1 Échauffement

Les satellites de l'ESA ont arrêté de communiquer avec la Terre! C'est à vous de comprendre pourquoi. Les ingénieurs de l'ESA vous ont donné le programme d'un satellite défaillant, ainsi qu'une description du CPU A.L.A.N dedans. Avant de commencer à le debugger, vous voulez vous échauffer un peu. Vous vous demandez combien de carburant les satellites défaillants ont utilisé pour arriver en orbite. Vous avez trouvé en ligne une liste des poids de chaque satellite (dans le fichier `poids.txt`) avec un entier par ligne.

Pour estimer la quantité de carburant nécessaire pour un satellite, divisez le poids par 3 en arrondissant vers le bas, puis soustrayez 2 (la quantité de carburant nécessaire est toujours positive). N'oubliez pas que le carburant fait partie du poids de la fusée, et a donc besoin de carburant lui-même!

(Note: chaque satellite a été lancé séparément)

Par exemple, un satellite qui pèse 1969 kg nécessite $\lfloor \frac{1969}{3} \rfloor - 2 = 654$ carburant. Les 654 kg de carburant nécessitent $\lfloor \frac{654}{3} \rfloor - 2 = 216$ kg de carburant. $\lfloor \frac{216}{3} \rfloor - 2 = 70$, $\lfloor \frac{70}{3} \rfloor - 2 = 21$, $\lfloor \frac{21}{3} \rfloor - 2 = 5$. Enfin, $\lfloor \frac{5}{3} \rfloor - 2 < 0$, donc on a tout le carburant nécessaire. Ainsi, le satellite a besoin de $654 + 216 + 70 + 21 + 5 = 966$ kg de carburant.

La fonction `probleme_echauffement` dans `template.py` prend en argument la liste des poids. Complétez-la de manière à retourner la somme des quantités de carburant nécessaires pour chaque satellite.

2 Explication du CPU

Le CPU du satellite fonctionne en exécutant des *instructions* prédéfinies qui opèrent sur des *registres*. Le système A.L.A.N utilise 4 registres différents, nommés `A`, `B`, `C` et `D`. Les registres contiennent des nombres de 32 bits.

Le satellite contient à son bord un programme, qui est simplement une liste d'instructions. Le CPU exécute chaque instruction une par une, dans l'ordre, et s'arrête quand il arrive à la fin.

Les instructions sont encodées par des nombres, divisés en blocs de 8 bits (1 octet). Il n'y a pas de délimitations entre les instructions; chaque instruction a une longueur prédéfinie, donc on peut toujours savoir où on est. Le premier octet de chaque instruction vous donne le type de l'instruction.

2.1 Instructions basiques

Pour commencer, vous allez implémenter les instructions d'addition, de soustraction, de multiplication, et de constante. Notez que chaque nombre ici est en hexadécimal! Un octet est ainsi exactement 2 caractères.

L'arithmétique est modulaire à 32 bits (ce qui veut dire que tout se calcule modulo 2^{32}).

- `add`: L'addition correspond à l'octet `01`. Cet octet est ensuite suivi de 3 paramètres, respectivement la destination et les deux arguments. Chaque paramètre est un registre. `00` correspond au registre `A`, `01` à `B`, `02` à `C`, et `03` à `D`. Ainsi, la séquence `01 01 02 01` correspond au code python `B = C + B`.
- `sub`: La soustraction correspond à l'octet `02`, et fonctionne exactement comme l'addition. Ainsi, `02 00 00 00` correspond au code python `A = A - A`. Comme l'arithmétique est modulaire, soustraire `1` à `0` et stocker le résultat dans `A` mettrait le nombre $2^{32} - 1$ dans `A`.
- `mul`: La multiplication correspond à l'octet `03`, et fonctionne exactement comme l'addition et la soustraction.

- `cst`: L'opération « charger une constante » correspond à l'octet `FF`, et prend 2 paramètres; un registre de destination, et un nombre de 32 bits (qui est donc divisé en 4 octets). Le troisième octet de l'instruction est l'octet du bas du nombre. Ainsi, `FF 00 FF 00 00 00` correspond au code python `A = 255`. `FF 02 00 00 00 01` correspond au code python `C = 16777216` ($16777216 = 2^{24}$)

Sur une instruction inconnue, le CPU doit s'arrêter.

Un programme qui n'utilise que ces instructions se trouve dans `programme_1.txt`. Chaque instruction est un octet hexadécimal. Exécutez-le et retournez la valeur de `A*B + C*D`.

Dans `template.py`, vous trouverez une endroit où mettre du code. Notez que chaque problème suivant ne sera qu'une extension sur le CPU, donc vous pouvez mettre le code du CPU dans un seul endroit.

2.2 Instructions de contrôle

Pour le moment, le CPU ne peut qu'exécuter les instructions linéairement. Il est impossible de former des boucles, des `if`, ou des fonctions. Pour cela, nous allons rajouter les instructions suivantes:

- `cmp`: Compare l'égalité de deux registres et met une valeur indicatrice dans un troisième. Cette valeur est 1 si les deux registres sont égaux et 0 sinon. L'instruction correspond à l'octet `10` et prend trois registres; respectivement le registre cible puis les deux paramètres. Ainsi `10 00 01 02` mets 1 dans `A` si `B == C`, et 0 sinon.
- `ltu`: Compare la valeur de deux registres et met une valeur indicatrice dans un troisième. Cette valeur est 1 si le premier registre est strictement plus petit que la deuxième, et 0 sinon. L'instruction correspond à l'octet `11` et prend trois registres; respectivement le registre cible puis les deux paramètres. Ainsi `10 00 01 02` mets 1 dans `A` si `B < C`, et 0 sinon.
- `jmp`: Saut inconditionnel à une instruction. L'instruction correspond à l'octet `20` et prend un paramètre, un registre. Ainsi, `20 00` saute à l'octet dont l'indice est contenu dans le registre `A`. Notez que la valeur est comptée en octets et non en instructions!
- `jz`: Saut conditionnel si un registre est zéro. L'instruction correspond à l'octet `21` et prend deux paramètres; un registre à comparer, et un registre contenant l'indice de saut.
- `jnz`: Saut conditionnel si un registre n'est pas zéro. Fonctionne exactement comme `jz` sauf que son octet est `22`.

Un programme qui n'utilise que ces instructions se trouve dans `programme_2.txt`. Chaque instruction est un octet hexadécimal. Exécutez-le et retournez la valeur de `A*B + C*D`.

2.3 Sous-routines

Enfin, pour rendre notre CPU plus facile à utiliser, nous allons rajouter des **sous-routines**. Elles sont similaires à des fonctions en Python, mais sans paramètres. Pour cela, nous rajoutons deux instructions:

- `call`: Comme `jmp`, mais rajoute l'adresse courante à une liste (la **pile d'appels**). Correspond à l'octet `30`.
- `ret`: Retourne à la dernière adresse à laquelle on a `call` (ou plutôt, l'instruction suivante), et l'enlève de la pile d'appels. Ne prend pas de paramètres et correspond à l'octet `31`. S'il n'y pas de valeur sur la pile, le CPU s'arrête.

Dans l'exemple suivant, nous allons utiliser les noms d'instructions et de registres, ainsi que des labels pour les sous-routines, plutôt que simplement les octets. Les labels sont remplacés par l'indice de l'octet de l'instruction suivante.

```
1   cst A, main
2   jmp A
3
4   bar:
5   add B, B, B
6   ret
7
8   foo:
9   cst A, bar
10  cst B, 256
11  call A
12  ret
13
14  main:
15  cst A, .foo
16  call A
17  add, A, B, B
```

Dans l'exemple précédent, on commence par *sauter* à `main`. Ensuite, on appelle `foo`. `foo` appelle `bar` avec `B=256`, et `bar` double la valeur de `B`, puis retourne. Ainsi, l'exécution est dans l'ordre suivant:

Ligne	Pile d'appels
1	Vide
2	Vide
15	Vide
16	Vide
9	[17]
10	[17]
11	[17]
5	[17, 12]
6	[17, 12]
12	[17]
17	Vide

et à la fin, le registre A contient 1024 et le registre B contient 512.

Notez qu'on utilise ici des lignes pour indexer mais qu'on utilise des indices d'octets en pratique.

Le fichier `programme_3.txt` contient un programme qui utilise des sous-routines. Exécutez-le et retournez la valeur de `A*B + C*D`.

2.4 Détection de cycles

Enfin, nous allons détecter si le CPU rentre dans une boucle infinie. Pour cela, on ne rajoute pas d'instructions. Il est possible de détecter une boucle infinie simplement en observant l'état du CPU au fil du temps. Vous pouvez soit chercher vous-mêmes comment faire, soit regarder l'indice qui se trouve à la fin de cette page[1].

Le fichier `programme_4.txt` contient un programme qui rentre dans une boucle infinie. Détectez la boucle infinie la plus courte et retournez sa longueur.

[1] Indice: comme le CPU est déterministe, une instruction avec les mêmes valeurs de registres fait toujours la même chose. Ainsi, si on revient sur le même état deux fois, le CPU continuera de la même manière, et refinira dans ce même état à l'infini.