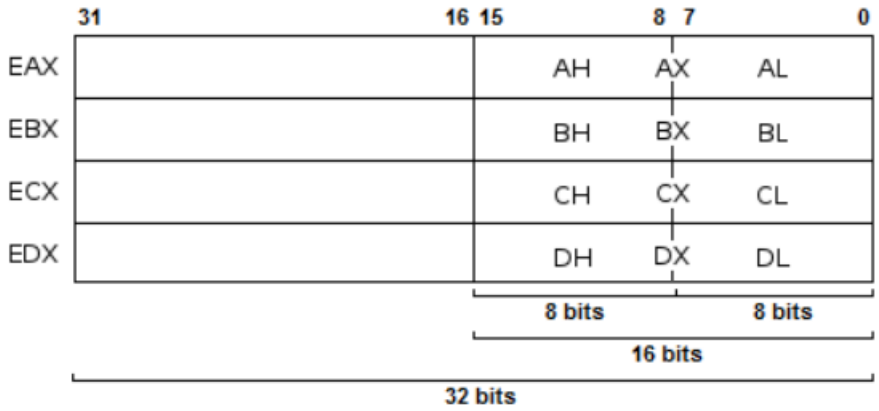
Liste d'exercices 3 : les nombres entiers signés et non signés

Notions à considérer pour pouvoir faire les exercices

La matière du cours se trouve dans le chapitre 3 dans les notes de cours.

Relire attentivement les notions à considérer présentées dans la liste d'exercices 2.

Organisation des 4 premiers registres généraux:



Types de données:

- Types de données pour les nombres entiers **signés**:

Types	*Tailles*	*Plages de valeurs*
`char`	8 bits	$[-2^7, ..., +2^7-1]$
`short`	16 bits	$[-2^{15}, ..., +2^{15}-1]$
`int`	32 bits	$[-2^{31}, ..., +2^{31}-1]$

$-2^7 = -128$; $+2^7-1 = +127$; $-2^{15} = -32768$; $+2^{15}-1 = +32767$; $-2^{31} = -2147483648$; $+2^{31}-1 = +2147483647$.

- Types de données pour les nombres entiers **non signés**:

Types	*Tailles*	*Plages de valeurs*
`unsigned char`	8 bits	$[0, ..., 2^8-1]$
`unsigned short`	16 bits	$[0, ..., 2^{16}-1]$
`unsigned int`	32 bits	$[0, ..., 2^{32}-1]$

$2^8-1 = 255$; $2^{16}-1 = 65535$; $2^{32}-1 = 4294967295$.

Priorité des opérateurs:

Priorités	*Opérateurs en C*	*Instructions en assembleur*
+++++	`()`	
++++	`~, -, (changement genre)`	`not, neg, movzx→movsx/movsx→movzx`
+++	`*, /, %`	`imul/mul, idiv/div, idiv/div`
++	`+, -`	`add, sub`
+	`=`	`mov`

Les opérateurs unaires sont plus prioritaires que les opérateurs binaires.

Instructions pour nombres entiers signés:

- *movsx opd, ops* : elle étend le nombre entier **signé** de l'opérande *ops* à la taille de l'opérande *opd*, puis elle copie ce nombre étendu dans *opd*. Concrètement, le processeur propage la valeur du bit de signe du nombre se trouvant dans *ops* vers tous les bits du/des octets ajoutés devant le nombre à étendre.

- *imul opd, ops* réalise opd = opd * ops. Les nombres à multiplier sont des nombres entiers **signés**.

- *cdq* : elle étend le nombre entier signé de EAX à la paire EDX, EAX. Elle précède toujours l'instruction idiv, car elle sert à préparer le dividende.

- *idiv op* : cette instruction sert à diviser un nombre entier **signé** par un autre. Le nombre entier signé de 64 bits formé par la concaténation des registres EDX et EAX est divisé par un nombre entier signé de 32 bits figurant dans l'opérande *op*. Après l'opération, EAX stocke le quotient et EDX stocke le reste de la division.

  Exemple:

```
int  i, j;
```

Langage C	*Assembleur*

`i = i / j;`	`mov  eax, i`
	`cdq`
	`idiv j`
	`mov  i, eax`

On a préparé dans edx, eax le dividende sur 64 bits. En **signé**, l'extension se fait avec l'instruction cdq. Puis on a divisé ce dividende par la valeur de la variable j. Enfin, on a copié dans la variable i le quotient figurant dans le registre eax.

- *neg op :* elle copie dans l'opérande *op* le complément à 2 du nombre se trouvant initialement dans *op*. Ceci a pour effet de changer le signe de la valeur se trouvant dans *op.*

  Exemple :

```
char b = -3;
int  i;
```

Langage C	*Assembleur*

`i = -b;`	`movsx eax, b`
	`neg  eax`
	`mov  i, eax`

  La valeur -3 de la variable b est étendue à 32 bits. Ensuite, le complémente à 2 de cette valeur est obtenu et copié dans la variable i.

Instructions pour nombres entiers non signés:

- *movzx opd, ops* : elle étend le nombre entier **non signé** de l'opérande *ops* à la taille de l'opérande *opd*, puis elle copie ce nombre étendu dans *opd*. Concrètement, le processeur propage la valeur 0 vers tous les bits du/des octets ajoutés devant le nombre à étendre.

- *div op* : cette instruction sert à diviser un nombre entier **non signé** par un autre. Le nombre entier non signé de 64 bits formé par la concaténation des registres EDX et EAX est divisé par un nombre entier non signé de 32 bits figurant dans l'opérande *op*. Après l'opération, EAX stocke le quotient et EDX stocke le reste de la division.

  Exemple :

```
unsigned int i;
unsigned int j;
```

Langage C	*Assembleur*

`i = i / j;`	`mov  eax, i`
	`mov  edx, 0`
	`div  j`
	`mov  i, eax`

  On a préparé dans edx, eax le dividende sur 64 bits. En non signé, l'extension se fait avec l'instruction mov edx, 0. Puis on a divisé ce dividende par la valeur de la variable j. Enfin, on a copié dans la variable i le quotient figurant dans le registre eax.

- *mul op* multiplie 2 nombres entiers **non signés**. Le nombre de 32 bits stocké dans EAX est multiplié par le nombre de 32 bits figurant dans l'opérande *op.* Le résultat est un nombre de 64 bits stocké dans la paire EDX, EAX.

  Exemple :

```
unsigned int i;
unsigned int j;
```

Langage C	*Assembleur*

`i = i * j;`	`mov  eax, i`
	`mul  j`
	`mov  i, eax`

Quand un nombre entier signé intervient avec un nombre entier non signé dans une multiplication ou dans une division, l'instruction pour nombres non signés est choisie.

Exemple:

```
unsigned int i;
int j;
```

Langage C	*Assembleur*

`i = i / j;`	`mov eax, i`
	`mov edx, 0`
	`div j`
	`mov i, eax`

Opérateur logique ~: cet opérateur unaire ~ inverse tous les bits d'un nombre. Il correspond à l'instruction *not op*.

Exemple :

```
unsigned char b = 255;
int i;
```

Langage C	*Assembleur*

`i = ~b;`	`movzx eax, b`
	`not   eax`
	`mov   i, eax`

La valeur 255 de la variable b est étendue à 32 bits. Ensuite, tous les bits de cette valeur sont inversés et copiés dans la variable i.

Changements explicites du genre:

- Du signé vers le non signé:

  Exemple:

```
unsigned int i;
char b;
```

Langage C	*Assembleur*
`i = (unsigned char)b;`	`movzx  eax, b` `mov    i, eax`

  Comme la valeur de b est temporairement non signée, l'instruction MOVZX est utilisée.

- Du non signé vers le signé:

  Exemple:

```
unsigned char b;
int i;
```

Langage C	*Assembleur*
`i = (char)b;`	`movsx  eax, b` `mov    i, eax`

  Comme la valeur de b est temporairement signée, l'instruction MOVSX est utilisée.

Le changement du genre a un impact au niveau du choix de l'instruction de multiplication ou de division.

Exemple:

```
int  i, j;
```

Langage C	*Assembleur*

`i = (unsigned int)i / (unsigned int)j;`	`mov eax, i`
	`mov edx, 0`
	`div j`
	`mov i, eax`

Les variables i et j sont signées, mais temporairement on les considère non signées. Ceci entraîne l'usage des instructions mov edx, 0 et div à la place de cdq et idiv.

Exercices

Exercice 1: apprendre à distinguer le fonctionnement des instructions pour nombres non signés du fonctionnement des instructions pour nombres signés.

Pour chaque groupe d'instructions en assembleur suivant, observez la valeur initiale placée dans la variable b (dans la variable i pour les derniers exercices), puis, observez et justifiez la valeur prise par la variable i en décimal et en hexadécimal après l'exécution dans le débogueur.

- ```
  char b = 0xff;
  int  i;
  ```

Langage C	*Assembleur*
`i = b;`	`movsx eax, b` `mov  i, eax`

- ```
  char b = 0x7f;
  int  i;
  ```

Langage C	*Assembleur*
`i = b;`	`movsx eax, b` `mov  i, eax`

- ```
  unsigned char b = 0xff;
  unsigned int i;
  ```

Langage C	*Assembleur*
`i = b;`	`movzx eax, b` `mov  i, eax`

- ```
  int i = 0xfffffffc;
  ```

Langage C	*Assembleur*

`i = i * 2;`	`mov  eax, i`
	`imul eax, 2`
	`mov  i, eax`

- ```
  unsigned int i = 0x7ffffffc;
  ```

Langage C	*Assembleur*

`i = i * 2;`	`mov eax, i`
	`mov ebx, 2`
	`mul ebx`
	`mov i, eax`

  ```

- `int i = 0xfffffff7;`

Langage C	*Assembleur*

`i = i / 2;`	`mov eax, i`
	`cdq`
	`mov ebx, 2`
	`idiv ebx`
	`mov i, eax`

- `unsigned int i = 0xfffffff7;`

Langage C	*Assembleur*

`i = i / 2;`	`mov eax, i`
	`mov edx, 0`
	`mov ebx, 2`
	`div ebx`
	`mov i, eax`

Exercice 2: écrivez la séquence d'instructions en assembleur qui correspond à chaque instruction d'affectation en langage C suivante (en commentaire en vert figure la valeur à obtenir au final).

```
int   i = -3;
short s = 4;
char  b = -2;
unsigned int   i1 = 8;
unsigned short s1 = 4;
unsigned char  b1 = 5;
```

- `i = -s1 + 2;` `// i = -2`
- `i = ~s + 2;` `// i = -3`
- `i = -(b + (char)b1 + s);` `// i = -7`
- `i = -(int)((i1 + -b) / b1);` `// i = -2`
- `i = (i1 * b1) / -(2 * b);` `// i = 10`
- `s1 = -i - 3 + ((unsigned char)b + (short)s1);` `// s1 = 258`
- `s1 = (i * 6 - s1) + 3 - (0xff + s + b1);` `// s1 = 65253`
- `i = -(int)(i1 % b1 * (unsigned char)b);` `// i = -762`

Remarque: ici, écrire `(int)` dans certaines instructions d'affectation en C n'entraîne aucune instruction en assembleur. Ceci évite simplement d'avoir un message d'erreur du compilateur C.