TABLEAU

```
// Première version : Copier Tab1 dans Tab2
// A chaque tour de boucle, il y a l'élément courant dans %r4
// N = 10        // N constante, pour éviter d'utiliser des valeurs en dur dans le programme
//        set Tab1, %r1
//        set Tab2, %r2
//        clr %r3
// Tantque:      cmp %r3, N
//        bgeu Stop              // branch if r3 greater or equal unsigned to N
//        ld [%r1+%r3], %r4      // ld = load - %r2+%r3]= adresse de tab(r3)
//        st %r4, [%r2+%r3]
//        add %r3, 1, %r3        // index <- index + 1
//        ba Tantque
// Stop :        ba Stop
// Tab1 :        .word 10, 9, 8, 7, 1, 6, 5, 4, 3, 2, 1
// Tab2 :        .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

// Deuxième version : Copier le miroir de Tab1 dans Tab2
N = 11          // N constante, pour éviter d'utiliser des valeurs en dur dans le programme
        set Tab1, %r1
        set Tab2, %r2
        clr %r3
        set N-1, %r5
Tantque:        cmp %r3, N
        bgeu Stop              // branch if r3 greater or equal unsigned to N
        ld [%r1+%r3], %r4      // ld = load - %r2+%r3]= adresse de tab(r3)
        sub %r5, %r3, %r6      // i2 = N-1 - i1
        st %r4, [%r2+%r6]
        add %r3, 1, %r3        // index <- index + 1
        ba Tantque
Stop :  ba Stop
Tab1 :          .word 10, 9, 8, 7, 1, 6, 5, 4, 3, 2, 1
Tab2 :          .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

// Le module a été testé et fonctionne correctement
```

AFFICHER_LEDS

```
PILE = 0x200
SWITCHES = 0x90000000
LEDS = 0xB0000000

        set PILE, %sp
        set SWITCHES, %r19
        set LEDS, %r20
boucle:   // ld [%r19], %r1
        // st %r1, [%r20]
```

```
        set 10, %r1
        // call afficher_leds_7_0
        call afficher_leds_15_8
        // ba boucle
Stop:   ba Stop


/* Afficher les 8 bits de faible poids de r1 sur les 8 leds de faible poids et met à 0 les 8 leds de fort poids.
   Les autres sont mises à 0.
   IN : r1 - la valeur à afficher
*/
afficher_leds_7_0:  push %r1
                    push %r2
                    and %r1, 0xFF, %r1
                    set LEDS, %r2
                    st %r1, [%r2]
                    pop %r2
                    pop %r1
                    ret


/* Afficher les 8 bits de faible poids de r1 sur les 8 leds de fort poids et met à 0 les 8 leds de faible poids.
   Les autres sont mises à 0.
   IN : r1 - la valeur à afficher
*/
afficher_leds_15_8: push %r1
                    push %r2
                    sll %r1, 8, %r1
                    set LEDS, %r2
                    st %r1, [%r2]
                    pop %r2
                    pop %r1
                    ret
```

COMMUTATEUR_TACHES_2

```
LEDS = 0xB0000000
PILE0 = 0x200
PILE1 = 0x300
ba init_contexte
handler_IT:     push %r1            // Sauvegarder l'état du programme interrompu
                push %r2
                push %r3
                push %r4
                push %r5
                push %r6
```

```
                push %r7
                push %r8
                push %r9
                push %r10
                push %r28

                st %sp, [%r19+%r17]   // Sauvegarder le pointeur de pile du programme
interrompu
                inc %r17              // Changer le programme courant
                cmp %r17, %r18
                blt prochain_sp       // On remet le numéro du programme courant à 0 s'il
atteint le nombre de programmes considérés (r18)
                clr %r17
prochain_sp:        ld [%r19+%r17], %sp   // Restaurer le pointeur de pile du programme
qui va reprendre

                pop %r28              // Restaurer l'état du programme qui va reprendre
                pop %r10
                pop %r9
                pop %r8
                pop %r7
                pop %r6
                pop %r5
                pop %r4
                pop %r3
                pop %r2
                pop %r1
                reti


/* Afficher les 8 bits de faible poids de r1 sur les 8 leds de faible poids et met à 0 les 8 leds
de fort poids.
  Les autres sont mises à 0.
  IN : r1 - la valeur à afficher
*/
afficher_leds_7_0:  push %r1
                push %r2
                and %r1, 0xFF, %r1
                set LEDS, %r2
                st %r1, [%r2]
                pop %r2
                pop %r1
                ret


/* Afficher les 8 bits de faible poids de r1 sur les 8 leds de fort poids et met à 0 les 8 leds de
faible poids.
  Les autres sont mises à 0.
```

```
   IN : r1 - la valeur à afficher
*/
afficher_leds_15_8: push %r1
                    push %r2
                    sll %r1, 8, %r1
                    set LEDS, %r2
                    st %r1, [%r2]
                    pop %r2
                    pop %r1
                    ret


/* Attendre N secondes.
   IN : r1 - contient N, le nombre de secondes à attendre
*/
wait:               push %r2
                    mov %r1, %r2
                    umulcc %r2, 10, %r2
wait_bcle:          deccc %r2
                    bgt wait_bcle
                    pop %r2
                    ret

/* Incrémente un compteur et l'affiche sur les 8 leds de faible poids.
   Puis réalise une temporisation d'environ 1 seconde.
   IN : r1 - compteur à incrémenter
   OUT : r1 - compteur incrémenté
*/
prog0:              inc %r1
                    call afficher_leds_7_0
                    push %r1
                    call wait
                    pop %r1
                    ba prog0


/* Incrémente un compteur et l'affiche sur les 8 leds de fort poids.
   Puis réalise une temporisation d'environ 1 seconde.
   IN : r1 - compteur à incrémenter
   OUT : r1 - compteur incrémenté
*/
prog1:              inc %r1
                    call afficher_leds_15_8
                    push %r1
                    call wait
                    pop %r1
                    ba prog1
```

```
init_contexte:  set Tab_sp, %r19        // r19 = adresse de Tab_sp
                set 2, %r18             // r18 = nombre de programmes
                clr %r17                // r17 = numéro du programme courant

                // On initialise artifitiellement le contexte des programmes qui n'ont encore
jamais été interrompus
                set PILE1, %sp
                set prog1, %r1
                push %r1
                clr %r1

                // Sauvegarde de l'état actuel
                push %r0
                push %r1
                push %r2
                push %r3
                push %r4
                push %r5
                push %r6
                push %r7
                push %r8
                push %r9
                push %r10
                push %r28

                // Mise en place du programme 0
                st %sp, [%r19+1]
                set PILE0, %sp
                ba prog0

Tab_sp: .word 0, 0
```

COMMUTATEUR_TACHES_16

```
N = 16
LEDS = 0xB0000000
PILE0 = 0x200

ba init_contexte // instruction à l'@ 0
handler_IT: push %r1 // handler d'IT à l'@ 1
        push %r2
        push %r3
        push %r4
        push %r5
        push %r6
        push %r7
        push %r8
```

```
        push %r9
        push %r10
        push %r28
        st %sp, [%r19 + %r17]
        add %r17, 1, %r17     //ligne mise en commentaire pour le programme1
        cmp %r17, %r18
        blu suite
        clr %r17

suite :    ld[%r17 + %r19], %sp
        pop %r28
        pop %r10
        pop %r9
        pop %r8
        pop %r7
        pop %r6
        pop %r5
        pop %r4
        pop %r3
        pop %r2
        pop %r1
        reti // retour dans le prog interrompu


init_contexte : set Tab_sp, %r19 //%r19 est l'adresse de Tab_sp
            set N, %r18
            PILE = 0x300
            set PILE, %sp
            set 1, %r17

init_bcle : cmp %r17,%r18
        bgeu init_prog0

        set prog, %r11
        push %r11
        push %r0
        push %r1
        push %r2
        push %r3
        push %r4
        push %r5
        push %r6
        push %r7
        push %r8
        push %r9
        push %r10
        push %r28
        st %sp, [%r19 + %r17]
```

```
        add %sp,100,%sp
        inc %r17
        ba init_bcle

init_prog0 : set 0,%r17
        set PILE0,%sp
        ba prog

prog : call affichage1
    call affichage0
    ba prog

affichage1 : push %r2
        push %r1
        set 1, %r1
        sll %r1,%r17,%r1
        set LEDS, %r2
        st %r1,[%r2]
        pop %r2
        pop %r1
        ret

affichage0 : push %r2
        push %r1
        set 0, %r1
        set LEDS, %r2
        st %r1, [%r2]
        pop %r1
        pop %r2
        ret

Tab_sp : .word PILE0
```

COMPTEUR_MODULO_IT

```
// Compteur modulo x avec afichage du compteur lorsque qu'il y a une interruption
PILE = 0x200
LEDS = 0xB0000000
        ba pgrp
handler_IT:   push %r2
        set LEDS, %r2
        st %r1, [%r2]
        pop %r2
        reti

pgrp:           set PILE, %sp
        set 6, %r2  // valeur de x
```

```
compteur:       clr %r1  // r1 - compteur modulo x
compteur_bcle:inc %r1
        cmp %r1, %r2
        be compteur
        ba compteur_bcle


PILE = 0x200

        set PILE, %sp
        set 10, %r6
        set T, %r1
        set 2, %r2
        call cpt_nb_occur
Stop:   ba Stop
```

COMPTEUR_OCCURENCES_TABLEAU

```
/* Compter le nombre d'occurrences d'un élément dans un tableau.
   IN : r1 - adresse du tableau T
        r2 - x
        r6 - m le nombre d'éléments du tableau
   OUT : r3 - nombre d'occurrences de x dans T
*/
cpt_nb_occur:           push %r4
                push %r5
                clr %r4
                clr %r3
cpt_bcle:       ld [%r1 + %r4], %r5
                cmp %r2, %r5
                bne incr_tab
                inc %r3
incr_tab:       inc %r4
                cmp %r4, %r6
                bgeu cpt_ret
                ba cpt_bcle
cpt_ret:        pop %r5
                pop %r4
                ret

T: .word 3, 2, 7, 5, 2, 11, 6, 9, 4, 2
```

ERATOSTHENE

```
// Définition des constantes
PILE = 0x400
N = 110
```

```
        set PILE, %sp                   // Initialisation de la pile
        set Tab_premiers, %r1                   // Accéder à Tab_premiers grâce à r1
        call Eratosthene
Stop:   ba Stop                         // Arrêt du programme principal


/* Initialise Tab_premiers avec les nombres de 0 à N.
   IN : r1 - l'adresse du tableau
*/
Initialiser_tab_premiers: push %r1              // Sauvegarder les registres utilisés
temporairement
                push %r2

                clr %r2
                st %r2, [%r1]
                inc %r1
                st %r2, [%r1]
                inc %r1
                set 2, %r2

Boucle_tab_premiers:            st %r2, [%r1]
                inc %r1
                inc %r2
                cmp %r2, N
                bleu Boucle_tab_premiers

Retour_tab_premiers:            pop %r2                 // Restaurer les registres utilisés
temporairement
                pop %r1
                ret                     // Retourner au programme appelant


/* Elimine les multiples d'un nombre dans Tab_premiers.
   IN : r1 - l'adresse du tableau
        : r2 - le nombre dont on veut supprimer les multiples
*/
Eliminer_multiples:     push %r0                // Sauvegarder les registres utilisés
temporairement
                push %r2
                push %r3
                push %r4

                clr %r0
                set 2, %r3              // Numéro de multiple
Boucle_eliminer_multiples:umulcc %r2, %r3, %r4    // r4 <- i * numéro de multiple
                st %r0, [%r1+%r4]
                inc %r3
                cmp %r4, N
```

```
                bleu Boucle_eliminer_multiples

Retour_eliminer_multiples:pop %r4                   // Restaurer les registres utilisés
temporairement
                pop %r3
                pop %r2
                pop %r0
                ret

/* Trouve tous les nombres premiers entiers inférieurs à N.
  IN : r1 - l'adresse du tableau
  OUT : les nombres premiers i sont les éléments du tableau tels que Tab_premiers[i] = i
*/
Eratosthene:        push %r28           // Sauvegarder l'adresse de retour
                push %r0            // Sauvegarder les registres utilisés temporairement
                push %r2
                push %r3
                call Initialiser_tab_premiers
                clr %r0
                set 2, %r2          // i = r2 <- 2

Tantque_eratosthene:    umulcc %r2, %r2, %r3    // r3 <- i*i
                cmp %r3, N
                bgeu Retour_eratosthene
                ld [%r1+%r2], %r3   // r3 <- Tab_premiers[r2]
                cmp %r3, %r0
                beq Fin_si_eratosthene
                call Eliminer_multiples
Fin_si_eratosthene:  inc %r2
                ba Tantque_eratosthene

Retour_eratosthene:     pop %r3             // Restaurer les registres utilisés
temporairement
                pop %r2
                pop %r0
                pop %r28            // Restaurer l'adresse de retour
                ret

Tab_premiers: .word 0

// Cycles consommés par le programme principal : 1364
// Le module a été testé et fonctionne correctement.
```

FACTORIELLE

```
// programme principal
PILE = 0x200                // fond de pile à l'adresse 0x200
N = 9
```

```
        set PILE, %sp           // initialisation du pointeur de pile : ABSOLUMENT
NECESSAIRE
        set N, %r1
        call Factorielle_recursive      // factorielle(N) – résultat dans r2
Stop :  ba Stop

// Calcule la factorielle d'un entier naturel de façon itérative.
// IN : r1, contient le nombre dont veut calculer la factorielle
// OUT : r2, contient le résultat
Factorielle:    push %r1                // r1 modifié dans le sous-programme : il doit donc être
                                        // sauvegardé dans la pile à l'entrée et restauré à la sortie
                set 1, %r2              // factorielle(0) ou factorielle(1)
Tantque:        cmp %r1, 1
                bleu Retour             // branchement à Retour si r1 <= 1
                umulcc %r1, %r2, %r2            // r2 <- r1 * r2
                dec %r1
                ba Tantque
Retour:         pop %r1                 // restaurer r1 qui doit retrouver sa valeur d'entrée
                ret

// Le registre r28 contient l'adresse de la dernière instruction call (ici call Factorielle)

/* Le pointeur de pile (registre r29) contient 0x00000200 au début et à la fin du
sous-programme,
  ce qui est normal.
  Pendant l'exécution du sous-programme, le pointeur de pile a la valeur 0x000001ff
  (une valeur de plus dans la pile, celle pour stocker la valeur initiale de r1)
*/

/* La valeur maximale qu'il est possible de stocker dans un registre est 0xffffffff (entiers non
signés),
  c'est-à-dire 4294967295 en décimal.
  Mais umulcc n'utilise que les 16 bits de faibles poids, car sinon on aurait un dépassement
à cause de
  la multiplication. Donc finalement, la valeur maximale est 0x0000ffff (soit 1048335 en
décimal).
  Ainsi, la valeur maximale de N est 9.
*/

// Calcule la factorielle d'un entier naturel de façon récursive.
// IN : r1, contient le nombre dont veut calculer la factorielle
// OUT : r2, contient le résultat
Factorielle_recursive :   push %r28             // sauvegarder l'adresse de retour
                push %r1
                set 1, %r2
                cmp %r1, 1
                bleu Retour_FR          // branchement à Retour_FR si r1 <= 1
                dec %r1
```

```
                call Factorielle_recursive
                inc %r1
                umulcc %r1, %r2, %r2
Retour_FR :             pop %r1
                pop %r28        // restaurer l'adresse de retour
                ret
```

INTERRUPTION_COMPTEUR

```
PILE = 0x200
LEDS = 0xB0000000
        ba progp
handler_IT:   push %r1
        push %r2
        push %r20
        set LEDS, %r20
        set compteur_IT, %r1
        ld [%r1], %r2
        inc %r2
        st %r2, [%r20]
        st %r2, [%r1]
        pop %r20
        pop %r2
        pop %r1
        reti            // retour dans le programme interrompu

progp:          set PILE, %sp
        set compteur_IT, %r1
        st %r0, [%r1]
pbcle:          nop
        nop
        nop
        ba pbcle

compteur_IT:  .word 0
```

INVERSER

```
// Empile les éléments de Chaine dans la PILE
// 00000044  Tour 4
// 00000043  Tour 3
// 00000042  Tour 2
// 00000041  Tour 1
PILE = 0x200                    // fond de pile à l'adresse 0x200
        set PILE, %sp           // initialisation du pointeur de pile
        set Chaine, %r1
        clr %r2         // %r2 <- 0 : nombre d'éléments
Repeter:        ld [%r1], %r3
```

```
        cmp %r3, %r0            // r3 ? 0
        beq Inverser
        push %r3        // %r3 -> sommet de pile
        inc %r2         // add %r2, 1, %r2
        inc %r1         // adresse du prochain élément
        ba Repeter
Inverser:   set Chaine, %r1
Repeter2:   pop %r3
        st %r3, [%r1]
        inc %r1
        deccc %r2
        bgt Repeter2
Stop :  ba Stop
Chaine :        .word 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0   // 0 = fin d chaine
```

// Le module a été testé et fonctionne correctement

MIN_TABLEAU

```
N = 11          // N constante, pour éviter d'utiliser des valeurs en dur dans le programme
        set Tab1, %r1
        clr %r3         // indice de boucle
        set 0x7FFFFFFF, %r5  // %r5 stocke le minimum du tableau
        clr %r6         // %r6 stocke l'indice du minimum du tableau
Tantque :   cmp %r3, N
        bgeu Stop               // branch if r3 greater or equal unsigned to N
        ld [%r1+%r3], %r4       // ld = load - %r2+%r3]= adresse de tab(r3)
        cmp %r4, %r5
        bge Incr
        mov %r4, %r5
        mov %r3, %r6
Incr :  add %r3, 1, %r3         // index <- index + 1
        ba Tantque
Stop :  ba Stop
Tab1 :          .word 10, 9, 8, 7, 1, 6, 5, 4, -3, 2, 1
```

// Définir le nombre binaire dont on va vérifier si c'est un palindrome ou non

```
        set 0x00024004, %r1

        // Initialiser la pile à 0x200
        set 0x200, %sp
        call palindrome_bin_inv2
Stop:   ba Stop
```

```
// Vérifier si un nombre est palindrome binaire
// Méthode : calcul du nombre inverse (faible poids <-> fort poids) et comparaison avec le nombre initial
// IN : r1, nombre
// OUT : r2 = 1 si palindrome, 0 sinon
palindrome_bin_inv:   push %r3                // sauvegarde des registres modifiés
                      push %r4
                      push %r5
                      push %r6
                      set 1, %r2              // résultat <- 1
                      set 32, %r3             // nombre de bits
                      mov %r1, %r4            // copie du nombre
                      clr %r5          // nombre inverse
pbi_bcle:             andcc %r2, %r4, %r6   // isoler bit0
                      sll %r5, 1, %r5         // r5 *2
                      or %r5, %r6, %r5        // + bit0
                      slr %r4, 1, %r4         // décaler d'une position -< droite
                      deccc %r3
                      bne pbi_bcle
                      cmp %r1, %r5            // nombre intial ?= nombre inverse
                      beq pbi_ret             // palindrome : r2 est déjà =1
                      clr %r2          // non palindrome
pbi_ret:              pop %r6                 // restauration des registres
                      pop %r5
                      pop %r4
                      pop %r3
                      ret

/* r1 = 0 : 210 cycles consommés
   r1 = 00000000010010011001001000000000 (0x00499200) : 210 cycles consommés
   r1 = 00000000000000100100000000000000 (0x00024000) : 210 cycles consommés
   r1 = 00000000000000110100000000000000 (0x00034000) : 211 cycles consommés
   r1 = 00000000000000100100000000000100 (0x00024004) : 211 cycles consommés
*/
```

PALINDROME_BINAIRE

```
// Vérifier si un nombre est palindrome binaire
// Méthode : calcul du nombre inverse (faible poids <-> fort poids) et comparaison avec le nombre initial
// IN : r1, nombre
// OUT : r2 = 1 si palindrome, 0 sinon
palindrome_bin_inv2:  push %r3                // sauvegarde des registres modifiés
                      push %r4
                      push %r5
                      push %r6

                      set 1, %r2              // résultat <- 1
```

```
                set 0x00000001, %r10  // pour accéder au bit 0
                set 0x80000000, %r11  // pour accéder au bit 31
                set 31, %r3           // nombre de décalages du bit de poids fort
                set 16, %r5           // nombre de tours de boucles (nombre d'opérations)
                mov %r1, %r4          // copie du nombre

pbi_bcle2:          andcc %r10, %r4, %r6  // isoler bit de poids faible
                andcc %r11, %r4, %r7  // isoler bit de poids fort
                slr %r7, %r3, %r7     // décaler le bit de poids fort vers la droite
                cmp %r6, %r7          // comparer le bit de poids faible avec celui de poids
fort
                bne non_pbi2          // on arrête si les deux bits symétriques sont différents

                sll %r10, 1, %r10     // numéro de bit de poids faible suivant
                slr %r11, 1, %r11     // numéro de bit de poids fort suivant
                slr %r7, %r3, %r7
                sub %r3, 2, %r3       // réduire de 2 le décalage du bit de poids fort
                                // (car on a fait 2 décalages)
                deccc %r5             // réduire le nombre de tours de boucle restants
                bgu pbi_bcle2
                ba pbi_ret2

non_pbi2:           clr %r2           // non palindrome
pbi_ret2:       pop %r6               // restauration des registres
                pop %r5
                pop %r4
                pop %r3
                ret


/* r1 = 0 : 195 cycles consommés
  r1 = 00000000010010011001001000000000 (0x00499200) : 195 cycles consommés
  r1 = 00000000000000100100000000000000 (0x00024000) : 195 cycles consommés
  r1 = 00000000000000110100000000000000 (0x00034000) : 189 cycles consommés
  r1 = 00000000000000100100000000000100 (0x00024004) : 46 cycles consommés

  On remarque que le nombres de cycles consommés réduit drastiquement lorsque
  le nombre binaire contient une asymétrie "sur ses extrémités" comme c'est le
  cas du nombre 0x00024004.
*/

// Le module a été testé et fonctionne correctement.
```

TRI_SELECTION

```
// programme principal
PILE = 0x200                    // fond de pile à l'adresse 0x200
```

```
N = 20          // N constante, pour éviter d'utiliser des valeurs en dur dans le programme

        set PILE, %sp           // initialisation du pointeur de pile : ABSOLUMENT
NECESSAIRE
        set Tab1, %r1
        clr %r2         // compteur de la boucle principale
        set N-2, %r11           // r11 <- N-2
        call Tri
        call Test_Pgr


/* Tri d'un tableau.
  IN : r1 - le tableau a trié
  OUT : r1 - le tableau trié
*/
Tri :   push %r28
Tri_Boucle: call cal_min

        // Echange de Tab[I] et de Tab[Indice_Min]
        ld [%r1+%r2], %r12    // r12 <- Tab[I]
        ld [%r1+%r6], %r13    // r13 <- Tab[Indice_Min]
        st %r13, [%r1+%r2]    // Tab[I] <- r13 = Tab[Indice_Min]
        st %r12, [%r1+%r6]    // Tab[Indice_Min] <- r12 = Tab[I]

        inc %r2
        cmp %r2, %r11
        bgu Retour_Tri
        ba Tri_Boucle
Retour_Tri: pop %r28
        ret

/* Calcule le minimum d'un tableau entre les indices r2 et N-1.
  IN : r2 l'indice du début
  OUT : r5 le minimum du tableau
        r6 l'indice du minimum
*/
        cal_min :   mov %r2, %r3                // indice de boucle
                set 0x7FFFFFFF, %r5             // %r5 stocke le minimum du tableau
                clr %r6                 // %r6 stocke l'indice du minimum du tableau
        Tantque :   cmp %r3, N
                bgeu Retour_CM                  // branch if r3 greater or equal unsigned to N
                ld [%r1+%r3], %r4       // ld = load - %r2+%r3]= adresse de tab(r3)
                cmp %r4, %r5
                bge Incr
                mov %r4, %r5
                mov %r3, %r6
        Incr :  add %r3, 1, %r3         // index <- index + 1
                ba Tantque
```

```
        Retour_CM : ret


/* Test le programme principal.
   IN : r1, indice du tableau
   OUT : r16 contient 0 si le test a réussi, 0 sinon
*/
        Test_Pgr :  clr %r2      // indice de parcours
                    clr %r16
                    set 1, %r3  // r3 <- 1
        Test_Boucle:ld [%r1+%r2], %r14       // r14 <- Tab[I]
                    add %r2, %r3, %r4     // r4 <- r2 + 1
                    ld [%r1+%r4], %r15  // r15 <- Tab[I+1]
                    cmp %r14, %r15
                    bgt Fail          // branchement si Tab[I] > Tab[I+1]
                    inc %r2
                    cmp %r2, %r11
                    bgu Stop
                    ba Test_Boucle
        Fail :    set 1, %r16               // r16 <- 1 si le test a échoué

Stop :  ba Stop
Tab1 :          .word 10, 9, 8, 0, 7, 6, 5, 4, 3000, -3, 2, 1, 5, 25, -12, 54, -85, 7, -1, 0

// Nombre de cycles consommés (10 éléments) : 586
// Nombre de cycles consommés (20 éléments) : 2057
```