

Ce TD explique comment programmer une itération, c'est à dire un traitement répété plusieurs fois. Il y a traditionnellement plusieurs types d'itérations, qu'on appelle des « *boucles* ».

- boucle *répéter jusqu'à*
- boucle *tant que*
- boucle *pour*

Une seule est vraiment à connaître, la boucle *tant que* car toutes les autres peuvent être construites avec. Par exemple, la boucle *pour* du langage C :

```
for (initialisation; condition; progression) {  
    instructions;  
}
```

se transforme en :

```
initialisation;  
while (condition) {  
    instructions;  
    progression;  
}
```

On va commencer par la plus simple, la boucle *répéter jusqu'à*.

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


👉 Agrandissez la fenêtre (ou zoomez) pour travailler confortablement, et configurez CimPU ainsi :

- Mettez en niveau 2.
- Affichez la mémoire centrale en mode *Contexte d'exécution*.
- Mettez le DSKY en mode 10.
- Désactivez les animations.

1. Boucle « répéter jusqu'à »

1.1. Nombre mystère

On commence par un petit jeu. Un nombre est caché et vous devez le deviner.

👉 Assemblez ce programme. 

```
LD R0, $54 ; nombre secret, ouh mince, fallait pas le dire !  
repeater: IN R1, 10 ; l'utilisateur doit saisir un nombre qui va dans R1  
jusqua:   CMP R1, R0 ; est-ce que R1 == R0 ?  
          BNE repeater ; saut si pas égaux  
  
OUT R0, 10 ; afficher le nombre secret  
HLT
```

👉 Lancez le programme. Il va vous demander de taper un nombre et ne s'arrêtera que si c'est 54 en hexadécimal. Si vous voulez tricher, vous pouvez taper \$54 dans le terminal.

Le traitement est répété tant que la condition $R0 == R1$ n'est pas vraie. Il y a donc un branchement au début du traitement lorsque la condition est fausse. L'instruction **BNE** saute à l'adresse fournie si Z vaut 0 ; ça arrive quand la comparaison montre une différence entre les valeurs ($R0 - R1 \neq 0$).

Relisez le TD7 concernant la programmation des conditionnelles. La différence ici, c'est qu'on saute au début de la boucle au lieu de sauter au *finis*.

Notez que la condition est vérifiée après avoir fait le traitement. Donc le traitement a lieu au moins une fois.

répéter {instructions...} jusqu'à (condition) se programme ainsi :

```

...                ; initialisations...

repeter:           ; instructions...

jusqua:           CMP R0, valeur ; évaluation de la condition
                  Bxx repeter   ; saut si la condition est FAUSSE

...                ; suite du programme
    
```

La boucle s'arrête quand la condition est vraie, c'est à dire qu'il n'y a pas le saut au *répéter*.

1.2. Nombre mystère, variante

Comme le jeu est trop difficile, on propose de le simplifier.

- ☛ Remplacez l'instruction **BNE** par **BLO** et ré-assemblez le programme.
- ☛ Lancez le programme. Cette fois le programme s'arrêtera quand le nombre que vous avez tapé satisfera une certaine condition. À vous de trouver laquelle.

L'instruction **BLO** s'adresse aux comparaisons entre entiers non signés, de 0 à 255. Voici le tableau des branchements pour les entiers non signés :

op	Saut si	Branchement	Signification
<	≥	BHS	Branch if higher or same
≤	>	BHI	Branch if higher
=	≠	BNE	Branch if not equal
≠	=	BEQ	Branch if equal
≥	<	BLO	Branch if lower
>	≤	BLS	Branch if lower or same

Relire le [TD7](#) pour les « branchements signés ».

- ☛ Vous devez comprendre pourquoi l'instruction **BLO** fait recommencer le traitement (demander un nombre et le comparer au nombre secret) sous la condition qui est exprimée dans le tableau ci-dessus et comment on écrirait l'algorithme en pseudo-code.

1.3. Nombre caché, version 3

☛ Codez l'algorithme suivant :

```
R0 = $54
repete {
  lire un nombre dans R1
  si (R0 < R1) {
    afficher -1 en utilisant R1
  }
  si (R0 > R1) {
    afficher 1 en utilisant R1
  }
} jusqu'à (R0 == R1)
afficher R0
```

Ce programme continue à demander un nombre tant qu'il n'est pas égal au nombre caché, mais il vous aide en vous disant si le nombre caché est plus petit ou plus grand que votre proposition. Remarquez que :

1. Les deux conditionnelles à l'intérieur du *répéter* sont indépendantes, mais on pourrait mettre un *sinon* entre elles.
2. La valeur proposée par l'utilisateur dans R1 est perdue lors de l'affichage du +1 ou -1, parce qu'on remplace R1 par 1 ou -1. Par contre, dans le cas où la réponse est bonne, on n'y touche pas. Donc la comparaison finale reste correcte, sauf si malheureusement le nombre caché est 1 ou -1.

Donc, l'algorithme peut être codé en n'utilisant que R0 et R1. Sinon, il aurait fallu employer une cellule mémoire pour, par exemple stocker le nombre caché.

Rappel du TD7 : quand on veut tester la condition ($A \text{ op } B$), on écrit `CMP A,B` et on choisit le branchement qui correspond à l'inverse de *op*. Ici, on veut tester $R0 < R1$, donc on écrit `CMP R0,R1` et on choisit le branchement *branch if higher or same*, BHS.

1.4. Nombre caché, version 4

On va améliorer la génération du nombre caché. À cause du fait qu'on n'utilise pas de cellules mémoire, que tout est mis dans les deux seuls registres, il faut que le nombre caché ne soit ni 1, ni 255 (-1).

En plus, on veut qu'il soit tiré aléatoirement. Dans CimPU, le port 254 est spécial : il retourne un octet aléatoire à chaque fois qu'on le lit, ex : `IN R0, 254`.

☛ Codez l'algorithme suivant au début de la version précédente :

```
repete {
  R0 = lire le port 254 /* octet aléatoire */
} jusqu'à ((R0 > 1) et (R0 < 255))
```

C'est plus compliqué. La condition est double, c'est une conjonction. Réfléchissez à ce qu'on doit

faire : retourner au *répéter* si l'une des deux conditions est fausse. Donc, c'est simplement deux couples `CMP + Bxx` l'un après l'autre.

Attention à renommer les labels `repeter` et `jusqua`.

2. Boucle « tant que »

Ces boucles sont plus générales que les précédentes. Elle vérifient la condition avant tout traitement. Donc il se peut que le traitement ne soit pas effectué si la condition est initialement fausse.

- On choisit une boucle *répéter jusqu'à* quand la condition est calculée par le traitement, donc quand on sait s'il faut continuer ou pas seulement après avoir fait le traitement.
- On choisit une boucle *tant que* quand le traitement ne doit être fait que si la condition est vraie auparavant.

Dans les deux cas, le traitement doit aussi faire en sorte de modifier la condition, sinon ça crée une boucle infinie.

`tantque (condition) faire {instructions...} fintantque` se programme ainsi :

```
... ; initialisations...  
  
tantque:  CMP R0, valeur ; évaluation de la condition  
          Bxx fintantque ; saut si la condition est FAUSSE  
  
faire:    ; instructions...  
  
          BRA tantque ; revenir au test de continuation  
fintantque:  
  
... ; suite du programme
```

La boucle s'arrête quand la condition est vraie, c'est à dire qu'il n'y a pas le saut au *répéter*.

👉 Assemblez ce programme :



```
LD R0, 89  
LD R1, 23  
  
tantque:  CMP R0, R1 ; tant que R0 > R1  
          BLS fintantque ; saut si R0 <= R1  
  
faire:    DEC R0 ; R0 = R0 - 1  
          INC R1 ; R1 = R1 + 1  
  
          BRA tantque ; revenir au test de continuation  
fintantque:  
          OUT R0, 10  
          OUT R1, 10  
          HLT
```

☛ Lancez le programme pour tester différentes valeurs. Que fait ce programme ? On avait vu une solution plus efficace dans le TD6.

2.1. Exercice

☛ Codez l'algorithme suivant :

```
R0 = lire le port 10 (lire un entier)
R1 = 0
tantque (R0 >= 7) {
    R0 = R0 - 7
    R1 = R1 + 1
}
afficher R1
afficher R0
```

☛ Lancez l'exécution avec différentes valeurs. Essayez des multiples de 7, et leurs successeurs comme 29, 36, 71. Que fait ce programme ?

2.2. Table de multiplication d'un nombre

On veut écrire un programme qui demande un nombre entier N et qui affiche sa table de multiplication de $0.N$ à $10.N$.

☛ Terminez ce programme (mettez les instructions sous les commentaires TODO) :



```
IN R0, 10 ; saisir un nombre dans R0
ST R0, [N] ; mettre ce nombre dans la mémoire

; TODO mettre R0 à zéro
; TODO mettre R1 à zéro

tantque: ; TODO condition : R1 <= 10
; TODO saut au fintantque si la condition est fausse

faire: ; afficher R0 sur le port 10
ADD R0, [N] ; R0 = R0 + n
; incrémenter R1
BRA tantque

fintantque: ; fin du programme
HLT

N: DB 0 ; variable N
```

Dans ce programme, R1 joue le rôle de i dans `for (int i=0; i<=10; i++)`.