la pile
    calculer f1 =  FIB(N-1)
    sauvegarder f1 dans la pile
    calculer f2 =  FIB(N-2)
    récuperer N, l'adresse de retour et f1
    retourner la valeur f1 + f2
fin
```

```
1  #
2  # Calcul de fibonacci
3  # récursif
4  #
5
6  # algo
7  #  fib(n):
8  #     si n < 2
9  #        retourner n
10 #     a = fib(n-1)
11 #     b = fib(n-2)
12 #     retourner a+b
13
14 # conventions d'appel
15 #
16 #  En entrée : Acc contient n
17 #              cadre   contient l'adresse d'un espace en sommet de pile
18 #                      disponible pour l'instance de la fonction
19 #  En sortie : Acc contient n
20
21
22 main
23          loadi    pile
24          store    cadre
25
26          load     N   # calcul fib(N)
27          call     fib
28          store    R
29          halt     0
30
31 N         word     8
32 R         word     0
```

```
33 # ------------------------------------
34
35
36 fib     word    0       # adresse de retour
37
38         store   tmp     # copie de N
39         loadi   1       # si N>1, aller à cas_general
40         sub     tmp
41         jneg    cas_general
42         load    tmp     # sinon retourner N
43         jmpx    fib
44
45 cas_general
46         load    tmp
47         storex  cadre   # sauvegarde N dans cadre[0]
48
49         loadi   1       # sauvegarde adresse de retour
50         add     cadre   # dans cadre[1]
51         store   ptr
52         load    fib
53         storex  ptr
54                                 # contenu du cadre de pile :
55                                 #    cadre[0] = n
56                                 #    cadre[1] = adresse de retour
57
58         loadi   2       # allocation nouveau cadre
59         add     cadre
60         store   cadre
61         loadi   -1      # appel fib(n-1)
62         add     tmp
63         call    fib
64         store   res     # copie résultat
65         loadi   -2      # libération cadre
66         add     cadre
67         store   cadre
68
69         loadi   2       # sauvegarde résultat fib(n-1) dans cadre[2]
70         add     cadre
71         store   ptr
72         load    res
73         storex  ptr
74
75         loadx   cadre   # recupération de N dans tmp
76         store   tmp
77                                 # contenu du cadre de pile :
78                                 #    cadre[0] = n
79                                 #    cadre[1] = adresse de retour
80                                 #    cadre[2] = valeur de fib(n-1)
81
82         loadi   3       # allocation nouveau cadre
83         add     cadre
84         store   cadre
85         loadi   -2      # appel fib(n-2)
86         add     tmp
87
88         call    fib
89         store   res     # copie résultat fib(n-2)
90         loadi   -3      # libération cadre
```

```
91
92          add     cadre
93          store   cadre
94          loadi   1       # récupération de l'adresse de retour
95          add     cadre
96          store   ptr
97          loadx   ptr
98          store   fib
99
100         loadi   2       # calcul fib(n-1) + fib(n-2)
101         add     cadre
102         store   ptr
103         loadx   ptr
104         add     res
105
106         jmpx fib
107
108 # ------------------------------------
109 ptr     word    0
110 tmp     word    0
111 res     word    0
112 # ------------------------------------
113 cadre   word    0       # pointeur de cadre de pile
114 pile    word    0       # la pile commence ici
115                         # ...
```