

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 [ReponsesTP2.txt](#) et mettez votre nom/prénom et groupe de TP aux endroits indiqués. Ne modifiez surtout pas la structure de ce fichier.

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

👉 Mettez CimPU en niveau 3.

1. Présentation générale

Le TP2 porte sur l'écriture de traitements conditionnels : comment faire pour exécuter ou non une partie du programme selon une condition, et comment faire pour recommencer une partie du programme tant qu'une condition est vraie ou jusqu'à ce qu'elle soit fausse. C'est ce qu'on appelle des *structures de contrôle* : alternatives et boucles définissent le déroulement du programme.

Voici une conditionnelle typique :

```
si (v1 > 3)
alors
    // bloc alors
    ...
sinon
    // bloc sinon
    ...
finsi
```

Il y a deux points à comprendre. Il y a d'abord un calcul de la condition, pour savoir si elle est vraie ou fausse. Dans l'exemple précédent, il faut comparer $v1$ à 3. Ensuite, selon que la condition est vraie ou fausse, il faut continuer soit dans le *bloc alors*, soit dans le *bloc sinon*. Et dans tous les cas, continuer après le *finsi*. Ça se fait avec des branchements conditionnels ou non.

Donc nous allons voir comment coder ces deux aspects en assembleur, condition et branchements. On commence par le calcul de la condition.

2. Codage d'un calcul de condition simple

On commence par les conditions simples, celles qui s'écrivent (*partie gauche opérateur partie droite*). Par exemple si la condition est $(N1-1 \leq N2+7)$, alors la partie gauche est $N1-1$, la partie droite est $N2+7$, et l'opérateur est \leq .

Pour évaluer une telle condition, c'est à dire savoir si elle est vraie ou fausse, le principe est d'écrire des instructions qui permettent de modifier les indicateurs **VCNZ** de l'unité de calcul pour qu'ils reflètent la condition. Par exemple, **Z** passe à 1 quand le résultat d'un calcul est nul. La

comparaison de deux valeurs consiste à les soustraire, et donc le résultat est nul quand les valeurs sont égales. C'est à dire que Z passe à 1 quand les valeurs comparées sont égales.

Donc il y a d'abord un chargement des valeurs en question dans les registres, LD R0, [variable], éventuellement des calculs, ex: ADD R0, ... selon la condition. Ça finit par une comparaison CMP R0, val avec val qui est une constante ou un registre ou un accès mémoire comme [variable]. C'est cette comparaison qui modifie les indicateurs VCNZ. Ça sera utilisé ensuite par les branchements.

On est obligé de passer par les registres, parce que l'instruction CMP est limitée. On ne peut pas comparer directement deux cellules mémoire, et souvent, il faut faire des calculs.

Le principe général, c'est que, au final il faut faire CMP *partie gauche*, *partie droite* quand on a une condition qui s'écrit (*partie gauche opérateur partie droite*).

Par exemple si la condition est (N1 > N2+7), alors on programme le calcul ainsi :

```
si:    LD R0, [N1]      ; R0 = partie gauche
        LD R1, [N2]      ; calcul de la partie droite
        ADD R1, 7        ; R1 = partie droite
        CMP R0, R1      ; comparer partie gauche et partie droite
```

C'est à dire qu'on fait en sorte que les deux parties, gauche et droite, de la condition soient calculées, l'une dans R0, l'autre dans R1 et on les compare.

Il peut être intéressant de simplifier au maximum les conditions, par exemple N1-6 > N2+7 peut devenir N1-N2 > 13, qu'on calculerait comme ceci :

```
si:    LD R0, [N1]      ; calcul de la partie gauche simplifiée
        SUB R0, [N2]      ; R0 = partie gauche simplifiée
        CMP R0, 13       ; partie droite = constante, pas de calcul
```

Donc la démarche est simple et toujours la même :

1. Simplifier la condition au maximum pour avoir le moins de calculs à faire. Ce qui peut être utile, c'est de placer les registres d'un côté et les constantes de l'autre et de faire tous les calculs. Placer les constantes à droite permet de simplifier l'instruction CMP.
2. Écrire les instructions qui placent chaque partie de la condition dans des registres, ex: R0 contient la partie gauche et R1 la partie droite.
3. Utiliser CMP pour comparer les deux parties.

2.1. Exemples

Voici quelques exemples. Regardez comment les conditions sont programmées :



```
;; programme
; DSKY en mode 11 (terminal texte)
LD R0, 11
OUT R0, 255

; V1 == 0 ?
LD R0, [V1]
CMP R0, 0
```

```
PRTCOND      ; 1: affiche "=" car V1==0 => indicateur Z = 1
              ; la condition est vraie

; V2 < 10 ?
LD  R0, [V2]
CMP R0, 10
PRTCOND      ; 2: affiche ">u <s", car 254>2 en non-signé, -2<2 en signé
              ; la condition est vraie en signé, mais fausse en non-signé

; V1 > V2 ?
LD  R0, [V1]
CMP R0, [V2]
PRTCOND      ; 3: affiche "<u >s" car 254 > 1 en non-signé et -2 < 1 en signé
              ; la condition est vraie en non-signé, mais fausse en signé

; V1 < V3 ?
LD  R0, [V1]
CMP R0, [V3]
PRTCOND      ; 4: affiche "<u <s" car 0 < 1 en non-signé et signé
              ; la condition est vraie

; V2 + V3 < 5 ?
LD  R0, [V2]
ADD R0, [V3]
CMP R0, 5
PRTCOND      ; 5: affiche ">u <s" car -2+1=255, 255>5 en non-signé, -1<5 signé
              ; la condition est vraie en signé, et fausse en non-signé

; -V2 > V1 + V3 ?
LD  R0, [V2]
NEG R0
LD  R1, [V1]
ADD R1, [V3]
CMP R0, R1
PRTCOND      ; 6: affiche ">u >s" car 2>1 en non-signé et signé
              ; la condition est vraie

; fin du programme
HLT

;; variables du programme (les valeurs peuvent être modifiées ailleurs)
V1:  DB  0
V2:  DB -2      ; -2 signé = 254 non signé
V3:  DB  1
```

L'instruction `PRTCOND` (*print condition*) est très spéciale. On ne la trouve pas sur un processeur normal. Elle sert uniquement pour le débogage des conditions. Elle affiche un texte dans le DSKY

en mode 11 qui exprime ce qu'a observé l'instruction `CMP`, d'après les indicateurs `VCNZ`. C'est une chaîne contenant `<`, `=` ou `>`. Par exemple, quand `Z=1`, elle affiche `"="`. Les signes `<` et `>` sont suivis d'un `u` pour les comparaisons non signées et d'un `s` pour les comparaisons signées. Il y a une différence entre elles : si vous comparez 172 à 23, vous aurez `"u> <s"` parce que 172 non signé est plus grand que 23, mais 172 est aussi le codage $C2^8$ de -84 , et -84 est plus petit que 23.

Pour voir tous les affichages, regardez dans la section Périphériques juste au dessus de l'Unité centrale.

Ça peut vous sembler bizarre qu'on fasse toujours la même chose quelque soit l'opérateur de comparaison dans la condition. Mais c'est l'instruction de saut qui est concernée par l'opérateur. Le saut n'est pas le même selon qu'on compare par `=`, `<=`, etc. On verra ça dans le § suivant.

2.2. Exercices

👉 Codez le calcul des conditions suivantes dans ce programme :



```
;; programme

; V2 == 9 ?
PRTCOND          ; 1: ne doit pas afficher = car V2 != 9

; V2 - 9 == 14 ?
PRTCOND          ; 2: doit afficher = car la condition est vraie

; V1 != V2 + 5 ?
PRTCOND          ; 3: ne doit pas afficher = car la condition est vraie

; V1 - V2 == -18 ?
PRTCOND          ; 4: doit afficher = car la condition est vraie

; V1 + V2 - 1 < 30 ?
PRTCOND          ; 5: doit afficher <s car la condition est vraie

; V1 + 1 > V2 + 3 ?
PRTCOND          ; 6: ne doit pas afficher >s car la condition est fausse

; 3*V1 - 1 >= V1 + 9 ?
PRTCOND          ; 7: doit afficher = ou >s car la condition est vraie

; V2 > V1*4 ?
PRTCOND          ; 8: doit afficher >s car la condition est vraie

; V1-1 > V2+V1-V2 ?
; non mais ça va pas ? cette condition toujours fausse n'est pas à coder !

; fin du programme
HLT
```

```
;; variables du programme toutes signées !  
V1:    DB 5  
V2:    DB 23
```

☞ Simplifier chaque condition au maximum si vous pouvez : amener toutes les constantes à droite et les variables à gauche, et programmer les calculs. Suivez les exemples précédents.

ATTENTION: ne changez pas la structure de ce programme (commentaires et lignes vides). Mettez vos réponses entre chaque commentaire et l'instruction `PRTCOND` et laissez bien une ligne vide après. Si vous enlevez le commentaire, le `PRTCOND` ou la ligne vide, votre réponse ne sera pas prise en compte.

☞ Mettez le DSKY en mode 11. Testez le programme. L'intégralité des affichages, numérotés, est dans la section Périphériques.

☞ Une fois fini, copiez-collez ce programme dans `ReponsesTP2.txt`.

3. Saut selon la condition

Il reste à prendre en compte l'opérateur de la condition, `==`, `>`, `<=`, etc. Cet opérateur se traduit en une instruction de branchement, c'est à dire une instruction qui dit : « va là-bas si la condition est fausse ». C'est ce qu'il faut comprendre, c'est qu'on fait un « saut » si la condition est fausse, et on continue si elle est vraie.

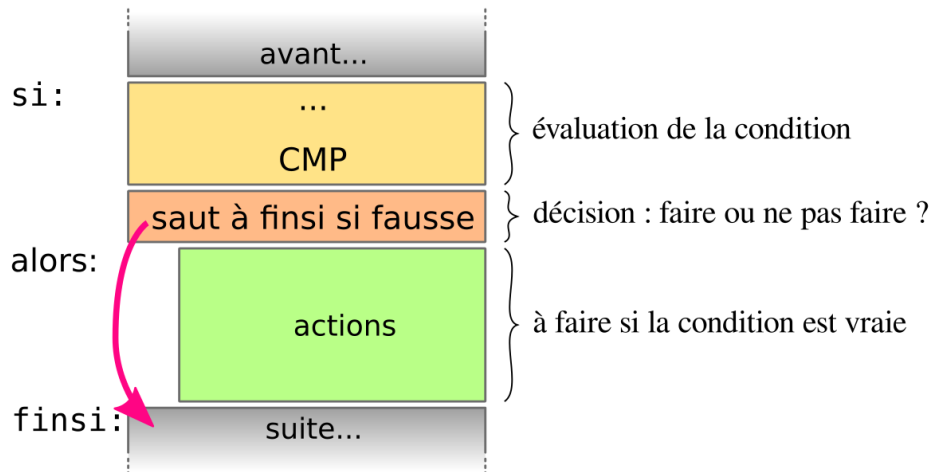


Figure 1: Saut à FINSI si la condition est fausse

Donc, forcément, tout repose sur l'instruction qui fait le saut. C'est elle qui consulte les indicateurs `VCNZ` affectés par l'instruction `CMP`. On doit choisir le saut qui correspond à la condition. Pour cela, il faut utiliser l'un des tableaux suivants qui indique quelle instruction de saut il faut selon l'opérateur.

Il y a deux tableaux. L'un pour les comparaisons d'entiers non signés, l'autre pour les comparaisons d'entiers signés. Comme écrit précédemment : $172 > 23$ en non-signé, mais 172 représente -84 et $-84 < 23$ en comparaison signée. Il est donc important de savoir dans quelle convention se trouvent vos entiers à comparer.

3.1. Sauts pour ces comparaisons non signées

op	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

Ce tableau vous donne l'instruction de saut à choisir selon l'opérateur de la condition, pour sauter quand la comparaison entre deux entiers non signés est fausse.

3.2. Sauts pour ces comparaisons signées

op	Branchement	Signification
<	BGE	Branch if greater or equal
≤	BGT	Branch if greater than
=	BNE	Branch if not equal
≠	BEQ	Branch if equal
≥	BLT	Branch if less than
>	BLE	Branch if less or equal

Ce tableau vous donne l'instruction de saut à choisir selon l'opérateur de la condition, pour sauter quand la comparaison entre deux entiers en convention $C2^8$ est fausse.

4. Codage des structures de contrôle

On va réviser [TD7](#) et [TD8](#), ainsi que le [CM3](#).

Voici par exemple comment on code une alternative *si/alors/sinon*.

Langage C

```
-----
if (condition)
{
    action1;
}
else {
    action2;
}
-----
```

CimPU

```
-----
si:    ... calcul de la condition ...
        CMP reg, ...
        B?? sinon    ; saut si faux

alors:
        action1...
        BRA finisi    ; sauter toujours

sinon:
        action2...

finisi:
-----
```

Vous devez choisir l'instruction B?? en fonction de l'opérateur *op* de la condition, à l'aide des tableaux.

N'oubliez pas le **BRA fin** à la fin du bloc alors. Si vous l'oubliez, le processeur exécutera la partie sinon à la suite de la partie alors.

NB: quand il y a plusieurs structures de contrôle, chacune possédant ses labels, il faut numéroter les labels. Par exemple **si1, alors1, si2, alors2...**

Une boucle TantQue ressemble structurellement (quand on la code avec des instructions) à une conditionnelle. La seule différence, c'est l'ajout d'un **BRA** à la fin des actions à faire quand la condition est vraie, pour revenir au test de la condition.


Langage C

```
-----  
while (condition)  
{  
    action1;  
}
```

CimPU

```
-----  
tantque:  ... calcul de la condition ...  
          CMP reg, ...  
          B?? fiantque    ; saut si faux  
faire:  
          action1...  
          BRA tantque    ; sauter toujours  
fiantque:  
-----
```

5. Exercice : dessin brouillé

☛ Complétez ce programme. Il y a deux alternatives imbriquées, donc attention à numéroter les labels. 

```
; mettre le DSKY en mode 8 = LCD matriciel  
LD R0, 8  
OUT R0, 255          ; port 255 = choix du mode du DSKY = 8  
  
;; recopier les données sur le port 8  
  
; indice = 0  
LD R1, 0  
repete:  
; lire le code à l'indice R1  
LD R0, [R1+donnees]  
  
; TODO programmez ces deux conditionnelles  
; if (R0 != $AA) {  
;     if (R0 != $55) {  
  
; afficher le code contenu dans R0 sur le LCD  
OUT R0, 8  
  
;     }  
; } ; fin du 2e if
```

```
        ; }      ; fin du 1er if

        ; passer au code suivant
        INC R1          ; R1++

jusqua: CMP R0, 0        ; calcul de la condition (R0 == 0)
        BNE repeter     ; retour à repeter si condition fausse
        HLT

donnees:
        DB %01010101
        DB %01111111
        DB %10101010
        DB %01010101
        DB %11101110
        DB %10101010
        DB %10111100
        DB %01010101
        DB %10101010
        DB %11011100
        DB %10101010
        DB %01010101
        DB %10111100
        DB %11101110
        DB %10101010
        DB %01010101
        DB %01111111
        DB 0
```

👉 Terminez-le, assemblez-le et essayez-le.

👉 Une fois qu'il marche, copiez-collez ce programme dans `ReponsesTP2.txt`.

Vérifiez maintenant si votre `ReponsesTP2.txt` est bien enregistré. Si vos quotas disque sont épuisés, les nouveaux fichiers sont vides ! Utilisez du `-hs ~` pour savoir combien de place vous occupez et `rm -fr .cache` pour récupérer un peu de place.

6. Exercice : division euclidienne

👉 Complétez ce programme :



```
;; division euclidienne d'un entier par un autre
        ; lire un nombre sur le port 10 et le mettre dans N1
        IN  R0, 10
        ST  R0, [N1]
        ; lire un autre nombre sur le port 10 et le mettre dans N2
        IN  R0, 10
        ST  R0, [N2]
```



```
; partie à programmer
; if (N2 > 0) {
;     Q = 0
;     while (N1 >= N2) {
;         N1 = N1 - N2
;         Q = Q + 1
;     }
; } else {
;     Q = 255
;     N1 = 255
; }

; afficher Q puis N1 sur le port 10
LD R0, [Q]
OUT R0, 10
LD R0, [N1]
OUT R0, 10
; fin du programme
HLT

;; variables du programme
N1:    RB 1    ; RB = reserve bytes = réservation de 1 octet non initialisé
N2:    RB 1
Q:     RB 1
```

NB: il y a une mise à zéro de Q avant la boucle.

☛ Terminez-le, assemblez-le et essayez-le avec différents nombres. Ce programme affiche le quotient et le reste de la division du premier nombre par le second.

☛ Une fois qu'il marche, copiez-collez ce programme dans `ReponsesTP2.txt`.

7. Exercice : Marche aléatoire

En mathématique, économie et physique, une [marche aléatoire](#) est un modèle de déplacement déterminé par le hasard. Plus précisément, l'objet mobile fait des pas dont la direction et la longueur dépendent du hasard. Par exemple, des sauterelles sont situées un jour à tel endroit, le lendemain elles se sont déplacées à un autre endroit situé à une distance et un cap (direction) plus ou moins aléatoire, par exemple à cause du vent. On s'intéresse à l'endroit où elles peuvent être au bout d'un certain temps.

On va simplifier le problème : on a un [pois sauteur](#) situé à une position représentée par un octet signé, -128 à $+127$. Il a une probabilité de $1/4$ de sauter de 2 cases vers la droite, $1/4$ de sauter de 1 case à gauche, $1/6$ de sauter de 3 cases vers la gauche, et le reste de 1 case à droite. On lui permet de faire 50 mouvements et on voudrait savoir où il est arrivé.

L'idée est de générer un nombre aléatoire *rnd* compris entre 0 et 255. On l'obtient en lisant le port 254. Et on compare ce nombre à différentes bornes étagées pour représenter les probabilités de mouvement. Ainsi on peut écrire, avec quelques approximations, que :

- si (rnd < 64) alors position += 2
- sinon si (rnd < 128) alors position -=1
- sinon si (rnd < 171) alors position -=3
- sinon position += 1
- finsi

Comment coder une chaîne de conditionnelles *sinon si* ? C'est toujours la même logique.

Langage C

```
-----
if (condition1)
{
    action1;
}
else if (condition2)
{
    action2;
}
else if (condition3)
{
    action3;
}
...
else {
    action sinon;
}
-----
```

CimPU

```
-----
si1:    ... calcul de la condition1 ...
        CMP reg, ...
        B?? si2    ; saut si faux
alors1:
        action1...
        BRA finsi   ; sauter toujours
si2:    ... calcul de la condition2 ...
        CMP reg, ...
        B?? si3    ; saut si faux
alors2:
        action2...
        BRA finsi   ; sauter toujours
si3:    ... idem ...
alors3: ... idem ...
        ...
        BRA finsi   ; sauter toujours
sinon:
        action sinon...
finisi:
-----
```

👉 Complétez ce programme :



```
;; marche aléatoire d'un pois sauteur
; position = 0
; mouvements = 0

; while (mouvements < 50) {      // non signée

;     lire un nombre aléatoire dans R0
IN R0, 254

;     if (R0 < 64) {              // non signée
;         position += 2
;     } else if (R0 < 128) {      // non signée
;         position -= 1
;     } else if (R0 < 171) {      // non signée
;         position -= 3
```

```
;      } else {  
;          position += 1  
;      }  
;      incrémenter mouvements  
; } // fin du while  
  
; afficher position sur le port 10  
LD R0, [position]  
OUT R0, 10  
; fin du programme  
HLT  
  
;; variables du programme  
position:      DB  0  
mouvements:   DB  0
```

👉 Terminez-le, assemblez-le et essayez-le...

Avant de paniquer parce que ça vous semble trop complexe, constatez que vous pouvez coder la première condition seulement et vous aurez déjà un début de fonctionnement correct. Ensuite codez la deuxième condition et ça s'améliorera. Et ainsi de suite. En fait, ce programme, c'est seulement une boucle avec 50 tours, et quelques conditions à la suite.

👉 Une fois qu'il marche, copiez-collez ce programme dans `ReponsesTP2.txt`.

8. Exercice : méthode de Héron

Voici un dernier algorithme mathématique. Il permet de calculer la racine carrée d'un nombre N par approximations successives. L'idée de Héron (1^{er} siècle avant JC) est de trouver le nombre n tel que $n * n = N$ en construisant des rectangles dont la surface reste égale à N et qui sont « de plus en plus carrés », c'est à dire dont les côtés sont de plus en plus égaux. Au début, le rectangle est allongé et à la fin, le rectangle est carré (largeur = hauteur). À chaque itération, la surface reste égale à N , donc à la fin, le côté du rectangle est la racine carrée de N .

La question est : comment construit-on un rectangle ayant un côté de longueur n tel que sa surface soit N ? Ce que propose Héron, c'est de considérer que l'autre côté est N/n , comme ça la surface du rectangle est $n \times (N/n) = N$. Alors quand, en plus, on a l'égalité des deux côtés, c'est à dire $n = N/n$, c'est que n est la racine carrée de N . Mais comment arriver à ça? Tout simplement, dit Héron, en calculant la moyenne entre n et N/n et en la remettant dans n . Ainsi, à chaque itération, n évolue vers une valeur qui est de plus en plus proche de N/n .

Donc, ce qu'il faut, c'est initialiser n à une valeur quelconque comprise entre 1 et N , et réaffecter n avec $(n + (N/n))/2$ jusqu'à ce que n ne change plus, ou qu'on ait fait un certain nombre de tours.

👉 Mettez CimPU en niveau 4. L'instruction de division DIV n'est pas disponible en dessous de ce niveau.

Voici un programme à compléter. Le calcul compliqué vous est fourni :



```
;; calcul de la racine carrée d'un entier par la méthode de Héron
; lire un nombre sur le port 10 et le mettre dans N
; NB: ce nombre est nécessairement non signé : 0..255
IN  R0, 10
ST  R0, [N]
LSR R0          ; n = N/2 pour le point de départ
ST  R0, [n]

; partie à programmer
; i=8
; répéter {

;     calculer n = (n + N/n) / 2
LD  R0, [N]      ; R0 = N
DIV R0, [n]      ; R0 = N/n
ADD R0, [n]      ; R0 = (N/n) + n
LSR R0          ; R0 = ((N/n) + n) / 2
ST  R0, [n]      ; n = ((N/n) + n) / 2

;     i = i - 1;
; } jusqu'à (i==0);

; afficher n sur le port 10
LD  R0, [n]
OUT R0, 10
; fin du programme
HLT

;; variables du programme
N:   RB 1      ; réservation de 1 octet non initialisé pour cette variable
n:   RB 1
i:   RB 1
```

👉 Terminez-le, assemblez-le et essayez-le avec différents nombres, en vérifiant à la calculatrice. Comment ça, on n'a que des entiers (en plus, on ne peut pas calculer racine de 255) ? Voilà pourquoi on cherche à avoir des processeurs dont les nombres sont plus grands qu'un octet (processeurs 64 bits par exemple).

👉 Une fois qu'il marche, copiez-collez ce programme dans **ReponsesTP2.txt**.

Au passage, vous avez vu comment on calcule une expression numérique assez complexe. Ça peut être très compact et très efficace sur un processeur.

NB: vous voyez que la variable *i* n'est pas vraiment nécessaire, car on peut utiliser *R1* à sa place.

Pensez à déposer votre **ReponsesTP2.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.