

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.

👉 Affichez la mise en page `layout3`.

## 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.

Comme une même fonction peut être appelée de plusieurs endroits du programme, le processeur possède un mécanisme pour se rappeler où il faut 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 en cascade. 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 reprendre 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 l'adresse du sommet de la pile. Il est initialisé à l'adresse 255, à la fin de la mémoire. La pile se remplit en descendant vers le bas de la mémoire. Empiler une adresse de retour lors d'un appel de fonction fait décrémenter le registre **SP**, et dépiler une adresse au retour de la fonction incrémente **SP**.

La figure 1, page 2 montre l'organisation de la mémoire. Au début de la mémoire, en haut, il y a le programme exécutable avec ses données. À la fin de la mémoire, en bas, il y a la pile qui se remplit vers les adresses décroissantes.

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.

NB: en fait, **SP** est devant le premier emplacement mémoire libre au dessus de la pile. C'est à cause de l'ordre des opérations lors d'un empilement et d'un dépilement : **CALL** met IP dans [**SP**]

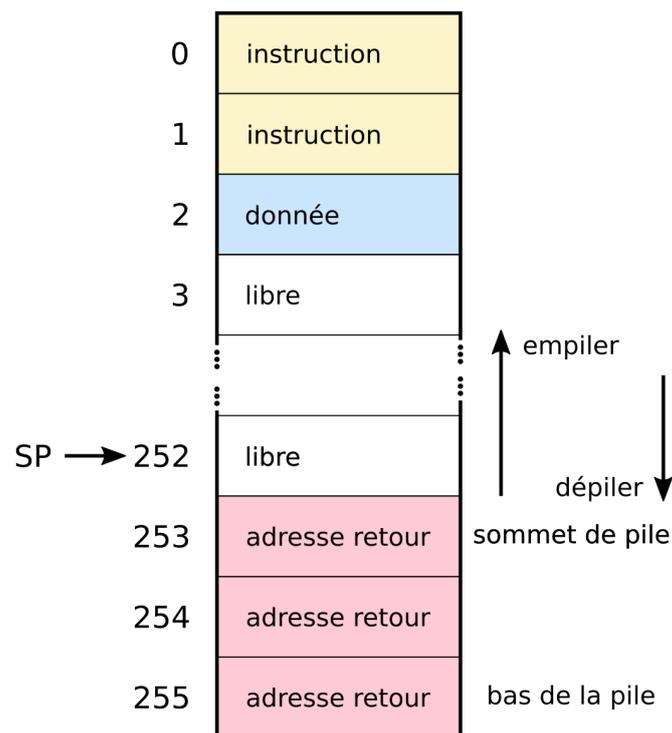


Figure 1: Organisation de la mémoire

puis décrémente SP tandis que RET incrémente SP puis affecte IP avec [SP].

## 1.1. Exemples de fonctions

🔧 Assemblez ce programme :



```
;; programme principal
main:      ; DSKY en mode 11 (affichage de caractères ascii)
           LD R0, 11
           OUT R0, 255
           ; affichage d'un B
           LD R0, 'B'
           OUT R0, 11
           ; appel de la fonction 1 dans le programme principal
           CALL fonction1
           ; affichage d'un n
           LD R0, 'n'
           OUT R0, 11
           ; appel de la fonction 2 dans le programme principal
           CALL fonction2
           ; affichage d'un r
           LD R0, 'r'
           OUT R0, 11
           ; fin du programme
           HLT
```

```
;; affichage d'un 'o'
fonction1: LD R0, 'o'
           OUT R0, 11
           RET

;; affichage de 'jou'
fonction2: ; affichage d'un j
           LD R0, 'j'
           OUT R0, 11
           ; appel de la fonction 1 dans la fonction 2, elle affiche un o
           CALL fonction1
           ; affichage d'un u
           LD R0, 'u'
           OUT R0, 11
           RET
```

Il y a toujours un label pour nommer une fonction. Les fonctions peuvent être dans un ordre quelconque dans la mémoire, mais pas en premier, parce que l'exécution commence toujours à l'adresse 0, et il faut que ça soit le programme principal. Ou alors, il faudrait rajouter un `BRA main` au tout début... c'est ce que fait le compilateur C.

Donc un programme en assembleur commence toujours par la partie principale (`main` en C), et ensuite on trouve les fonctions, chacune identifiée par un label.

☛ Cliquez le bouton `Rst+Run` ou `Reset` puis `RUN`. Ça affiche un message.

☛ Cliquez sur le bouton `Reset` puis appuyez 4 fois sur `Next`. On est arrivé à l'adresse `IP=08`, celle du `CALL fonction1`. Regardez la pile : `SP` vaut `FF`, c'est à dire 255. Appuyez sur `Next`. `IP` vaut maintenant 15 (en hexa) au début de la `fonction1`. `SP` vaut `FE`, c'àd 254 en décimal et dans la mémoire, la cellule d'adresse `FE` contient `0A` et désigne l'instruction qui suit `CALL` qu'on vient juste d'exécuter.

☛ Cliquez 2 fois sur le bouton `Next`. Ça affiche un `o`. On est maintenant devant l'instruction `RET`. Regardez `IP` et `SP`. Appuyez sur `Next`. Regardez les nouvelles valeurs de `IP` et `SP` : `IP` vaut `0A`, c'est ce qu'il y avait sur la pile. On est maintenant devant l'instruction qui suit le `CALL fonction1` et la pile est revenue à l'état initial. NB: le contexte n'affiche pas les cellules mémoire en dessous de `SP`, seulement celles comprises entre `SP` et 255.

☛ Continuez à appuyer sur `Next` en regardant ce qui arrive à `IP` et `SP` à chaque `CALL` et `RET`.

Quand il y a `CALL`, `SP` est décrémenté et la cellule mémoire d'adresse `SP` contient l'adresse de l'instruction qui suit le `CALL`. Quand il y a `RET`, `IP` reprend la valeur qu'il y a sur la pile, de manière à revenir juste après le `CALL`, et `SP` est incrémenté. C'est ce mécanisme qui permet d'appeler des fonctions en cascade et de toujours revenir au bon endroit.

☛ Modifiez le programme précédent, ligne 31, `CALL fonction1`, faites une toute petite erreur : mettez `CALL fonction2`. C'est à dire que `fonction2` s'appelle elle-même. Vous savez peut-être ce que ça provoque quand on fait ça en C. Lancez l'exécution et attendez la fin.

Le programme va planter parce que la pile, en se remplissant vers le bas, vient écraser les instructions du programme. Dans un ordinateur moderne, c'est détecté et l'exécution s'arrête

Dans un véritable ordinateur, il y a plusieurs piles, une pour chaque processus (voir le cours de Systèmes). Ces piles sont dans des zones mémoires séparées des données et des instructions. Les piles ont une taille supposée assez grandes pour les besoins du processus. Le système d'exploitation surveille le remplissage des piles et tue les processus dont la pile a débordé. Il affiche un message d'erreur « incident de segmentation » ou, en Java, c'est l'erreur « maximum recursion depth exceeded (StackOverflowError) ».

## 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 et essayez ce programme :



```
;; programme principal
main:      ; DSKY en mode 11 (affichage de caractères ascii)
          LD  R0, 11
          OUT R0, 255
          ; appels de fonctions
          LD  R0, '0'
          CALL putchar
          CALL changeCase
          CALL putchar
          LD  R0, 'K'
          CALL changeCase
          CALL putchar
          ; fin du programme
          HLT

;; affichage de R0 (code ascii) sur le port 11
;; paramètres : R0 = code ascii du caractère à écrire
;; résultat : aucun
putchar:   OUT R0, 11
          RET

;; transforme une minuscule dans R0 en majuscule et inversement
;; paramètres : R0 = code ascii du caractère à transformer
;; résultat : R0 = code ascii du caractère transformé
changeCase: XOR R0, %00100000
          RET
```

👉 Analysez ce programme. Il y a deux fonctions :

- `putchar` affiche le caractère représenté par son code ascii. Cette fonction existe vraiment en langage C [cours](#) [↗](#). Ici, le code à afficher doit être mis dans R0.
- `changeCase` n'existe pas en langage C. Cette fonction ici transforme une lettre minuscule en majuscule et inversement.

Le calcul est un peu mystérieux et on ne vous demande pas de le comprendre. Tout part

du codage ASCII, voir la [table](#) ↗. Regardez les colonnes Hex et Char pour les lettres. Par exemple, un `s` est codé `73 = 01110011` en base 2 et un `S` codé `53 = 01010011`. Il suffit d'inverser le bit `b5`. C'est ce que fait l'instruction `XOR`, elle inverse les bits du registre qui sont à 1 dans son second paramètre.

### 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      ; i = R0 = nombre de boucles
          ; répéter {
.repeter:
          ; i--
          DEC R0
          ; afflettre(i)
          CALL afflettre
          ; } jusqu'à (i == 0)
.jusqua:
          CMP R0, 0
          BNE .repeter
          ; fin du programme
          HLT

;; affichage de R0 (entier 0..25) sur le port 11
;; paramètres : R0 = numéro 0..25 du caractère à afficher
;; résultat : aucun
afflettre: 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`. Ça permet de reprendre les mêmes noms de labels dans des fonctions différentes, sans les mélanger.

Dans la fonction `main`, il y a une boucle pour `R0=26` à `R0=1` (arrêt quand `R0==0`). Notez qu'on décrémente `R0` avant d'appeler la fonction `afflettre`, donc cette fonction est appelée avec `R0=25` à `R0=0`.

Dans la fonction `afflettre`, 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. Ainsi, `afflettre` affiche le caractère `A..Z` dont le numéro `0..25` est passé dans `R0`.

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.

👉 Mettez le `PUSH R0` de la ligne 23 en commentaire (placez un `;` au début de la ligne) et voyez ce qui se passe. C'est très rapide. Le `POP` n'est pas en face d'une valeur, mais d'une adresse de retour. CimPU empêche de la dépiler par un `POP`.

Remettez en état avant de continuer.

👉 Mettez le `POP R0` de la ligne 26 en commentaire et voyez ce qui se passe. CimPU interdit de faire un retour de fonction à une adresse empilée qui ne fait pas partie des instructions du programme. Ici, c'est une valeur de registre quelconque qui a été empilée au dessus de l'adresse de retour. Si on ne fait pas le `POP` correspondant, alors le `RET` dépilerait cette valeur qui est une mauvaise adresse. CimPU empêche ça. Par contre, c'est une technique qui est utilisée pour du hacking, de placer des instructions dans une zone mémoire, d'empiler son adresse par un `PUSH` et de faire un `RET` ensuite. L'exécution continue alors dans la zone mémoire...

👉 Mettez le `PUSH R0` de la ligne 23 et le `POP R0` de la ligne 26 tous les deux en commentaire et voyez ce qui se passe. `R0` revient modifié n'importe comment par la fonction et ça empêche la boucle de fonctionner normalement.

Deux constatations :

- Toute fonction doit sauver tous les registres qu'elle utilise. Une fonction doit faire son travail de manière invisible pour le programme qui l'appelle.
- 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.

## 2. Exercices avec des fonctions

### 2.1. Recopie d'un texte

👉 Essayez ce programme :



```
;; programme principal
main:      ; DSKY en mode 11 (lecture et affichage de caractères ascii)
           LD R0, 11
           OUT R0, 255
           ; TODO repeter {

           ; lire un caractère dans R0
           CALL getchar
           ; mettre le caractère en majuscules
           CALL toUpper
           ; afficher le caractère R0
           CALL putchar

           ; TODO } jusqu'à (c == '\n')

           ; fin du programme
           HLT

;; fonction getchar() lit un caractère sur le terminal ascii
;; paramètres : aucun
;; résultat : R0 = code ascii du caractère lu
getchar:   IN R0, 11
           RET

;; affichage de R0 (code ascii) sur le port 11
;; paramètres : R0 = code ascii du caractère à afficher
;; résultat : aucun
putchar:   OUT R0, 11
           RET

;; mise en majuscules de R0 (code ASCII)
;; paramètres : R0 = code ascii du caractère à transformer
;; résultat : R0 = code ascii du caractère transformé
toUpper:   ; TODO si (R0 >= 'a' et R0 <= 'z') {

           ; mettre le bit 5 à zéro
           AND R0, %11011111

           ; TODO }

           ; return R0
           RET
```

☛ Votre travail consiste à programmer la boucle *répéter* de la partie principale et la conditionnelle qui correspond aux commentaires dans la fonction `toUpper`. Cette conditionnelle a une condition double. Pour la coder, ce sont simplement deux comparaisons et deux branchements au « finis »

quand les conditions sont fausses, comme s'il y avait deux *si* à la suite avec le même *finsi* : 

```
.si:      CMP R0, valeur1
          B?? .finsi
          CMP R0, valeur2
          B?? .finsi
.alors:
    ...
.finsi:
```

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

## 2.2. Fibonacci, V1

Voici maintenant un calcul mathématique classique, la fonction de Fibonacci, voir [sa définition](#) . Le but est d'avoir un programme qui calcule  $F(n)$ . Il doit par exemple afficher 21 pour  $F(8)$ . 

```
;; programme principal
main:      ; DSKY en mode 10 (lecture et affichage de nombres)
          LD  R0, 10
          OUT R0, 255
          ; lire un entier n dans R0
          IN  R0, 10
          ; calculer F(n), résultat dans R0
          CALL F
          ; afficher le résultat
          OUT R0, 10
          HLT

;; fonction de Fibonacci
;; paramètres : R0 = n
;; résultat : R0 = F(n)
F:         ; TODO si (R0 == 0 ou R0 == 1) {
          ; TODO      ne rien faire car R0 = résultat
          ; TODO } sinon {
          ; TODO      calculer F(R0-1) + F(R0-2) dans R0
          ; TODO }

          ; retour de la fonction, R0 contient le résultat
          RET
```

Comme vous le constatez, la fonction  $F(R0)$  n'est pas programmée. Elle se contente de retourner la valeur qu'on lui passe dans  $R0$ . L'algorithme à suivre est écrit dans les commentaires mais il est compliqué, avec deux appels récursifs. Vous allez le programmer par étapes.

On va commencer par programmer la conditionnelle, `si (R0 == 0 ou R0 == 1) {`. Elle suit le schéma *si alors sinon finsi* vu dans le TP2. La différence, c'est la condition en deux comparaisons mais contrairement à l'exercice précédent, c'est un *ou* qui les relie.

Alors il vaut mieux inverser cette condition double et le *alors* avec le *sinon*, car il est bien plus facile de programmer une conjonction qu'une disjonction. On s'aperçoit aussi que le *alors* devenu

un *sinon* est inutile, car la valeur à retourner est toujours dans R0. Voici la transformation : 

```
;; fonction de Fibonacci
;; paramètres : R0 = n
;; résultat : R0 = F(n)
F:      ; TODO si (R0 != 0 et R0 != 1) {
        ; TODO   calculer F(R0-1) + F(R0-2) dans R0
        ; TODO }

        ; retour de la fonction, le résultat est dans R0
RET
```

Donc finalement, il reste à programmer la partie *alors* contenant les deux appels récursifs. Voici quelques éléments de réflexion :

- Pour calculer  $F(R0-1)$ , il suffit de décrémenter R0 puis de faire CALL F. Le résultat sera dans R0.
- Le problème, c'est qu'on n'a plus l'ancienne valeur de R0 quand on veut calculer  $F(R0-2)$ . La solution consiste à empiler R0 avant d'appeler la fonction et de le dépiler une fois qu'on a utilisé le résultat de la fonction.
- Le problème alors, c'est qu'on a besoin du résultat de la fonction et du nombre qu'on lui passe. La solution consiste à utiliser R1 pour calculer l'addition de  $F(R0-1)$  avec  $F(R0-2)$ .
- Le problème, si on utilise R1, c'est qu'il va être modifié pendant l'appel récursif. La solution consiste à l'empiler avant de commencer les calculs et le dépiler tout à la fin, avant le retour de la fonction.

👉 Voici donc un scénario possible, à programmer et tester :

1. empiler R1
2. décrémenter R0
3. empiler R0
4. appeler la fonction
5. recopier R0 dans R1
6. dépiler R0
7. décrémenter R0
8. appeler la fonction
9. additionner R0 et R1 dans R0
10. dépiler R1

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

Quelle est la valeur maximale avec laquelle on peut appeler la fonction  $F$  pour obtenir un résultat correct ? Vous pouvez essayer d'appeler la fonction avec des entiers au delà de cette limite, mais d'une part le temps de calcul va être gigantesque, et d'autre part le résultat sera faux.

Cette manière de faire, avec deux appels récursifs par appel de la fonction, n'est pas du tout recommandée car son temps de calcul ou le nombre d'instructions exécutées, ce qu'on appelle la complexité, est exponentiel. Nous verrons une autre méthode à la fin du TP qui permet de réduire très nettement la complexité.

### 3. 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  $n^{\circ}i$  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

; char T3[n] = {valinit, valinit, valinit, valinit...};
T3:      RB n, valinit ; avec n = taille, valinit = valeur à mettre partout
```

On écrit `LD R1, [R0+tab]` pour accéder à la case d'indice  $R0$  (registre) du tableau `tab`. Ça marche aussi avec  $R1$  en tant qu'indice : `LD R0, [R1+tab]`.

NB: on ne peut pas écrire `LD R1, [tab+R0]`, le registre doit toujours être avant la constante `tab`.

👉 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, [R0+message] ; premier caractère dans R1
          ; tantque (R1 != 0) {
.tantque:
          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 "Salut, tout va bien ?", 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];`.

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.

### 3.1. 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 (R1 != 0)

.tantque:
          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 "Salut, tout va bien ?", 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.

### 3.2. 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 "Salut, tout va bien ?", 0

;; fonction puts(R0)
;; affiche la chaîne située en R0 sur le port 11
;; paramètres : R0 = adresse de la chaîne terminée par un 0
;; résultat : aucun
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`.

### 3.3. Exercice : dessiner une chenille

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

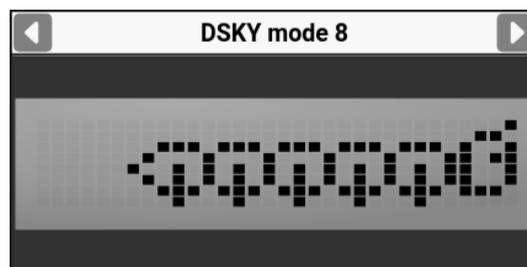


Figure 2: Chenille

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 queue
           LD R0, data_queue
           CALL dessiner
           ; TODO dessiner 5 segments pour le corps
           ; dessiner la tete
           LD R0, data_tete
           CALL dessiner
           ; fin du programme
           HLT

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

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

data_queue: DB %00000000
           DB %00000000
           DB %00110000
           DB %01001000

;; recopie 4 octets à partir de R0 sur le port 8
;; paramètres : R0 = adresse de 4 octets à la suite
;; résultat : aucun
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.

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

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

### 3.4. Fibonacci, V2

On revient sur le calcul de la fonction de Fibonacci.

☛ Reprenez votre réponse à l'exercice 2.2.

On va utiliser un tableau pour accélérer très nettement les calculs. En effet, la fonction  $F(n)$  demande les valeurs de  $F(n-1)$  et  $F(n-2)$ . Et pour calculer  $F(n-1)$ , il y a besoin de  $F(n-2)$  et  $F(n-3)$  et ainsi de suite. Alors le principe est de calculer  $F(i)$  seulement une fois et l'enregistrer dans un tableau, à l'indice  $i$ . Dans ce tableau appelé `F_mem`, on a initialisé  $F(0) = 0$  et  $F(1) = 1$  et toutes les autres cases  $i > 1$  sont à une valeur sentinelle, par exemple 255 pour indiquer que  $F(i)$  est inconnue.

Voici l'algorithme simplifié :

```
fonction F(n) {
  si (F_mem[n] == 255) {
    F_mem[n] = F(n-1) + F(n-2)
  }
  retourner F_mem[n]
}

tableau F_mem[14] = { 0, 1, 255, 255, 255... }
```

Cet algorithme est correct si on admet qu'on n'appellera jamais la fonction avec une valeur autre que 0..13. L'algorithme semble plus simple parce qu'il n'y a pas les tests sur  $n$ , s'il vaut 0 ou 1. Ces tests sont inutiles parce que  $F(0)$  et  $F(1)$  sont déjà dans le tableau.

Il reste à programmer cet algorithme. Ce serait très simple en langage C, mais ça va être nettement plus difficile en assembleur. Le problème est qu'il faut gérer plusieurs valeurs :  $n$ , `F_mem[n]` ainsi que les valeurs retournées par les appels récursifs à  $F$ . On n'a pas assez avec R0 et R1. Il faut utiliser la pile.

☛ Voici donc un scénario possible, à programmer et tester :

1. empiler R1
2. mettre `F_mem[R0]` dans R1
3. conditionnelle :
  - si (R1 == 255)
  - alors :
    - a. empiler R0 ; sauve  $n$  sur la pile
    - b. décrémenter R0 ; R0 =  $n - 1$
    - c. empiler R0 ; sauve  $n - 1$  sur la pile
    - d. appeler la fonction F ; au retour, R0 =  $F(n - 1)$
    - e. recopier R0 dans R1 ; R1 =  $F(n - 1)$
    - f. dépiler R0 ; récupérer  $n - 1$  dans R0
    - g. décrémenter R0 ; R0 =  $n - 2$
    - h. appeler la fonction F ; au retour, R0 =  $F(n - 2)$
    - i. additionner R0 et R1 dans R1 ; R1 =  $F(n - 1) + F(n - 2)$
    - j. dépiler R0 ; récupérer  $n$  dans R0,  $F(n)$  est dans R1
    - k. stocker R1 dans `F_mem[R0]`
  - finsi
4. recopier R1 dans R0 ; R0 =  $F(n)$
5. dépiler R1
6. revenir de la fonction

Pour calculer  $F(13)$ , on passe d'environ 7500 instructions à un peu plus de 300 seulement.

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

Pensez à déposer votre `ReponsesTP3.txt` dans la zone de dépôt sur Moodle. Rappel : c'est un travail personnel, individuel qui est soumis à une notation. Les tricheries seront sanctionnées.