PROJET D'ARCHITECTURE L2
La machine à Pile

1. Description

Le but de ce projet est de simuler une machine fictive appelée machine à pile. Il y a un programme à écrire qui effectue deux tâches. La première est un assembleur qui va transformer un programme écrit en langage assembleur (dans un fichier) en un programme écrit en langage machine (dans un autre fichier). La deuxième est la simulation proprement dite qui récupère le programme écrit en langage machine et l'exécute instruction après instruction.

La machine à pile

Notre machine comprend deux registres, un espace mémoire de travail et exécute un fichier d'instructions en langage machine. Cela n'a pas grande importance de récupérer directement les instructions dans le fichier ou de les charger d'abord dans un tableau ; on ne peut pas changer les instructions en cours de programme et elles ne sont pas stockées dans l'espace de travail.

Les deux registres de la machine sont :

• PC : dans ce registre se trouve l'adresse de la prochaine instruction à exécuter. Par convention, on fera commencer le programme à l'adresse 0. Chaque instruction occupera une adresse ; autrement dit, PC désigne la prochaine instruction à exécuter (donc pas celle en cours d'exécution), PC+1 celle d'après, PC+2 celle encore après…

• SP : ici se trouve le pointeur de pile. Ce registre indique le premier emplacement libre sur la pile de travail qui se trouve en mémoire. Cette pile commencera à l'adresse zéro. Une partie de la mémoire est donc occupée par cette pile. Au démarrage, SP vaudra 0 car la pile est vide.

Chaque adresse mémoire de l'espace de travail doit pouvoir stocker un nombre entier signé sur quatre octets. L'espace mémoire de travail aura une taille de 5000 adresses.

Le langage assembleur

Chaque instruction est codée sur cinq octets. Le premier contient le code de l'instruction et les quatre autres contiennent (éventuellement) une donnée. Si l'instruction n'a pas de paramètre, les quatre octets de données seront égaux à zéro. Voici les différentes instructions avec leur code machine donné entre parenthèses :

- (0) `push x` : empile le contenu de l'adresse `x` (et donc incrémente ensuite SP) [`(SP)=(x) ; SP++`].
- (1) `ipush` : empile le contenu de l'adresse `n`, où `n` est la valeur du sommet de la pile ; `n` est enlevé de la pile. [`X=(SP-1) ; (SP-1)=(X)`].
- (2) `push# i` : empile la valeur `i` (et donc incrémente ensuite SP) [`(SP)=i ; SP++`].
- (3) `pop x` : met le contenu du haut de la pile à l'adresse `x` et décrémente SP [`SP-- ; (x)=(SP)`].
- (4) `ipop` : met le contenu du haut de la pile -1 à l'adresse `n`, où `n` est la valeur du sommet de la pile. Décrémente SP de 2 [`((SP-1))=(SP-2) ; SP=SP-2`].
- (5) `dup` : duplique le sommet de la pile [`(SP)=(SP-1) ; SP++`].
- (6) `op i` : effectue l'opération indiquée par la donnée `i` (voir plus loin).
- (7) `jmp adr` : additionne `adr` au registre PC (`adr` peut être négatif). [PC désigne l'instruction suivante, donc `jmp -1` est une boucle infinie sur l'instruction elle-même, `jmp 0` ne change rien et `jmp 2` permet de sauter 2 instructions.] [Idem pour `call` et `jpc`]
- (8) `jpz adr` : dépile un élément. Si celui-ci est nul, additionne `adr` au registre PC [`SP-- ; X=(SP) ; Si X=0, PC=PC+adr`].
- (9) `call adr` : appel de procédure. Il faut empiler PC. Ensuite, `adr` est additionné au registre PC.
- (10) `ret` : retour de procédure. On dépile PC.
- (11) `rnd x` : met au sommet de la pile un nombre aléatoire ente 0 et `x-1`.
- (12) `write x` : affiche à l'écran le contenu de la variable d'adresse `x`.
- (13) `read x` : demande à l'utilisateur de rentrer une valeur qui sera mise dans la variable à l'adresse `x`.
- (99) `halt` : arrête la simulation ; fin du programme.

Les codes opération

L'instruction machine « `op` » permet d'effectuer une opération arithmétique ou logique sur les données se trouvant au sommet de la pile. Suivant la donnée, voici les opérations possibles.

0 : et logique bit à bit [`SP-- ; (SP-1)=(SP-1)&(SP)`].
1 : ou logique bit à bit [`SP-- ; (SP-1)=(SP-1)|(SP)`].
2 : ou-exclusif logique bit à bit [`SP-- ; (SP-1)=(SP-1)^(SP)`].
3 : non logique bit à bit [`(SP-1)=~(SP-1)`].
4 : inverse la valeur au sommet de la pile [`(SP-1)=-(SP-1)`].
5 : additionne les deux valeurs au sommet [`SP-- ; (SP-1)=(SP-1)+(SP)`].
6 : soustraction [`SP-- ; (SP-1)=(SP-1)-(SP)`].
7 : multiplication [`SP-- ; (SP-1)=(SP-1)*(SP)`].
8 : division entière [`SP-- ; (SP-1)=(SP-1)/(SP)`].
9 : modulo [`SP-- ; (SP-1)=(SP-1)%(SP)`].
10 : test d'égalité [`SP-- ; (SP-1)=0 si (SP-1)==(SP), 1 sinon`].
11 : test d'inégalité [`SP-- ; (SP-1)=0 si (SP-1)≠(SP), 1 sinon`].
12 : test > [`SP-- ; (SP-1)=0 si (SP-1)>(SP), 1 sinon`].
13 : test ≥ [`SP-- ; (SP-1)=0 si (SP-1)≥(SP), 1 sinon`].
14 : test < [`SP-- ; (SP-1)=0 si (SP-1)<(SP), 1 sinon`].
15 : test ≤ [`SP-- ; (SP-1)=0 si (SP-1)≤(SP), 1 sinon`].

2. Projet

Ce projet est à faire en binôme.
Nous vous demandons d'écrire un programme qui :

- récupère un fichier texte dans lequel est écrit un programme en assembleur (une instruction par ligne) et génère un fichier texte, appelé `hexa.txt`, où est stocké le programme en langage machine (une instruction, soit cinq octets, par ligne). S'il y a des erreurs de syntaxe (mauvaise orthographe, mauvais nombre de paramètres) dans le fichier source, il ne faudra pas générer un fichier code machine mais signaler l'erreur en indiquant la ligne erronée ;
- s'il n'y a pas eu d'erreur, exécute ensuite le fichier `hexa.txt` en simulant le fonctionnement de la machine. Au début de la simulation, on prendra `PC = 0` et `SP = 0`.

Règle habituelle : la discussion entre groupes est autorisée, le partage direct du code est interdit.

Le projet est à réaliser en binôme en langage C et devra m'être envoyé pour pouvoir être testé au Crio Unix. Il sera à rendre avant le 11 février 00h00 sur l'espace MyCourse dédié. Une fois votre fichier envoyé, vous devez avoir un écran de confirmation avec votre fichier et la date de l'envoi. Le format de rendu est une archive au format ZIP contenant :

- le code-source de votre projet (éventuellement organisé en sous-dossiers) ;
- un répertoire *docs* contenant :
  - une documentation pour l'utilisateur *user.pdf* décrivant à un utilisateur quelconque comment se servir de votre projet,
  - une documentation pour le développeur *dev.pdf,* donnant des explications sur l'implémentation et indiquant quelles ont été les difficultés rencontrées au cours du projet ;
- un exécutable C dont le nom sera `simulateur`.

L'archive aura pour nom `Nom1Nom2.zip`, où `Nom1` et `Nom2` sont les noms des membres du binôme par ordre alphabétique. L'extraction de l'archive devra créer un dossier `Nom1Nom2` contenant les éléments précisés ci-dessus.

Notation

La note de projet tiendra compte de :

- la qualité du programme (correction de la simulation, lisibilité du code, facilité de mise en œuvre, robustesse...) ;
- la qualité du rapport (description détaillée du programme, améliorations possibles...).

Tests
Les projets seront testés sur des fichiers similaires à celui donné dans l'exemple ci-dessous (aussi mis sur MyCourse). Afin de faciliter les tests, l'exécutable prendra le nom du fichier à tester en argument, effectuera la traduction en langage machine en générant le fichier de sortie, puis effectuera la simulation. Une fois la commande lancée, l'utilisateur ne doit pas devoir entrer lui-même quoi que ce soit au clavier, sauf des données demandées par le programme (avec l'instruction `read`).

Le programme sera lancé par la commande ci-dessous (avec un fichier en assembleur appelé par exemple `pgm.txt`) :
```
$ ./simulateur pgm.txt
```

3. Spécifications
Les fichiers assembleur lus par votre programme auront la structure suivante :

- une instruction par ligne ;
- chaque ligne est composée d'une étiquette optionnelle, de 0, 1 ou plusieurs espaces ou d'une tabulation, du code opération de l'instruction (en minuscule), suivi, s'il y a une donnée, d'une espace et de la donnée ;
- la valeur d'une donnée sera spécifiée en décimal.

Le fichier `hexa.txt` en langage machine généré par l'assembleur sera composé d'une instruction, soit 5 octets, par ligne. Chaque octet sera écrit en hexadécimal et le premier octet (celui du code opération) sera séparé par une espace des quatre octets suivants correspondant à la donnée.

Une instruction peut avoir une étiquette, représentée par « `etiq:` » avant l'instruction. Le nom de l'étiquette peut être composée de lettres et chiffres et sera immédiatement suivi d'un deux-points lors de sa définition.
Un saut « `jmp` (ou `jpz`) `etiq` » pourra alors se traduire par « `jmp adr` » où *adr* est la différence entre l'adresse actuelle et l'adresse calculée de l'étiquette. La première instruction sera toujours placée à l'adresse 0, la deuxième à l'adresse 1, ce qui permet de calculer l'adresse des sauts si ceux-ci sont donnés par une étiquette.

Exemple
Le programme à gauche (qui affiche l'inverse d'un nombre tant que l'utilisateur ne rentre pas la valeur 0) devra générer le fichier de droite :

`ici: read 1000`	`0d 000003e8`
`push 1000`	`00 000003e8`
`jpz fin`	`08 00000005`
`push 1000`	`00 000003e8`
`op 4`	`06 00000004`
`pop 1000`	`03 000003e8`
`write 1000`	`0c 000003e8`
`jmp ici`	`07 fffffff8`
`fin: halt`	`63 00000000`