TP3 - Fonctions et tableaux

P. Nerzic

Ce TP est à rendre, car il compte pour une note de travaux pratiques. Il faudra donc déposer une feuille de réponses faite pendant la séance sur Moodle.
Le travail est personnel, la note est individuelle. Toute tentative pour copier les réponses de quelqu'un d'autre sera sanctionnée par un zéro (Discord et autres). Et le TP doit être commencé et fini uniquement pendant les séances de TP, pas à la maison. Par contre, vous pouvez communiquer verbalement pour vous aider, pendant les séances. Le deuxième DS portera sur le contenu des TP.

☛ Téléchargez ReponsesTP3.txt et mettez votre nom/prénom et groupe de TP aux endroits indiqués. Ne modifiez surtout pas la structure de ce fichier.

Le TP3 porte sur l'écriture de fonctions et l'utilisation de tableaux.

☛ Ouvrez l'URL https://perso.univ-rennes1.fr/pierre.nerzic/IntroArchi/CimPU/.

☛ Mettez CimPU en niveau 4.

1. Fonctions

En assembleur, une *fonction*, ou *procédure*, ou *sous-programme*, est une partie du programme qui peut être appelée à plusieurs reprises, de différents endroits du programme principal, pour effectuer une tâche particulière. Le programme principal appelle la fonction, elle s'exécute et quand elle est finie, le contrôle revient dans le programme principal juste après l'appel de la fonction.

Le processeur possède un mécanisme pour se souvenir où revenir après la fonction. Le principe est simple : quand il appelle une fonction, le processeur note où il en était dans l'exécution du programme, c'est à dire la valeur de `IP` au moment de l'appel – on l'appelle *adresse de retour*. Quand la fonction se termine, `IP` est réaffecté avec cette adresse de retour.

Comme une fonction peut elle-même en appeler d'autres, cela peut faire plusieurs adresses de retour à mémoriser. Le mécanisme pour les mémoriser et les rappeler dans l'ordre s'appelle une *pile* et le principe, c'est qu'à chaque appel de fonction, l'adresse de retour est mise au dessus des autres (on dit *empiler*). Quand on revient d'une fonction, il suffit de retirer l'adresse qui est tout au dessus (on dit *dépiler*).

La pile est gérée par un registre spécial, `SP`. C'est un registre qui contient une adresse, celle du sommet de la pile. Il est initialisé à l'adresse 255, à la fin de la mémoire. La pile se remplit vers le bas de la mémoire. Empiler une adresse de retour fait décrémenter le registre, et dépiler le fait incrémenter.

Il y a deux instructions à connaître :

- `CALL adresse` : elle empile la valeur de `IP` puis fait un saut à l'adresse indiquée.
- `RET` : elle dépile l'adresse de retour dans `IP`.

1.1. Exemples de fonctions

☛ Assemblez ce programme :

```
main:       ;; programme principal
            ; DSKY en mode 10 (affichage de nombres)
            LD  R0, 10
            OUT R0, 255
```

```
            ; affichage d'un 0
            LD  R0, 0
            OUT R0, 10
            ; appel de la fonction 1 dans le programme principal
            CALL fonction1
            ; affichage d'un 0
            LD  R0, 0
            OUT R0, 10
            ; appel de la fonction 2 dans le programme principal
            CALL fonction2
            ; fin du programme
            HLT

fonction2:  ; affichage d'un 2
            LD  R0, 2
            OUT R0, 10
            RET

fonction1:  ; affichage d'un 1
            LD  R0, 1
            OUT R0, 10
            ; appel de la fonction 2 dans la fonction 1
            CALL fonction2
            ; affichage d'un 1
            LD  R0, 1
            OUT R0, 10
            RET
```

Il y a toujours un label pour nommer la fonction. D'autre part, les fonctions peuvent être dans un ordre quelconque, mais pas en premier dans un programme, parce que l'exécution commence toujours en 0, donc dans le programme principal. Ou alors, il faudrait rajouter un `BRA main` au tout début… c'est ce que fait le compilateur C.

☛ Cliquez sur le bouton `Reset` puis appuyez sur `Next` jusqu'au retour de `fonction2` au milieu de `fonction1` en regardant ce qui se passe :

- la succession des affichages à cause des appels des fonctions,
- l'endroit où se trouve `IP` (programme principal puis dans les fonctions),
- comment évoluent `SP` et `IP` lors des `CALL` et des `RET`,
- et ce qu'il y a dans la RAM au niveau de `SP`.

Vous allez voir que l'adresse de retour est située en plein milieu de l'instruction `CALL`. C'est parce que dans CimPU, `IP` est empilé avant d'aller chercher l'adresse de la fonction. Et donc `RET` incrémente `IP` après avoir dépilé l'adresse de retour. Dans les vrais processeurs, l'adresse de retour est celle de l'instruction qui suit le `CALL`.

NB: la totalité des lignes affichées est visible dans la section *Périphériques*.

1.2. Paramètres et résultats

Les fonctions peuvent recevoir des paramètres et elles peuvent retourner des résultats. Le plus simple est d'utiliser les registres pour fournir les paramètres et recevoir les résultats, mais ça limite ces fonctions à seulement deux paramètres. On peut faire beaucoup mieux, avec la pile, mais c'est trop complexe pour ce TP.

☛ Assemblez ce programme :

```
;; programme principal
main:       ; DSKY en mode 10 (affichage de nombres)
            LD  R0, 10
            OUT R0, 255
            ; appels de fonctions
            LD  R0, 99
            CALL afficherR0
            CALL incrementerR0
            CALL afficherR0
            LD  R0, 199
            CALL incrementerR0
            CALL afficherR0
            ; fin du programme
            HLT

;; affichage de R0 sur le port 10
afficherR0:
            OUT R0, 10
            RET

;; ajoute 1 à R0
incrementerR0:
            INC R0
            RET
```

☛ Exécutez ce programme, soit à pleine vitesse, soit instruction par instruction.

1.3. Sauvegarde des registres sur la pile

Mettons qu'une fonction ait besoin des registres pour faire ses calculs, et qu'on ait aussi besoin de ces registres dans le programme principal. Il faut donc que la fonction préserve leurs valeurs, et les rétablisse avant le retour. On pourrait utiliser des variables dans la mémoire, mais il y a mieux avec la pile. Voici deux instructions pour cela :

- `PUSH registre` : empile la valeur du registre,
- `POP registre` : dépile le sommet de la pile dans le registre.

On utilise `PUSH` pour « sauver » une valeur qui est dans un registre, quand on a besoin de ce registre pour faire autre chose, mais qu'on ne veut pas perdre la valeur qu'il contenait. Quand on a fini avec le registre, on récupère son ancienne valeur avec `POP`.

NB: on n'est pas obligé de dépiler dans le même registre. On peut empiler `R0` et dépiler `R1` ; ça va

transférer la valeur de `R0` dans `R1`. C'est comme si vous tenez une balle dans la main droite, vous la posez par terre, puis vous la reprenez de la main gauche.

☛ Assemblez ce programme :

```
;; programme principal
main:       ; DSKY en mode 11 (affichage de caractères ascii)
            LD  R0, 11
            OUT R0, 255
            ; int i = 26
            LD  R0, 26        ; nombre de boucles
.repeter:   ; répeter {
            ;   R0-- => appel de la fonction avec 25..0
            DEC R0
            ;   afficher(i)
            CALL afficherR0
.jusqua:    ; } jusqu'à (R0 == 0)
            CMP R0, 0
            BNE .repeter
            ; fin du programme
            HLT

;; affichage de R0 sur le port 11
afficherR0:
            PUSH R0           ; sauver la valeur de R0 sur la pile
            ADD R0, 'A'       ; ajouter le code ascii de la lettre A
            OUT R0, 11        ; afficher le caractère obtenu
            POP R0            ; remettre R0 comme il était
            RET
```

Remarquez les labels qui commencent par un point. Ils sont « locaux » à la fonction `main`. On verra plus loin que ça permet de reprendre les mêmes noms de labels dans des fonctions différentes, sans les mélanger.

Dans la fonction `afficherR0`, on transforme en code ASCII l'entier 0..25 qui est dans `R0`. Les codes ASCII des lettres sont entre `'A'`=65 et `'Z'`=90. Quand on ajoute `'A'` à un entier i compris entre 0 et 25, on obtient le code ASCII de la lettre correspondante.

Comme on utilise `R0` pour faire ce calcul, et qu'il faut quand même garder `R0` pour les itérations du programme principal, on sauve ce registre sur la pile. On n'aurait pas besoin si on utilisait une variable globale.

Deux constatations :

a. Dans une fonction, il est recommandé de sauver tous les registres utilisés. Une fonction doit faire son travail de manière invisible pour le programme qui l'appelle.
b. Il faut impérativement qu'il y ait autant de `POP` que de `PUSH` dans une fonction, sinon l'instruction `RET` ne pourra pas récupérer l'adresse de retour.

☛ Mettez le `POP R0` en commentaire dans le programme et voyez ce qui se passe. C'est très rapide. CimPU interdit de faire un retour de fonction à une adresse qui ne fait pas partie d'un programme. Ici, c'est une valeur de registre qui a été empilée au dessus de l'adresse de retour. Si on ne fait pas

le `POP` correspondant, alors le `RET` dépilera une mauvaise adresse. CimPU empêche ça.

1.4. Exercice : affichage dans l'ordre A..Z

☛ Modifiez le programme précédent pour qu'il affiche les lettres dans l'ordre croissant A..Z. Vous avez deux possibilités, choisissez celle que vous préférez :

- soit vous modifiez la boucle dans le programme principal,
- soit vous modifiez le calcul du code à afficher dans la fonction, `'Z' - R0` avec simplement un `NEG R0` pour calculer `-R0`.

☛ Une fois qu'il marche, copiez-collez ce programme dans `ReponsesTP3.txt`.

1.5. Exercice : échanger `R0` et `R1` via la pile

Les instructions `PUSH` et `POP` permettent de placer les registres sur la pile. On peut dépiler dans l'ordre qu'on veut, du moment qu'on dépile autant de valeurs empilées.

L'exercice consiste à échanger `R0` et `R1` en utilisant uniquement la pile.

☛ Complétez ce programme :

```
;; programme principal
main:       ; DSKY en mode 10 (affichage de nombres)
            LD  R0, 10
            OUT R0, 255
            ; R0 = 14, R1 = 23
            LD  R0, 14
            LD  R1, 23
            ; appel de la fonction
            CALL swap
            ; affichages
            OUT R0, 10       ; devra afficher 23
            OUT R1, 10       ; devra afficher 14
            ; fin du programme
            HLT

;; échange les valeurs de R0 et R1
swap:       ; TODO au retour, R0 = ancienne valeur de R1, et réciproquement
            ; contrainte : uniquement des PUSH et des POP
            RET
```

☛ Une fois qu'il marche, copiez-collez ce programme dans `ReponsesTP3.txt`.

NB: L'instruction `PUSH` accepte la forme `PUSH R0,R1` ou `PUSH R1,R0`, mais l'ordre des registres n'a aucune importance, dans tous les cas `R0` est empilé en premier puis `R1`. Il y a aussi `POP R0,R1` et `POP R1,R0`, dans les deux cas `R1` est dépilé en premier, puis `R0`.

Il faut retenir de cet exercice que si on ne dépile pas dans l'ordre inverse de l'empilement, les valeurs des registres sont interverties.

2. Tableaux

Dans un microprocesseur, un tableau est simplement une succession de N octets à partir d'une certaine adresse ADR. On accède à la case n°i en accédant à l'adresse $ADR + i$, i étant compris entre 0 et $N - 1$. Quand l'indice i est une variable, il faut employer un *mode d'adressage* particulier, aperçu au TP1, l'adressage *indirect avec décalage*.

On définit un tableau ainsi, soit initialisé, soit non initialisé :

```
; char T1[] = { contenu du tableau, ... };
T1:         DB contenu du tableau, ...

; char T2[n];
T2:         RB n ; avec n = taille du tableau
```

On écrit `LD R0,[R1+T1]` pour accéder à la case d'indice `R1` du tableau `T1`.
NB: on ne peut pas écrire `LD R0,[T1+R1]`, le registre doit toujours être avant la constante `T1`.

☛ Essayez ce programme :

```
;; programme principal
main:       ; DSKY en mode 11 (affichage de caractères ascii)
            LD  R0, 11
            OUT R0, 255
            ; écrire le message sur le port 11
            LD  R0, 0           ; indice du premier caractère
            LD  R1, [message]   ; premier caractère dans R1
.tantque:   ; tantque (R1 != 0)
            CMP R1, 0
            BEQ .fintantque
.faire:     ; {
            ;    afficher le caractère lu précédemment
            OUT R1, 11
            ;    indice de la case suivante
            INC R0
            ;    lire la case suivante dans R1, son indice est dans R0
            LD  R1, [R0+message]
            ; }
            BRA .tantque
.fintantque:
            ; afficher un retour à la ligne final
            LD  R1, '\n'
            OUT R1, 11
            ; fin du programme
            HLT

message:    DB "Bonjour, comment ca va ?", 0
```

Dans ce programme `R0` joue le rôle d'un indice dans le tableau `message`. L'octet à afficher est désigné par `[R0+message]`. En langage C, on écrirait `char R1 = message[R0];` avec `int R0 =`

`0;` au début.

La boucle de ce programme s'arrête quand elle rencontre un code nul. C'est le caractère spécial qui marque la fin d'une chaîne en langage C.

Les données du tableau peuvent être écrites sous la forme d'une chaîne, sauf qu'en assembleur, le code nul final, un `0`, est à écrire explicitement, alors qu'en langage C, il est mis automatiquement.

2.1. Variante

Le programme précédent peut aussi être écrit ainsi :

```
;; programme principal
main:       ; DSKY en mode 11 (affichage de caractères ascii)
            LD  R0, 11
            OUT R0, 255
            ; écrire le message sur le port 11
            LD  R0, message     ; adresse du premier caractère
            LD  R1, [R0]        ; premier caractère dans R1
.tantque:   ; tantque (R1 != 0)
            CMP R1, 0
            BEQ .fintantque
.faire:     ; {
            ;    afficher le caractère lu précédemment
            OUT R1, 11
            ;    adresse de la case suivante
            INC R0
            ;    lire la case suivante dans R1, son adresse est dans R0
            LD  R1, [R0]
            ; }
            BRA .tantque
.fintantque:
            ; afficher un retour à la ligne final
            LD  R1, '\n'
            OUT R1, 11
            ; fin du programme
            HLT

message:    DB "Bonjour, comment ca va ?", 0
```

Dans cette variante, `R0` n'est plus un indice, mais directement l'adresse du caractère courant. On dit que c'est un *pointeur*. En langage C, on écrirait `char R1 = *R0;` avec `char* R0 = &message;` au début (le `&` n'est pas nécessaire pour un tableau).

En assembleur, on préfère écrire des programmes qui travaillent avec des pointeurs plutôt qu'avec des indices. L'utilisation de pointeurs est généralement plus efficace, car les indices obligent à faire davantage d'additions. L'accès mémoire `[R0+adr]` de la variante indice coûte une addition de plus que l'accès indirect `[R0]` de la variante pointeur.

2.2. Exercice : transformation en fonction

☛ Transformez le programme précédent en déplaçant ce qu'il faut dans une fonction. Cette fonction existe vraiment, elle s'appelle `puts` en langage C (tapez `man puts` dans un shell) :

```
;; programme principal
main:       ; DSKY en mode 11 (affichage de caractères ascii)
            LD  R0, 11
            OUT R0, 255
            ; écrire le message sur le port 11
            LD  R0, message
            CALL puts
            ; fin du programme
            HLT

message:    DB "Bonjour, comment ca va ?", 0


;; fonction puts(R0)
;; affiche la chaîne située en R0 sur le port 11
puts:       ; TODO sauver les registres utilisés
            ; TODO recopier tout ce qu'il faut du programme précédent
            ; TODO restaurer les registres
            RET
```

☛ Une fois qu'il marche, copiez-collez ce programme dans `ReponsesTP3.txt`.

2.3. Exercice : lecture d'une chaîne

On continue sur le programme précédent. On va programmer la fonction `gets` qui existe vraiment en langage C. Elle demande à l'utilisateur de taper une ligne de texte et l'enregistre en mémoire.

Cette fonction doit lire les caractères venant du port 11 et les placer successivement dans la mémoire. Elle s'arrête quand elle rencontre un retour à la ligne (`'\n'`). Elle le remplace par le code 0.

☛ Complétez ce programme, en reprenant la fonction `puts` de l'exercice précédent :

```
;; programme principal
main:       ; DSKY en mode 11 (affichage de caractères ascii)
            LD  R0, 11
            OUT R0, 255
            ; afficher un message d'invite (prompt)
            LD  R0, invite
            CALL puts
            ; lire une ligne dans la zone réservée en mémoire (buffer)
            LD  R0, ligne
            CALL gets
            ; ré-afficher cette ligne (en écho)
            ; R0 n'a pas été changé, donc c'est toujours ligne
```

```
            CALL puts
            ; fin du programme
            HLT

invite:     DB "Saisissez une ligne", 0
ligne:      RB 40


;; fonction puts(R0)
;; affiche la chaîne située en R0 sur le port 11
puts:       ; TODO recopier la fonction de l'exercice précédent


;; fonction gets(R0)
;; enregistre les codes ascii venant du port 11 dans la mémoire à partir de R0
gets:       ; TODO sauver les registres utilisés
            ; lire le premier caractère du port 11 dans R1
            IN  R1, 11
.tantque:   ; TODO tant que R1 != '\n' {
            ;    TODO stocker R1 dans la case [R0] du tableau
            ;    TODO adresse de la case suivante
            ;    TODO lire le prochain caractère du port 11 dans R1
            ; }
            BRA .tantque
.fintantque:
            ; mettre un code nul à l'adresse donnée par R0, à la place du \n
            LD  R1, 0
            ST R1, [R0]
            ; TODO restaurer les registres
            RET
```

On voit l'utilité des labels locaux qui permettent de reprendre les mêmes noms, mais dans des fonctions différentes.

☛ Une fois qu'il marche, copiez-collez ce programme dans `ReponsesTP3.txt`.

Vous avez remarqué qu'on réservait 40 octets pour la chaîne. Et si l'utilisateur tapait une chaîne plus longue... C'est pour cette raison que la fonction `gets` du langage C est très déconseillée, comme toutes celles qui ne gèrent pas la taille du buffer, parce qu'elles sont la cause de graves faiblesses dans les logiciels, permettant des attaques de hackers.

☛ Modifiez la taille de la ligne, mettez `RB 10` et relancez le programme. Il ne tarde pas à s'arrêter parce que CimPU surveille les accès mémoire. Il sait que telles et telles cases sont des données ou des instructions. Ce n'est pas toujours le cas dans un processeur actuel, parce que les cases mémoire sont indifférenciées.

☛ Maintenant, intervertissez les deux tableaux :

```
ligne:      RB 10
invite:     DB "Saisissez une ligne", 0
```

puis relancez le programme... Ça semble avoir résolu le problème. Seulement si vous réaffichez le message d'invite, vous verrez qu'il est cassé. La fonction `gets` a rempli plus que le tableau prévu, elle a écrasé le début du tableau suivant, parce qu'elle ne sait pas quelle est la place disponible. C'est pour ça que cette fonction ne doit pas être utilisée dans un programme C.

2.4. Exercice : longueur d'une chaîne

Vous connaissez la fonction `strlen` du langage C. Vous allez la programmer en assembleur.

☛ Complétez ce programme, les explications sont après :

```
;; programme principal
main:       ; DSKY en mode 10 (affichage de nombres)
            LD  R0, 10
            OUT R0, 255
            ; obtenir la longueur du message dans R0
            LD  R0, message
            CALL strlen
            ; afficher cette longueur sur le DSKY
            OUT R0, 10
            ; fin du programme
            HLT

message:    DB "Bonjour, comment ca va ?", 0


;; fonction strlen(R0)
;; calcule la longueur de la chaîne située en R0 et la retourne dans R0
strlen:     ; TODO sauver les registres utilisés
            ; TODO parcourir la chaîne en comptant les caractères, jusqu'à un 0
            ; TODO restaurer les registres
            RET
```

Le cœur de l'algorithme est une boucle partant du début de la chaîne, dans `R0`, fourni en paramètre. À chaque tour de boucle, on doit regarder si le caractère courant est le code nul. On arrête si c'est le cas. Sinon, on passe au caractère suivant. Et à chaque tour, on incrémente un compteur qu'on retourne dans `R0` en résultat à la fin.

Il semble très complexe d'employer le concept des indices pour accéder aux caractères de la chaîne. `R1` serait initialisé à 0 et avancerait dans la chaîne, et on consulterait le caractère `[R1+R0]`. Le problème est qu'on ne peut pas écrire cela. On doit faire l'addition à part puis utiliser `[R0]` ou `[R1]`. Pas évident.

Le problème est qu'on n'a que deux registres, or il semble qu'il en faudrait davantage.

Une astuce consiste à utiliser `R0` pour avancer dans la chaîne, et initialiser `R1` à la valeur de départ de `R0`. `R1` reste sur le début de la chaîne. Donc seul `R0` avance dans la chaîne, tant que `[R0]` n'est pas nul. Et le résultat est tout simplement `R0` − `R1`. On peut en effet soustraire deux adresses et ça donne une sorte de distance entre la fin de la chaîne et son début, donc la longueur de la chaîne.

Le problème alors est de regarder si le caractère désigné par `[R0]` est nul. On ne peut pas écrire

directement `CMP [R0],0`, donc là aussi, il faut une astuce. Cela consiste à empiler la valeur de `R0` au début de la fonction, à utiliser librement `R1` pour lire le caractère courant et le comparer à 0, puis à la fin de la fonction, dépiler `R1` au lieu de `R0`, donc `R1` contient l'adresse de début de la chaîne, avant de faire la soustraction entre `R0` et `R1`.

2.5. Exercice : dessiner une chenille

On va finir par un exercice moins sérieux que les précédents. Il faut dessiner ceci :

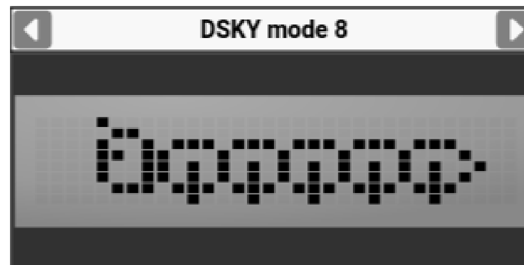

Figure 1: Chenille

Voici un programme à compléter. Les explications sont après.

```
;; programme principal
main:       ; mettre le DSKY en mode 8 = LCD matriciel
            LD  R0, 8
            OUT R0, 255
            ; dessiner la tête
            LD  R0, data_tete
            CALL dessiner
            ; TODO dessiner 5 segments pour le corps
            ; dessiner la queue
            LD  R0, data_queue
            CALL dessiner
            ;fin du programme
            HLT

data_tete:  DB %00111101
            DB %01001010
            DB %01000010
            DB %01111100

data_corps: DB %00111000
            DB %01000100
            DB %11110100
            DB %01000100

data_queue: DB %00101000
            DB %00010000
            DB %00000000
            DB %00000000
```

```
;; recopie 4 octets à partir de R0 sur le port 8
dessiner:   ; TODO votre fonction doit envoyer les 4 octets situés à
            ; l'adresse R0 sur le port 8, mais sans modifier R0 au retour
            RET
```

Le programme principal consiste en appels de la même fonction, mais avec un paramètre `R0` différent. Ce paramètre est l'adresse de 4 octets donnant les pixels à allumer. La fonction `dessiner` doit envoyer ces 4 octets sur le port 8, peu importe comment. C'est tout.

Souvent le compilateur C transforme de petites boucles dont le nombre d'itérations est connu à l'avance en simples répétitions d'instructions.