PROJET D'ARCHITECTURE L2
Simulation d'une machine virtuelle

1. Description

Le but de ce projet est de simuler une machine fictive composée d'une mémoire et d'un microprocesseur. Vous trouverez ci-dessous la description du processeur. 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

Notre machine comprend une mémoire de 64 kilo-octets, adressée de 0 à 65535. Dans cette mémoire sont stockés le programme, à partir de l'adresse 0, ainsi que les données de ce programme.
Le processeur comprend les registres suivants :

• PC : dans ce registre se trouve l'adresse de la prochaine instruction à exécuter ;

• r0 à r31 : 32 registres généraux de 16 bits chacun. r0 est toujours égal à 0, même après une instruction le modifiant ;

• un registre d'état : ce registre comprend 3 bits intéressants, les bits Z, C et N. Ils sont mis à jour après l'exécution d'une instruction, suivant le résultat de l'instruction. Z est mis à 1 si le résultat est nul, à 0 sinon ; C est mis à 1 s'il y a une retenue, à 0 sinon et N est la recopie du bit de poids fort du résultat.
Aucune instruction ne travaille directement sur ce registre, on pourra donc stocker ces trois bits séparément.

Le langage machine

Chaque instruction est codée sur 32 bits. Voici le format général.

5 bits	5 bits	5 bits	1 bit	16 bits
code op.	Dest (rd)	Src 1 (rn)	Imm.	Src 2 (rm ou #n)

Les 5 premiers bits correspondent au code opératoire (première colonne dans les tables d'instructions).
Les 5 bits suivant donnent le numéro du registre destination (r*d* dans l'assembleur)
Les 5 bits suivant donnent le numéro du premier registre source (r*n* dans l'assembleur).
Le bit suivant indique, quand il est à 1, que le deuxième opérande est une valeur immédiate (#n dans l'assembleur), codée dans les 16 bits suivant ; s'il est à 0, l'opérande est un registre (r*m* dans l'assembleur) dont le numéro *m* est codé dans les 5 bits de poids faible du champ Src 2.

i) instructions arithmétiques et logiques

0	`add rd, rn, S`	rd ← rn + S
1	`sub rd, rn, S`	rd ← rn - S
2	`mul rd, rn, S`	rd ← rn × S (on ne prend que les 8 bits de poids faible de rn et S)
3	`div rd, rn, S`	rd ← rn / S (division entière)
4	`or rd, rn, S`	rd ← rn | S
5	`xor rd, rn, S`	rd ← rn ^ S
6	`and rd, rn, S`	rd ← rn & S
7	`shl rd, rn, S`	rd ← rn décalé S fois à gauche (ou à droite si S est négatif), on remplace les nouveaux bits par 0.

Après l'exécution de ces instructions, les bits d'état sont mis à jour suivant le résultat du calcul. Attention, il s'agit bien du résultat du calcul, même si le registre destination est r0, qui vaut toujours zéro. Dans le cas du décalage (`shl`), le bit C reçoit le dernier bit qui disparaît.

S peut valoir r*m*, le bit Imm est alors à 0 et les 5 bits de poids faible de l'instruction porte la valeur *m* ; soit S vaut une valeur numérique immédiate sur 16 bits, écrite comme #*n*, avec *n* en décimal par défaut (ou #*hn* si elle est en hexadécimal) et elle est codée directement dans les 16 bits de poids faible de l'instruction.

Toutes ces instructions travaillent sur les 16 bits des registres sauf la multiplication qui ne prend que les 8 bits de poids faible des deux opérandes pour que le résultat, sur 16 bits, rentre dans r*d*.

ii) instructions de transfert

8	`ldb rd, (rn)S`	rd ← contenu de l'adresse (rn+S), sur 1 octet
9	`ldw rd, (rn)S`	rd ← contenu de l'adresse (rn+S), sur 2 octets
10	`stb (rd)S, rn`	l'adresse (rd+S) reçoit le contenu de rn, sur 1 octet
11	`stw (rd)S, rn`	l'adresse (rd+S) reçoit le contenu de rn, sur 2 octets

Ces instructions affectent le registre d'état.

`ldw` charge l'octet de l'adresse précisée dans les 8 bits de poids faible de r*d* et l'octet suivant dans les 8 bits de poids fort. Si le chargement est sur un octet (`ldb`), la valeur est étendue à 16 bits dans r*d* (c'est-à-dire que le bit de signe est répété dans les 8 bits de poids fort).

`stw` sauvegarde en mémoire dans le même ordre.

S peut être r*m*, l'adresse de l'opérande est alors la somme du contenu des deux registres (et Imm vaut 0), ou une valeur numérique décimale immédiate (ou hexadécimale si elle est précédée d'un h) et l'adresse est alors la somme du contenu du registre et de la valeur (et Imm vaut 1).

iii) instructions de sauts

16	`jmp S`	PC ← S
17	`jzs S`	PC ← S, si Z=1 *(Jump if Z Set)*
18	`jzc S`	PC ← S, si Z=0 *(Jump if Z Clear)*
19	`jcs S`	PC ← S, si C=1 *(Jump if C Set)*
20	`jcc S`	PC ← S, si C=0 *(Jump if C Clear)*
21	`jns S`	PC ← S, si N=1 *(Jump if N Set)*
22	`jnc S`	PC ← S, si N=0 *(Jump if N Clear)*

S peut être r*m*, l'adresse de saut est alors égale au contenu du registre (et Imm vaut 0), ou une valeur numérique décimale immédiate (ou hexadécimale si elle est précédée d'un h) et l'adresse est alors directement cette valeur (et Imm vaut 1).

Les 5 bits du champ Dest et les 5 bits du champ Src 1 ne sont pas significatifs. Ces instructions n'affectent pas le registre d'état.

L'exécution se poursuit à l'adresse indiquée si le saut est pris, à l'instruction suivante si ce n'est pas le cas.

iv) instructions d'entrée-sortie

28	`in rd`	met dans rd une valeur entrée par l'utilisateur au clavier
29	`out rd`	affiche à l'écran en décimal le contenu du registre rd

Ces instructions affectent le registre d'état.
Elles travaillent sur les 16 bits du registre. Les valeurs sont signées en complément à 2.
Les champs Src 1, Imm et Src 2 sont sans signification.

v) instructions diverses

30	`rnd rd, rn, S`	met dans rd un nombre aléatoire entier entre rn et S-1 inclus
31	`hlt`	termine l'exécution du programme

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 sera stocké le programme en langage machine (une instruction, soit quatre 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`.

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 **4 février 2020** à **01h00** 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 :

- principalement de la qualité du programme (correction de la simulation, lisibilité du code, facilité de mise en œuvre, robustesse...) ;
- également de 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 `in`).

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

Une soutenance au Crio Unix sera probablement organisée juste après le rendu du projet.

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 tabulations, du code opération de l'instruction (en minuscule), suivi d'une ou plusieurs espaces, et des opérandes séparés par des virgules et éventuellement des espaces ;

Attention, toutes les valeurs numériques décimales peuvent être négatives.
Le fichier `hexa.txt` en langage machine généré par l'assembleur sera composé d'une instruction, soit quatre octets, par ligne. Chaque octet sera écrit en hexadécimal et les quatre seront écrits à la file sans espace.

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 autre) `etiq` » pourra alors s'écrire « `jmp adr` » où *adr* est l'adresse calculée de l'étiquette.
La première instruction sera toujours placée à l'adresse 0, la deuxième à l'adresse 4 (puisque chaque instruction fait 32 bits, soit 4 octets), 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: in r1`	`E0400000`
`jzs fin`	`88010014`
`sub r1, r0, r1`	`08400001`
`out r1`	`E8400000`
`jmp ici`	`80010000`
`fin: hlt`	`F8000000`