Liste 1 : saisies, affichages et alternatives pour entiers signés (1 séance)

Notions à considérer pour pouvoir faire les exercices

Instructions utiles :

- LEA reg, id copie dans le registre général *reg* l'adresse de la variable ou de la constante nommée *id.* Pour obtenir l'adresse d'une variable ou d'une constante **locale**, on doit utiliser LEA (exemple : lea eax, i). Par contre, pour obtenir l'adresse d'une variable ou d'une constante **globale**, on peut utiliser soit LEA, soit la directive OFFSET (exemple : mov eax, OFFSET i).

- CMP opd, ops compare *opd* avec *ops*. CMP fonctionne comme SUB à ceci près que *opd* n'est pas modifié. Le but de la comparaison est uniquement de modifier certains indicateurs. Les 2 valeurs à comparer avec CMP sont des entiers.

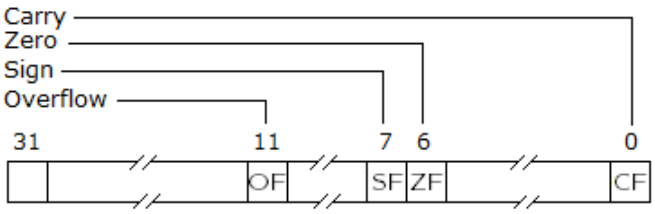
- JMP id effectue un branchement vers l'étiquette *id*.

- Après la réalisation d'une comparaison entre *opd* et *ops* avec CMP opd, ops, voici les instructions de branchement conditionnel JCC pour entiers **signés** pouvant être utilisées :

Condition	*JCC*	*Branchement si*	*Signification*
opd = ops	JE	ZF = 1	Jump if Equal
opd <> ops	JNE	ZF = 0	Jmp if Not Equal
opd > ops	JG ou JNLE	SF = OF et ZF = 0	Jump if Greater ou Jump if Not Less nor Equal
opd < ops	JL ou JNGE	SF <> OF	Jump if Less ou Jump if Not Greater nor Equal
opd >= ops	JGE ou JNL	SF = OF	Jump if Greater or Equal ou Jump if Not Less
opd <= ops	JLE ou JNG	SF <> OF ou ZF = 1	Jump if Less or Equal ou Jump if Not Greater

Le registre des indicateurs :

Le registre des indicateurs est un registre spécial de 32 bits qui regroupe de nombreux indicateurs dont les suivants : CF, ZF, OF et SF. Chacun de ces indicateurs occupe 1 bit et se trouve à une position spécifique :



Dans un processeur, la comparaison entraîne une soustraction et, donc, elle fait intervenir l'UAL. Prenons l'indicateur **ZF**. L'UAL fixe ZF à 1 quand le résultat de l'opération arithmétique ou logique qu'elle vient de réaliser est égal à zéro et elle fixe ZF à 0 dans le cas contraire.

Exemple :

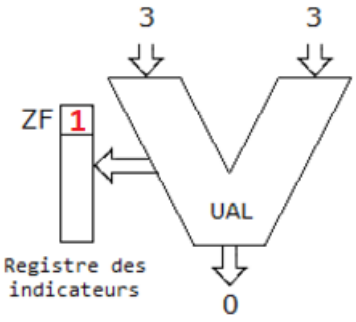
```
int i = 3;
...

if(i == 3)
      i = i + 2;
else
      i = i - 2;
```

Voici le code équivalent en assembleur :

```
int i = 3;
...

    cmp  i, 3
    je   contenuif   // si i == 3, alors ZF a la valeur 1 et branchement vers contenuif
    jne  contenuelse // si i != 3, alors ZF a la valeur 0 et branchement vers contenuelse

contenuif:
    add  i, 2
    jmp  fin         // jmp vers fin pour éviter d'exécuter sub i, 1

contenuelse:
    sub  i, 2

fin:
```

Le registre des indicateurs est ce qui lie une comparaison avec un branchement conditionnel :

1. La comparaison CMP i, 3 est réalisée par l'UAL et le résultat est 0 (3 – 3 = 0). Ce résultat n'est pas stocké dans la variable i, mais l'UAL place dans ZF la valeur 1.



2. L'instruction JE provoque un branchement uniquement quand ZF contient la valeur 1. Dans cet exemple, *JE contenuif* provoque donc un branchement vers l'étiquette *contenuif* dans le programme.

Construction en assembleur d'une alternative if... else :

Exemple :

```
int i = 8;

if(i > 4)
	printf("contenu du if");
else
	printf("contenu du else");
```

Etapes de conversion vers les instructions en assembleur :

1. Les 2 éléments i et 4 de l'expression booléenne (i > 4) deviennent les 2 opérandes de l'instruction CMP.

```
	CMP i, 4
```

2. Après CMP i, 4, on ajoute l'instruction JCC qui correspond à la fois à l'opérateur de comparaison utilisé en C et à la nature signée ou non signée des valeurs qui interviennent dans l'expression booléenne. Dans cet exemple, l'opérateur est > et la nature des 2 valeurs est signée. Ceci donne JG contenuif dont le but est d'effectuer un branchement vers l'intérieur du if uniquement quand l'expression booléenne est vraie.

```
	CMP i, 4
	JG  contenuif

contenuif:
```

3. Après JG contenuif, on ajoute l'instruction JCC opposée à JG qui est JNG contenuelse, de sorte à effectuer un branchement vers l'étiquette contenuelse uniquement quand l'expression booléenne est fausse.

```
	CMP i, 4
	JG  contenuif
	JNG contenuelse

contenuif:
	// contenu du if

contenuelse:
	// contenu du else
```

4. Pour éviter d'entrer automatiquement dans le contenu du else lorsque se termine l'exécution du contenu du if, on ajoute à la fin de celui-ci l'instruction JMP finif.

```
        CMP i, 4
        JG  contenuif
        JNG contenuelse

contenuif:
        // contenu du if

        jmp finif

contenuelse:
        // contenu du else

finif:
```

5. On ajoute les instructions à l'intérieur du contenu du if et du contenu du else. Le résultat final est le suivant :

```
#include <stdio.h>

int i = 8;
const char msgif[] = "contenu du if";
const char msgelse[] = "contenu du else";

void main()
{
      _asm
      {
            CMP  i, 4
            JG   contenuif
            JNG  contenuelse

      contenuif:
            PUSH OFFSET msgif
            CALL DWORD PTR printf
            ADD  esp, 4

            jmp  finif

      contenuelse:
            PUSH OFFSET msgelse
            CALL DWORD PTR printf
            ADD  esp, 4

      finif:
      }
}
```

6. On peut optimiser le programme ainsi :

```
#include <stdio.h>

int i = 8;
const char msgif[] = "contenu du if";
const char msgelse[] = "contenu du else";

void main()
{
	_asm
	{
		CMP  i, 4
		JNG  contenuelse

		PUSH OFFSET msgif
		CALL DWORD PTR printf
		ADD  esp, 4

		jmp  finif

	contenuelse:
		PUSH OFFSET msgelse
		CALL DWORD PTR printf
		ADD  esp, 4

	finif:
	}
}
```

On a enlevé l'instruction JG contenuif, car elle n'est pas indispensable. Ainsi, soit l'instruction JNG contenuelse effectue un branchement vers l'étiquette contenuelse, soit ce branchement n'a pas lieu et l'exécution du contenu du if commence alors.

Exemple : saisir 2 entiers signés et afficher d'abord le plus grand, puis le plus petit.

```
#include <stdio.h>

const char formatSaisie[] = "%d";
const char txtSaisie[] = "Entrer le 1er nombre : ";
const char txtSaisie1[] = "Entrer le 2e nombre : ";
const char formatAffichage[] = "Le nombre %d est plus grand que le nombre %d";
const char formatAffichage1[] = "Le nombre %d est plus grand ou egal au nombre %d";

void main()
{
	int i, i1;

	_asm
	{
		push offset txtSaisie
		call dword ptr printf
		add  esp, 4

		lea  eax, i
		push eax
		push offset formatSaisie
		call dword ptr scanf
		add  esp, 8
```

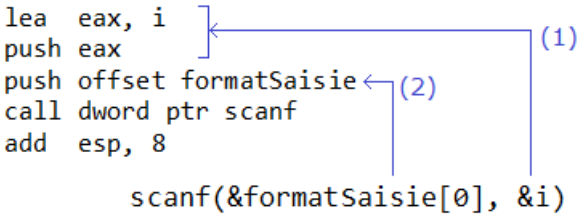
```
            push offset txtSaisie1
            call dword ptr printf
            add  esp, 4

            lea  eax, i1
            push eax
            push offset formatSaisie
            call dword ptr scanf
            add  esp, 8

            /* if (i < i1)
                   printf("Le nombre %d est plus grand que le nombre %d", i1, i);
               else
                   printf("Le nombre %d est plus grand ou egal au nombre %d", i, i1);
            */

            mov eax, i
            cmp eax, i1
            jl  contenuif                // instruction non nécessaire
            jnl contenuelse

      contenuif:
            push i
            push i1
            push offset formatAffichage
            call dword ptr printf
            add  esp, 12

            jmp  fin

      contenuelse:
            push i1
            push i
            push offset formatAffichage1
            call dword ptr printf
            add  esp, 12

      fin:
      }
}
```

Quelques commentaires :

- Un paramètre peut être passé à une fonction par la pile de 2 façons différentes : par adresse ou par valeur.

    - **Passage par adresse** : pour empiler l'adresse d'une variable ou d'une constante **globale**, on peut utiliser directement PUSH OFFSET. Par contre, pour empiler l'adresse d'une variable ou d'une constante **locale**, on doit tout d'abord récupérer son adresse dans un registre général avec LEA, puis on empile le contenu de ce registre avec PUSH.

Exemple :



Lors de cet appel de fonction scanf(&formatSaisie[0], &i), on passe 2 adresses en paramètre : l'adresse de la variable locale i qui est la variable dans laquelle doit être stocké le nombre, et l'adresse de la constante globale formatSaisie qui est une chaîne dans laquelle se trouve le spécificateur de format %d. Le passage des paramètres à une fonction se fait dans l'ordre contraire à celui en C d'écriture des paramètres entre parenthèses.

- **Passage par valeur** : pour empiler la valeur d'une variable de 4 octets, d'une constante de 4 octets ou d'un registre, on utilise PUSH (exemples : push eax, push i, ...). Pour empiler une valeur immédiate entière, on utilise PUSH DWORD PTR, où la directive DWORD PTR indique que la valeur immédiate est codée sur 4 octets (exemple : push dword ptr 25).

- Une chaîne de caractères qu'on écrit directement dans les parenthèses d'appel à une fonction en C est tout d'abord déclarée en tant que constante par le compilateur, puis c'est l'adresse de cette constante qui est transmise en paramètre à la fonction.

  Par exemple, les 3 codes suivants sont équivalents :

```c
#include <stdio.h>

void main()
{
    printf("Bonjour");
}
```

```c
#include <stdio.h>

const char msg[] = "Bonjour";

void main()
{
    printf(&msg[0]);
}
```

```c
#include <stdio.h>

const char msg[] = "Bonjour";

void main()
{
    _asm
    {
        push offset msg
        call dword ptr printf
        add  esp, 4
    }
}
```

- Les n octets occupés pas les paramètres doivent être libérés de la pile après l'exécution d'une fonction. Le registre ESP est le pointeur de pile. Avec ADD ESP, n, on augmente l'adresse se trouvant dans ESP de n pour libérer n octets dans la pile.

Priorité des opérateurs du langage C :

Classes de priorité	*Opérateurs en C*
++++++++++	`~, -, !, *, &, (casting)`
+++++++++	`*, /, %`
++++++++	`+, -`
+++++++	`<<, >>`
++++++	`<, >, <=, >=`
+++++	`==, !=`
++++	`&`
+++	`^`
++	`|`
+	`=`

Quelques remarques :

- Les opérateurs de comparaison sont plus prioritaires que les opérateurs logiques, mais moins prioritaires que les opérateurs de décalages et les opérateurs arithmétiques.
- Tous les opérateurs de comparaison n'ont pas la même priorité.
- L'opérateur unaire ! inverse la valeur d'une expression booléenne. Il est très différent de l'opérateur unaire logique ~ qui inverse tous les bits d'une valeur.

  Exemple :

Langage C	*Assembleur*

`int i = 2;`
 	
`void main()`
`{`
`  if (!(i > 3))`
`  i = 100;`
`}`
	`int i = 2;`
	
	`void main()`
	`{`
	`  _asm`
	`  {`
	`  cmp i, 3`
	`  jg  fin`
	
	`  mov i, 100`
	
	`  fin:`
	`  }`
	`}`

Avec l'opérateur !, on a pris JG pour aller vers l'étiquette fin plutôt que JNG.

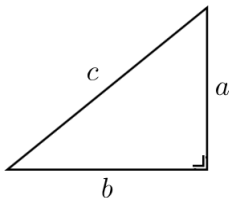
Exercices

1. Saisir une température en degrés Celsius et l'afficher en degrés Fahrenheit. La formule de conversion est F = C * 9 / 5 + 32.

2. Saisir la distance (en kilomètres) et la durée d'un trajet (en heures et minutes). Afficher la vitesse moyenne du trajet (par exemple, pour un trajet de 200 km effectué en 2 heures et 30 minutes, on a une vitesse moyenne de 80 km/h).

3. Saisir la taille des 3 côtés d'un triangle. Afficher si ce triangle est équilatéral (= les 3 côtes ont la même taille) ou non.

4. Saisir la taille pour les côtés a, b et c d'un triangle et afficher si ce triangle est rectangle ou non, c'est-à-dire s'il satisfait le théorème de Pythagore ou non (par exemple, quand a = 3, b = 4 et c = 5, alors le triangle est rectangle).



   Théorème de Pythagore: dans un triangle rectangle, le carré de la longueur de l'hypoténuse est égal à la somme des carrés des longueurs des côtés de l'angle droit.

Toutes les valeurs utilisées dans les saisies, affichages et alternatives sont des entiers signés. Dans les exercices 1 et 2, pour faire les calculs, vous pouvez temporairement passer par des flottants.

Attention : les fonctions des librairies telles que printf, scanf, etc, modifient le contenu de certains registres.