Ce TD va permettre de comprendre comment les programmes peuvent effectuer des traitements conditionnels : en modifiant sous condition la valeur de `IP`. Changer `IP` fait « sauter » à une autre instruction que celle qui est normalement la suivante. L'exécution du programme se poursuit plus loin, en ignorant certaines instructions.

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

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

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

1. Indicateurs

1.1. Expérimentations

Le premier concept est celui des indicateurs booléens `V`, `C`, `N` et `Z` dessinés en bas de l'UAL. Ils sont modifiés à chaque calcul fait dans l'UAL. Voici quelques essais à faire.

👉 Assemblez ce programme :

```
        IN  R0, 10      ; lire le premier nombre
        IN  R1, 10      ; lire le second nombre
        ADD R0, R1
        OUT R0, 10      ; afficher le résultat
        HLT
```

👉 Lancez le programme (`Rst+RUN` à chaque fois). Saisissez ces différentes valeurs et à chaque fois examinez `V`, `C`, `N` et `Z` pour comprendre quand ils passent à 0 ou à 1.

- Saisir 10 puis 10 : le résultat est 20. Les indicateurs restent à 0. On est dans aucun cas particulier.
- Saisir 0 puis 0 : `Z` (*zero*) passe à 1 car le résultat est nul. Les autres indicateurs sont à 0.
- Saisir -10 puis 0 : `N` (*negative*) passe à 1 car le résultat est négatif.
- Saisir 150 puis 10 : `N` est également 1, parce que, d'un certain point de vue, le résultat est négatif. Mais si on considère qu'on travaille avec des entiers non signés, alors on n'a pas à s'occuper de `N`.
- Saisir 120 puis 10 : `N` est passé à 1 parce que le résultat est négatif, et `V` (*overflow*) est passé à 1 aussi pour signaler que c'est une erreur : 120 en convention $C2^8$ est positif, 10 aussi, mais leur somme est négative dans cette convention, c'est donc un dépassement de possibilité de représentation, le résultat est faux. `V` à 1 signale toujours une erreur de signe.
- Saisir 250 puis 10 : `C` (*carry*) passe à 1, c'est la retenue sur 8 bits. C'est à dire que le résultat écrit en base 2 serait sur 9 bits, $260 = (100000100)_2$.
- Saisir -10 puis 10 : le résultat est nul donc `Z` est à 1. D'autre part, $-10 = 246$ en convention $C2^8$, donc le calcul $-10 + 10 = 246 + 10 = 256$ produit une retenue, donc `C` est aussi à 1.

En conclusion :

- Si on travaille avec des entiers naturels, non signés, on ne doit s'occuper que de `Z` et `C`. `Z` indique qu'on a obtenu 0. `C` est la retenue du calcul.

- Inversement, si on travaille avec des entiers relatifs en convention $C2^8$, alors on doit regarder seulement `Z`, `N` et `V`. `N` indique un résultat négatif en convention $C2^8$ et `V` indique que le résultat est faux en convention $C2^8$.

👉 Remplacez l'instruction `ADD` par `SUB` et essayez ceci :

- Saisir 20 puis 10 : le résultat est 10 qui n'a rien de spécial, donc les indicateurs restent à 0.
- Saisir 10 puis 10 : le résultat est nul sans erreur, donc seul `Z` passe à 1.
- Saisir 10 puis 20 : le résultat est négatif, donc `N` passe à 1, mais si on considère les nombres comme non-signés, soustraire 20 de 10 impose une retenue : le résultat, 246, n'est correct que parce qu'on a emprunté une retenue à gauche, c'est à dire $(100000000)_2 = 256$. En réalité, on a calculé $10 + 256 - 20 = 246$. Cette situation est représentée par cette retenue.
- Saisir 140 puis 20 : `V` passe à 1 car le résultat en convention $C2^8$ est positif, donc faux. Attention, c'est un piège : 140 est la représentation de -116. Or $-116 - 20 = -136$ qui ne peut pas être écrit en convention $C2^8$. Donc on ne doit pas s'occuper de `V`.
- Saisir 20 puis 140 : en non-signé, la soustraction demande un emprunt (retenue) $20 + 256 - 140 = 136$, donc `C` passe à 1. En convention $C2^8$, il y a un souci car on veut soustraire 140 de 20, or 140 est la représentation de -116, donc en fait, ce qu'on fait, c'est $20 - (-116) = 20 + 116 = 136$. Or ce dernier est la représentation $C2^8$ d'un nombre négatif, donc il y a une erreur car la somme de deux nombres positifs ne peut pas être négative, donc `V` passe à 1. Et en plus, comme le résultat semble négatif, `N` est à 1.

1.2. Comparaisons entre nombres

Pour écrire des conditions, on va avoir besoin de comparer des valeurs. Par exemple pour $i == 3$, il va falloir comparer i à 3.

Pour comparer deux nombres, il suffit de les soustraire et ensuite de regarder les indicateurs. En effet, `Z` vaut 1 pour le calcul $i - 3$ quand i est égal à 3.

Le problème, c'est que faire une soustraction entre un registre et un nombre modifie le registre. Alors pour éviter ça, on utilise `CMP`. Cette instruction soustrait ses opérandes sans les modifier (le premier moins le second).

👉 Assemblez ce programme.

```
        IN  R0, 10      ; lire le premier nombre
        IN  R1, 10      ; lire le second nombre
        CMP R0, R1      ; compare les nombres sans changer les registres
        HLT
```

👉 Lancez le programme (`Reset` puis `RUN`). Saisissez quelques unes des valeurs essayées pour la soustraction (20 puis 10, 10 puis 10, etc.) et constatez les mêmes changements dans les indicateurs, mais sans altération de `R0`.

2. Instructions de branchement

Il s'agit d'un ensemble d'instructions qui affectent `IP` en cours d'exécution. On va commencer avec un programme stupide pour voir ce qui se passe.

👉 Assemblez ce programme.

```
        LD  R0, %00000001   ; initialisation de R0
repeter:
        OUT R0, 8           ; écrire R0, bit 1 => pixel noir
        ROL R0              ; faire tourner le 1 dans l'écriture binaire
        BRA repeter         ; retourner à repeter
        HLT
```

👉 Mettez le DSKY en mode 8. Dans ce mode, le port 8 permet d'allumer des points sur l'écran LCD (simulé). On écrit un octet et ça affiche des points là où il y a des bits à 1, dans la colonne de droite, en décalant toutes les colonnes vers la droite.

👉 Lancez le programme (`Reset` puis `RUN`). Il faudra sûrement appuyer sur `STOP` quand vous en aurez assez.

Les points descendent parce que `R0` écrit en binaire contient seulement un 1, et que l'instruction `ROL` déplace ce 1 vers la gauche. Sur l'écran, ça se traduit par un pixel de plus en plus bas. Il repart du haut parce que l'instruction `ROL` fait une boucle, voir la figure 1. Vous pouvez remplacer l'instruction `ROL` par `ROR` pour changer le comportement.

Figure 1: Rotations

👉 Réinitialisez le processeur (`Reset`) puis appuyez sur `NEXT` un certain nombre de fois pour voir l'exécution instruction par instruction. Faites notamment très attention à ce qui se passe pour `BRA`.

L'instruction `BRA` (*branch always*) affecte `IP` avec l'adresse de son opérande, celle du label `repeter`. C'est à dire qu'en arrivant sur cette instruction, le programme repart de l'adresse `repeter`. Donc il ne finit jamais. On appelle ça une « boucle infinie », et ça n'est pas très intéressant. On verra les boucles qui se finissent dans le TD8.

👉 Remplacez l'instruction `BRA` par `BCC` qui signifie *branch if carry is clear*, c'est à dire *modifier `IP` si l'indicateur `C` est à 0*. Donc ne pas sauter, continuer plus bas si `C` vaut 1.

Cette fois le programme s'arrête parce qu'à un moment, l'indicateur `C` passe à 1. Alors attention : la boucle continue tant que `C` est à zéro et elle s'arrête, parce qu'il n'y a pas de branchement, quand `C` vaut 1. Relisez bien la signification de l'instruction `BCC`.

3. Mise au point d'un programme (optionnel)

On va voir une technique très utile pour surveiller un programme : placer un point d'arrêt et exécuter pas à pas certaines parties.

Dans le panneau *Contexte d'exécution*, il y a un bloc *Instructions*, et une colonne avec une main contenant des cases à cocher. C'est pour placer un « point d'arrêt », *break point* en anglais. Ce sont des feux rouges pour arrêter l'exécution sous certaines conditions.

👉 Cochez la case à l'adresse `04`, celle du `ROL`. Un dialogue apparaît, cliquez `Enregistrer` en bas à gauche. La case est maintenant cochée dans le listing.

👉 Lancez le programme (`Rst+RUN`). Au lieu de finir tout son travail, l'exécution s'arrête devant le `ROL`.

👉 Regardez l'état de l'indicateur `C` et de `R0` puis cliquez sur `NEXT`. Ok, `C` est resté à 0. Cliquez sur `NEXT` (sans faire `Reset`) pour continuer l'exécution à partir de là. Le `BCC` fait revenir en `02`. Maintenant, cliquez sur `NEXT` en surveillant `C` jusqu'à ce que `C` passe à 1 à cause du `ROL`.

👉 Quand `C` vient juste de passer à 1, alors appuyez sur `NEXT` pour voir que le `BCC` ne modifie pas `IP`, on ne retourne pas en `02`, on arrive donc au `HLT`.

Pour aller plus vite au moment intéressant, on peut rajouter une condition sur le point d'arrêt. On peut lui dire de n'arrêter l'exécution en `04` que si `R0` vaut `$80` (en hexa).

👉 Cliquez sur la case cochée du point d'arrêt en `04`, saisissez `WHEN R0=$80` dans le champ *Condition*.

👉 Lancez le programme de zéro : `Rst+RUN`. L'exécution va à pleine vitesse, jusqu'au point d'arrêt quand `R0` est devenu égal à `80` en hexa. Continuez en appuyant sur `NEXT` jusqu'à la fin du programme.

4. Alternatives

4.1. Schémas de traduction algorithme vers instructions

Le plus important à comprendre, c'est comment on programme une alternative. Voici les deux schémas de traduction :

- **si (condition) alors { instructions1... } finsi**

```
        ...

si:     CMP R0, valeur  ; évaluation de la condition
        Bxx finsi       ; saut si la condition est FAUSSE
alors:  ; instructions1, effectuées si la condition est vraie
        ...
finsi:

        ...             ; suite du programme
```

Le principe est : si la condition est fausse, alors on passe par dessus les instructions du *alors*. Et si la condition est vraie, alors on continue dans le *alors*.

Il faut donc qu'il y ait un saut au finsi quand la condition est fausse. C'est le rôle de l'instruction `Bxx`, voir plus bas.

- **si (condition) alors { instructions1... } sinon { instructions2... } finsi**

```
        ...

si:     CMP R0, valeur  ; évaluation de la condition
        Bxx sinon       ; saut si la condition est FAUSSE
alors:  ; instructions1, effectuées si la condition est vraie
        ...
```

```
        BRA finsi       ; ne pas faire le sinon
sinon:  ; instructions2, effectuées si la condition est fausse
        ...
finsi:

        ...             ; suite du programme
```

Le principe est : si la condition est fausse, alors on passe par dessus les instructions du *alors* et on va aux instructions du *sinon*. Si la condition est vraie, on continue par les instructions du *alors* mais ensuite, on saute par dessus celles du *sinon*.

C'est vraiment important de comprendre ces deux schémas de traduction.

Dans les deux schémas, il faut choisir l'instruction de branchement `Bxx` correspondant à la condition : il doit y avoir un saut quand la condition est fausse.

4.2. Construction de la condition

Soit une condition (*var op val*), *var* étant une variable, *op* étant un opérateur de comparaison comme $<$, $\leq$, $=$, ... et *val* une valeur ou une autre variable. Exemples : ($i \leq 4$), ($i \neq j$)...

Le principe est de :

1. Placer la variable dans un registre, par exemple `LD R0, [variable]` ou `IN R0, port`
2. Comparer le registre à la valeur *val* par `CMP`, par exemple `CMP R0, valeur`
3. Choisir l'instruction de branchement selon *op* avec le tableau suivant :

op	Saut si	Branchement	Signification
$<$	$\geq$	BGE	Branch if greater or equal
$\leq$	$>$	BGT	Branch if greater than
$=$	$\neq$	BNE	Branch if not equal
$\neq$	$=$	BEQ	Branch if equal
$\geq$	$<$	BLT	Branch if less than
$>$	$\leq$	BLE	Branch if less or equal

NB: la colonne « *saut si* » est la négation (l'opposée) de la colonne *op*.
NB: en langage C, les opérateurs s'écrivent `<`, `<=`, `==`, `!=`, `>=` et `>`.

Exemples :

- si la condition s'écrit ($R0 \geq 20$), alors on programme `CMP R0,20` suivi de `BLT finsi`
- si la condition s'écrit ($R1 < -1$), alors on programme `CMP R1,-1` suivi de `BGE finsi`
- si la condition s'écrit ($R0 = 0$), alors on programme `CMP R0,0` suivi de `BNE finsi`.

Vous devez constater que c'est extrêmement mécanique. Il y a très peu à réfléchir.

NB: ce tableau concerne les comparaisons entre entiers signés, en convention $C2^8$. Dans le TD8, il y aura le tableau pour les comparaisons d'entiers non signés.

4.3. Application : valeur absolue d'un nombre

👉 Assemblez et testez ce programme.

```
        IN  R0, 10      ; lire un entier quelconque
si:     CMP R0, 0       ; comparer R0 à 0
        BGE finsi       ; saut si R0 >= 0, donc faire le "alors" si R0 < 0

alors:  NEG R0          ; changer le signe de R0

finsi:  OUT R0, 10      ; afficher le nombre
        HLT
```

L'instruction `BGE` saute au *finsi* si la comparaison montre que `R0`≥ 0. Vérifiez avec le tableau. Comprenez bien comment on écrit la comparaison et quelle instruction de branchement on choisit.

NB: cela ne marche pas bien pour -128 car $+128$ ne peut pas être codé en $C2^8$.

👉 Après avoir bien compris ce programme, écrivez son équivalent en pseudo-code.

4.4. Application : le plus grand de deux nombres

👉 Assemblez et testez ce programme.

```
        IN  R0, 10      ; lire un entier
        IN  R1, 10      ; lire un second entier
si:     CMP R0, R1      ; comparer R0 à R1
        BLT sinon       ; saut si R0 < R1

alors:  OUT R0, 10      ; afficher R0
        BRA finsi

sinon:  OUT R1, 10      ; afficher R1

finsi:
        HLT
```

N'oubliez pas de tester avec des valeurs négatives (codées en $C2^8$) et des mélanges.

👉 Après avoir bien compris ce programme, écrivez son équivalent en pseudo-code.

4.5. Application : le plus petit de deux nombres

👉 Modifiez le précédent programme pour qu'il affiche le plus petit des deux nombres saisis. Il y a trois manières à essayer (une à la fois car elles s'excluent mutuellement) :

- mettre `CMP R1, R0` à la place de `CMP R0, R1`
- changer l'instruction de saut,
- garder l'instruction de saut, mais intervertir les instructions du *alors* et du *sinon*.

4.6. Application : signe d'un nombre, version 1

👉 Codez et testez l'algorithme suivant

```
lire un nombre dans R0
si (R0 < 0) {
    mettre -1 dans R1
    afficher R1
} sinon {
    mettre +1 dans R1
    afficher R1
}
```

4.7. Application : signe d'un nombre, version 2

👉 Codez et testez l'algorithme suivant

```
lire un nombre dans R0
si (R0 < 0) {
    mettre -1 dans R1
    afficher R1
} sinon {
    si (R0 > 0) {
        mettre +1 dans R1
        afficher R1
    } sinon {
        mettre 0 dans R1
        afficher R1
    }
}
```

Il y a deux alternatives, l'une imbriquée dans l'autre. Il faut alors numéroter les labels : `si1`, `alors1`, `sinon1`, `finsi1` et `si2`, `alors2`, `sinon2`, `finsi2`.

Optionnel on peut optimiser ce programme de deux manières :

- Comme chaque alternative se termine par l'affichage de `R1`, on peut ne mettre qu'un seul affichage en tout, après les `finsi`.
- Comme les instructions de saut ne modifient pas les indicateurs VCNZ, il est inutile de refaire un `CMP R0, 0` dans le `si2` car celui du `si1` suffit. Ce ne serait plus vrai s'il y avait d'autres calculs avant la comparaison.

Ne tentez aucune optimisation si vous ne comprenez pas parfaitement, car votre programme pourrait devenir faux.

Vous savez en faire autant que `gcc`. Il vous reste seulement à voir comment on fait avec des conditions composées avec des *et* et des *ou*, dans le prochain TD.