TP ASSEMBLEUR

Énoncé général

On utilisera le **simulateur Web** accessible par le lien suivant : https://www.peterhigginson.co.uk/RISC/

Pour les exercices de ce TP, on se donne les **12 opérations du langage assembleur** suivantes. On n'utilisera **pas d'autres opérations**, pas mêmes celles proposées par le simulateur.

Catégorie	Opération	Code opération	Exemple	Description de l'exemple
Déplacement	Charger	`LDR`	`LDR R1,78`	Place la valeur stockée à l'adresse mémoire 78 dans le registre R1
	Stocker	`STR`	`STR R3,125`	Place la valeur stockée dans le registre R3 en mémoire à l'adresse 125
	Déplacer	`MOV`	`MOV R1,#23`	Place la valeur constante 23 dans R0
Arithmétique	Ajouter	`ADD`	`ADD R1,R0,#128`	Additionne 128 à la valeur du registre R0, place le résultat dans R1
			`ADD R0,R1,R2`	Additionne la valeur du registre R1 à la valeur du registre R2, place le résultat dans R0
	Soustraire	`SUB`	`SUB R0,R1,R2`	Soustrait la valeur du registre R2 de la valeur du registre R1, place le résultat dans R0
Rupture de séquence	Arrêt	`HALT`	`HALT`	Arrête l'exécution du programme
	Saut inconditionnel	`BRA`	`BRA 45`	La prochaine instruction sera l'instruction située à l'adresse mémoire 45
	Comparaison	`CMP`	`CMP R0,#23`	Compare la valeur du registre R0 avec la valeur constante 23
	Saut « égal »	`BEQ`	`BEQ LABEL`	Si le précédent CMP est égal, la prochaine instruction sera l'instruction située au label `LABEL`
	Saut « Différent »	`BNE`	`BNE LABEL`	Si le précédent CMP est différent, la prochaine instruction sera l'instruction située au label `LABEL`
	Saut « Plus grand »	`BGT`	`BGT 48`	Si le précédent CMP est strictement plus grand, la prochaine instruction sera l'instruction située à l'adresse mémoire 48 (BGE: Branch Greater or Equal)
	Saut « Plus petit »	`BLT`	`BLT 50`	Si le précédent CMP est plus petit, la prochaine instruction sera l'instruction située à l'adresse mémoire 50 (BLE : Less or Equal)

Exercice 1 : Prise en main du simulateur

Ouvrir le simulateur. **Sélectionner le mode d'affichage** (OPTIONS) de la mémoire **en binaire** ou **en hexadécimal** (selon votre préférence).

Copier-coller le code ci-contre dans le **cadre vert** prévu au **langage assembleur** à **gauche**, puis cliquer sur [**Submit**] en bas du cadre pour valider. Votre code en assembleur doit être traduit en langage machine dans le **cadre bleu** à droite.

```
LDR R0,5
LDR R1,6
ADD R2,R1,R0
STR R2,7
HALT
```

Vous pouvez toujours **modifier les valeurs de la mémoire** à la main, en **cliquant** sur un emplacement ; **La valeur est à saisir en décimal**.

1. **Exécuter** le programme (bouton **RUN**). Est-ce que la mémoire a été **modifiée** avant et après l'exécution ?
2. **Modifier** les emplacements 5 et 6 de la mémoire (y écrire des valeurs numériques) et **ré-exécuter** le programme. Est-ce que la **mémoire** a été modifiée ?
3. À l'aide des **descriptions des opérations** dans le tableau au début du sujet, **décrire** chaque instruction en français et **en déduire** le but de ce programme.
4. **Ajouter** une instruction quelconque (par exemple stocker ou charger une valeur) juste avant l'instruction **`HALT`**. Une erreur se produit, **quelle est-elle** ?

Pour **éviter ce type d'erreur**, on stockera les données d'entrée et de sortie **dans les derniers emplacements disponible dans la mémoire** de la simulation (195 à 199). En réalité, dans un vrai processeur, **il n'y a pas ce genre de sécurité** : on peut tout à fait écrire des données par-dessus le programme, mais cela cause quasiment systématiquement une erreur.

5. **Modifier** le code pour **respecter le précédent paragraphe**, ainsi qu'en remplaçant **l'addition** par une **soustraction**.

Exercice 2 : Labels

Copier-coller le code ci-contre. **En mémoire**, stocker à la main **deux valeurs** dans les emplacements 197 et 198.

```
LDR R0,197
LDR R1,198
CMP R0,R1
BGT CAS2
CAS1:
STR R1,199
BRA FIN
CAS2:
STR R0,199
FIN:
HALT
```

1. **Exécutez** le programme avec plusieurs paires de valeurs. **Quel est son résultat** ?
2. À l'aide des **descriptions des opérations** dans le tableau au début du sujet, **décrire** chaque instruction en français. **En déduire** la signification de **`CAS1:`**, **`CAS2:`**, et **`FIN:`**.
3. En utilisant les mêmes procédés, **écrire un programme** en assembleur qui **détermine le minimum de 3 valeurs** (stockées en mémoire aux emplacements 195, 196, et 197 ; le résultat à stocker en 199).

Exercice 3 : Conversion de Python à l'assembleur

À partir de maintenant, on s'autorise à utiliser les opérations suivantes :

Entrée	`INP`	`INP R1`	Place la valeur écrite dans le cadre d'entrée dans le registre R1
Sortie	`OUT`	`OUT R3`	Affiche dans le cadre de sortie la valeur du registre R3

Convertir les **3 programmes en Python** suivants en assembleur. On rappelle qu'il n'existe pas de concept de variables en assembleur : il faut utiliser les **registres** ou bien des **emplacements mémoire**.

```python
# échange de
# valeurs
a = 15
b = 12
c = a
a = b
b = c
```

```python
# Combien d'années pour
# doubler son capital
capital = 3000
annee = 0
double = capital * 2
while capital <= double:
  capital += 150
  annee += 1
print(annee)
```

```python
# Quel capital après x
# années
capital = input()
annees = input()
for i in range(annees):
  capital += 150
print(capital)
```

Exercice 4 : Multiplication

Nous n'avons pas **d'opération de multiplication**. On veut écrire un programme **le plus optimisé possible** pour réaliser cette opération.

1. **Écrire un programme** en assembleur qui prend **en entrée deux nombres** et **affiche leur produit**.

On sait que $3\times 6=\underbrace{3+3+3+3+3+3}_{6\ fois}=\underbrace{6+6+6}_{3\ fois}$. Le nombre d'additions **est différent** selon quel nombre est utilisé dans l'addition successive.

2. En sachant cela, **améliorer** votre programme pour qu'il **n'exécute qu'un strict minimum d'opérations**.

Exercice 5 : Division euclidienne

Nous n'avons pas **d'opération de division**. Comme on n'a aucune opération sur des valeurs stockée sous format en virgule flottante, on ne peut que se limiter **aux nombres entiers**. On veut donc écrire un programme qui réalise une **division entière** (ou euclidienne).

On rappelle **l'algorithme de division euclidienne** ci-contre.

Implémenter cet algorithme en assembleur (n'afficher que le quotient q en sortie, pas besoin d'afficher r)

ALGORITHME DE DIVISION EUCLIDIENNE

Entrée *: Deux entiers strictement positifs a et b avec $a>b$.*

Sortie *: Un couple (q,r) tel que $a=bq$ et $r<b$*

```
1  q ← 0
2  r ← a
3  Tant que r > b faire :
4  |    q ← q+1
5  |    r ← r−b
6  Retourner (q,r)
```

Exercice 6 : Suite de Fibonacci

Écrire un programme en assembleur qui affiche en mémoire les x (donné en entrée) premiers nombres de la suite de Fibonacci définit ainsi : $F_0=0$, $F_1=1$, $F_n=F_{n-1}+F_{n-2}$ pour $n \geq 2$. Soit : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 …