TD d'architecture des ordinateurs XV
Assembleur

Exercice 1 : Remarques préliminaires

1.1 Utilisation de l'accumulateur A

Le registre A du Z80 joue le rôle d'accumulateur. Il est donc plutôt utilisé pour effectuer des calculs que pour conserver des données permanentes. En ce sens, la plupart des instructions arithmétiques et d'acces à la mémoire le privilégient. Par exemple, pour pouvoir réaliser l'addition de deux valeurs stockées dans deux registres B et C, il faut copier la valeur de B dans A puis ajouter à A la valeur de C :

```
LD  A,B
ADD A,C
```

L'accumulateur contient alors le résultat.

Il en est de même pour charger une valeur stockée en mémoire dans un registre de travail (ici D) :

```
LD A,(100h)
LD D,A
```

1.2 Modes d'adressage

Pour les questions de ce TD, nous n'utiliserons que les modes d'adressage suivants :

immédiat :

L'adressage immédiat permet de manipuler des valeurs constantes comme par exemple pour charger la valeur 0 dans le registre B :

```
LD B,0
```

étendu :

L'adressage étendu permet de charger ou de sauver une valeur en mémoire. Par exemple, pour charger la valeur stockée en mémoire à l'adresse 100h dans le registre A, on écrit :

```
LD A,(100h)
```

alors que pour écrire la valeur contenue dans le registre A à l'adresse 101h, on écrit :

```
LD (101h),A
```

indirect par registre :

L'adressage indirect permet de charger ou de sauver une valeur en mémoire à l'adresse 16 bits contenue dans un registre. Par exemple, l'instruction suivante charge dans l'accumulateur la valeur dont l'adresse mémoire est contenue dans le registre 16 bits HL.

```
LD A,(HL)
```

Si avant l'exécution de cette instruction, le registre HL contient la valeur 102h et que la valeur à l'adresse 102h est 52, alors l'instruction charge la valeur 52 dans A.

Ce mode d'adressage est utile pour accéder successivement à plusieurs valeurs consécutives en mémoire.

1.3 Branchements et branchements conditionnels

Le jeu d'instruction du Z80 comprend de nombreuses instructions de branchement ; on se limite à l'utilisation de l'instruction `JR`.

Pour plus de clarté dans la rédaction des programmes assembleur, on utilise des étiquettes associées à une instruction. Les instructions de branchement qui permettent de sauter à une ligne du programme précisent simplement l'étiquette qui lui est associée. Par exemple :

```
BOUCLE: INC A
        JR  BOUCLE
```

L'exécution de l'instruction `JR BOUCLE` est suivie de l'instruction `INC A` associée à l'étiquette `BOUCLE`.

Le saut effectué par l'instruction `JR` peut être conditionnel. Par exemple, l'instruction :

```
CP B
JR Z,BOUCLE
```

effectue un saut à l'étiquette `BOUCLE` si l'indicateur Z est positionné, c'est-à-dire (pour cet exemple) lorsque les registres A et B sont égaux, car `CP B` effectue la comparaison entre B et A.

Alors que les instructions :

```
CP B
JR NC,BOUCLE
```

effectue un saut à l'étiquette `BOUCLE` si l'indicateur C n'est pas positionné, c'est-à-dire ici lorsque le registre B est inférieur ou égal au registre A.

Exercice 2 : Exercices

- Traduire en assembleur Z80 le fragment de programme ci-dessous.

```
int int_v1; //-- codé sur 8 bits à l'adresse 100h
int int_v2; //-- codé sur 8 bits à l'adresse 101h
int int_v3; //-- codé sur 8 bits à l'adresse 102h

int_v3 = int_v1 + int_v2;
```

- Reprendre la question précédente en faisant comme hypothèse que les variables variables V1, V2 et V3 sont codées sur 16 bits et stockées aux adresses 100h-101h, 102h-103h et 104h-105h. On utilisera les opérations

d'addition 8 bits et d'addition 8 bits avec retenue.

Remarque : la convention qui a été choisie sur le Z80 pour stocker en mémoire les valeurs 16 bits consiste à placer la partie de poids faible sur le premier octet de l'adresse et partie de poids fort sur le second. Ainsi, la valeur hexadécimale 1F0Ah sera codée à partir de l'adresse 100h de la manière suivante :

adresse	valeur
100h	0Ah
101h	1Fh

– Traduire ce nouveau fragment de code en assembleur. Les 5 éléments du tableau t sont codés sur 8 bits consécutivement à partir de l'adresse 100h.

```
int *int_t; //-- un tableau de 4 entier stocké à l'adresse 100h
int int_i;  //-- 8 bits

for(int_i = 0; int_i < 4; int_i++){
  int_t[int_i] = 0;
}
```

– Même question pour le fragment de code suivant.

```
int *int_t1; //-- un tableau de 4 entier stocké à l'adresse 100h
int *int_t2; //-- un tableau de 4 entier stocké à l'adresse 105h
int int_i;   //-- 8 bits

for (int_i = 0; int_i < 4; int_i++){
  int_t2[int_i] = int_t1[int_i];
}
```

– Même question pour le fragment de code suivant.

```
int *int_t1; //-- un tableau de 4 entier stocké à l'adresse 100h
int *int_t2; //-- un tableau de 4 entier stocké à l'adresse 105h
int *int_t3; //-- un tableau de 4 entier stocké à l'adresse 10Ah
int int_i;   //-- 8 bits

for (int_i = 0; int_i < 4; int_i++){
  int_t3[int_i] = int_t1[int_i] + int_t2[int_i];
}
```

– Ci dessous est présenté l'algorithme d'Euclide pour le calcul du PGCD. Il utilise deux variables B et C. Il est demandé de traduire ce programme

en utilisant les registres B et C du processeur Z80 en lieu et place des variables correspondantes de l'algorithme.

```
// algorithme d'Euclide
// de calcul du PGCD
int_b = 21;
int_c = 35;

while (int_b != int_c){
  if (int_b > int_c) {
    int_b = int_b - int_c;
  } else {
    int_c = int_c - int_b;
  }
}
```