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 elle permet de construire toutes les autres. 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*, puis étudier les autres.

👉 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 !

repeter:    IN  R1, 10      ; l'utilisateur doit saisir un nombre qui va dans R1

jusqua:     CMP R1, R0      ; est-ce que R1 == R0 ?
            BNE repeter     ; 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 corps de la boucle est répété tant que la condition `R0 == R1` n'est pas vraie. Il y a donc un retour au début 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` ≠ 0).

Relisez le TD7 concernant la programmation des conditionnelles, en particulier le tableau des conditions dans le §4.2, ce que signifie `BNE` « *branch if not equal* », c'est à dire faire un saut au label `repeter` si les deux valeurs ne sont pas égales. La différence entre une conditionnelle et cette boucle, c'est qu'on saute au début de la boucle au lieu de sauter au *finsi*.

Notez que la condition est testée après avoir fait les instructions de la boucle. Donc les instructions sont exécutées au moins une fois.

répeter {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, il n'y a alors pas de 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
$<$	$\geq$	BHS	Branch if higher or same
$\leq$	$>$	BHI	Branch if higher
$=$	$\neq$	BNE	Branch if not equal
$\neq$	$=$	BEQ	Branch if equal
$\geq$	$<$	BLO	Branch if lower
$>$	$\leq$	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

👉 Écrivez l'algorithme en pseudo-code.

1.3. Nombre caché, version 3

👉 Codez et testez l'algorithme suivant :

```
R0 = $54
repeter {
    lire un nombre dans R1
    si (R0 < R1) {          // comparaison non signée
        mettre -1 dans R1
    } sinon {
        si (R0 > R1) {      // comparaison non signée
            mettre 1 dans R1
        }
    }
    afficher 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 (-1 codé par 255) ou plus grand ($+1$) que votre proposition.

Remarquez que 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.

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

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. Mode d'emploi : `IN reg,254` met un nombre aléatoire dans le registre *reg*, `R0` ou `R1`.

Dans les ordinateurs actuels, il y a toujours un circuit électronique qui fabrique de tels nombres aléatoires, voir ce lien . C'est utilisé pour crypter des données.

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

```
repeter {
    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 au moins 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`. Vous aurez une première boucle avec les labels `repeter1` et `jusqua1` et une seconde avec `repeter2` et `jusqua2`.

2. Boucle « tant que »

Ces boucles sont plus générales que les précédentes. Elle vérifient la condition avant l'exécution du corps. Donc il se peut que le corps ne soit pas effectué si la condition est initialement fausse.

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

Dans les deux cas, le corps doit 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 fausse, c'est à dire qu'il y a un saut au *fintantque*. On choisit l'instruction de saut exactement comme pour les conditionnelles du TD7.

☛ Assemblez ce programme :

```
            ; lire deux nombres sur le terminal, port 10
            IN  R0, 10
            IN  R1, 10

            ; tant que ???
tantque:    CMP R0, R1      ; tant que R0 > R1
            BLS fintantque  ; saut si R0 <= R1

            ; corps de la boucle
faire:      DEC R0          ; R0 = R0 - 1
            INC R1          ; R1 = R1 + 1

            ; fin du corps, retour au test de continuation
```

```
            BRA tantque
fintantque:
            ; afficher les deux nombres résultants
            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    (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 $N \times 0$ à $N \times 10$.

👉 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 tanque (R1 <= 10) {
            ; TODO saut au fintantque si la condition est fausse

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

            ; revenir au test de continuation
            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++)`.