INF1600

Travail pratique 2

Architecture et Introduction à l'assembleur IA-32

Tarek Ould-Bachir

Sami Sadfa

Wail Ayad

Département de Génie Informatique et Génie Logiciel
Polytechnique Montréal

1 Sommaire

1.1 Remise

Voici les détails concernant la remise de ce travail pratique :

- Méthode : sur Moodle, une seule remise par équipe, incluant un rapport PDF et les fichiers sources des exercices 1, 2 et 3.
- Format: un rapport PDF. Incluez une page titre où doivent figurer les noms et matricules des deux membres de l'équipe, votre groupe de laboratoire, le nom et le sigle du cours, la date de remise et le nom de l'École. Dans une seconde page, incluez le barème de la section 1.2. Finalement, pensez à inclure les réponses aux questions et des captures d'écran lorsque requis.
- Pour l'exercice 1, remettez le fichier source complété sous la forme `<matricule1>-<matricule2>-<tp2exo1>.vhdl`
- Pour l'exercice 2 et 3, remettez les fichiers sources complétés sous la forme `<matricule1>-<matricule2>-<tp2exo2>.s` `<matricule1>-<matricule2>-<tp2exo3>.s`
- Langue écrite : Français.
- Distribution : les deux membres de l'équipe recevront la même note.

Attention : L'équipe de deux que vous formez pour ce TP sera définitive jusqu'au TP5. Il ne sera pas possible de changer d'équipe au cours de la session.

1.2 Barème

Le travaux pratiques 1 à 5 sont notés sur 4 points, pour un total de 20/20 Le TP2 est noté selon le barème suivant. Reproduisez ce tableau dans le document PDF que vous remettrez.

TP 2			**/4,00**
Exercice 1			/1,00
	Question 1	/0,50	
	Question 2	/0,50	
Exercice 2			/2,00
	Question 1	/1,00	
	Question 2	/1,00	
Exercice 3			/1,00
	Question 1	/1,00	

2 Exercice 1 : Architecture processeur

Les instructions du microprocesseur à accumulateur vu au TP1 ont été modifié. Le processeur possède un registre de plus (le registre `MA`) et son jeu d'instructions à été étendu.

- L'instruction `sub` permettant d'effectuer une soustraction a été ajoutée.
- Les instructions `adda` et `suba` permettent de d'ajouter ou de soustraire à `MA` une valeur en mémoire disponible à l'adresse fixe `disp`.
- Les instructions `addx` et `subx` permettent d'ajouter ou de soustraire à `ACC` une valeur en mémoire disponible à l'adresse `MA`.
- Les instructions `lda` et `sta` servent à (dé)charger le contenu de la mémoire à `disp` vers le registre d'adresse `MA`.
- Les instructions d'adressage indirect `sti` et `ldi` qui exploitent le registre `MA` sont également ajoutées au processeur.
- Finalement, les instructions `br`, `brz` et `brnz` permettent d'exécuter une instruction de branchement non conditionnelle (`br`) ou conditionnel (`brz` et `brnz`).

OPCODE	**Hexadécimal**	**Description**
`add`	`0x0000`	`ACC ← ACC + mem[disp]`
`sub`	`0x0100`	`ACC ← ACC – mem[disp]`
`mul`	`0x0200`	`ACC ← ACC * mem[disp]`
`adda`	`0x0300`	`MA ← MA + mem[disp]`
`suba`	`0x0400`	`MA ← MA – mem[disp]`
`addx`	`0x0500`	`ACC ← ACC + mem[MA]`
`subx`	`0x0600`	`ACC ← ACC – mem[MA]`
`ld (load)`	`0x0800`	`ACC ← mem[disp]`
`st (store)`	`0x0700`	`mem[disp] ← ACC`
`lda (load address)`	`0x0A00`	`MA ← mem[disp]`
`sta (store address)`	`0x0900`	`mem[disp] ← MA`
`ldi (load indirect)`	`0x0C00`	`ACC ← mem[MA]`
`sti (store indirect)`	`0x0B00`	`mem[MA] ← ACC`
`br`	`0x0D00`	`PC ← disp`
`brz`	`0x0E00`	`if( ACC = 0) PC ← disp`
`brnz`	`0x0F00`	`if( ACC ≠ 0) PC ← disp`
`stop`	`0x1000`	La boucle d'exécution arrête.
`nop (no operation)`	`0x1100`	Ne fait rien.

Un programme, `program_0`, a déjà été écrit pour vous dans le fichier `acc_proc_programs.vhd`. Analysez ce programme pour répondre aux sous questions ci-dessous.

Question 1 : Donnez le contenu du registre ACC quand le processeur a terminé d'exécuter `program_0`. Expliquez brièvement ce que fait le programme.

Question 2 : Modifiez le contenu de `program_1` dans `acc_proc_programs.vhd` afin d'implémenter un programme qui calcule les six (6) premières valeurs de la suite de Fibonacci, soit `Fib(0)` à `Fib(5)`, et les stocke séquentiellement en mémoire en utilisant les instructions de branchement. On supposera que les deux premières valeurs, `Fib(0) = 0` et `Fib(1) = 1` sont déjà présentes en mémoire. Pour rappel, `Fib(n+2) = Fib(n+1) + Fib(n)`. Comparez le nombre d'instructions nécessaires pour écrire `program_1` au nombre obtenu au TP1.

3 Exercice 2 : Assembleur avec processeur à pile

L'architecture IA-32 implémente un coprocesseur spécialisé pour le calcul scientifique, appelé le FPU (Floating Point Unit). Le FPU possède sa propre pile pouvant stocker jusqu'à 8 registres de 80 bits. La notation **`st`** réfère à un registre sur la pile, par exemple **`st[0]`** point vers le premier registre en haut de la pile. Le FPU étend le jeu d'instruction x86 en ajouter des instructions supplémentaires, voici quelques instructions du FPU qui sont pertinentes pour cet exercice.

Instruction	**Description**
`flds` **`addr`**	Ajoute au-dessus de la pile l'entier à l'adresse mémoire **`addr.`** (`st[1]` prend la valeur de `st[0]` et `st[0]` devient la nouvelle valeur chargée de la mémoire.
`fstps` **`addr`**	Retire l'élément `st[0]` pour le mettre en mémoire principale à l'adresse **`addr.`** `st[1]` devient `st[0]`
`faddp`	`st[0]` est additionné à `st[1]` et le résultat remplace `st[0]`. `st[1]` est libéré.
`fsubp`	`st[1]` est soustrait à `st[0]` et le résultat remplace st[0]. st[1] est libéré.
`fsubrp`	`st[0]` est soustrait à `st[1]` et le résultat remplace `st[0]`. `st[1]` est libéré.
`fmul` **`x`**	`st[0] = st[0] * x`
`fmulp`	`st[0]` est multiplié avec `st[1]` et le résultat remplace `st[0]`. `st[1]` est libéré.
`fdivp`	`st[0]` est divisé par `st[1]` et le résultat remplace `st[0]`. `st[1]` est libéré.
`fdivrp`	`st[1]` est divisé par `st[0]` et le résultat remplace `st[0]`. `st[1]` est libéré.
`fcos`	`st[0] = cos(st[0])`
`fsqrt`	`st[0] = sqrt(st[0])`

Question 1 : À l'aide des instructions ci-dessus, écrivez l'expression suivante en assembleur

$$a = \left(\frac{e \cdot d - b}{g + f} - f\right)\left(\frac{b + c}{d}\right)$$

Question 2: À l'aide des instructions ci-dessus, écrivez l'expression suivante en assembleur

$$c = \sqrt{a^2 + b^2 - (2 \cdot a \cdot b \cdot cos(\theta))}$$

Modifiez le fichier `exo2.s` et utilisez la commande pour compiler votre exécutable

```
make
```

Lancer l'exécutable avec la commande suivante pour vérifier votre résultat.

```
./exo2
```

Note : Les variables `a, b, c, d, e, f, g, theta` et `constant` sont des variables de type `float` et leurs adresses en assembleur sont respectivement les symboles `a, b, c, d, e, f, g, theta` et `constant`. Par exemple, il est possible d'appeler :

```
flds a
```

pour mettre a au-dessus de la pile.

4 Exercice 3 : Conditions et branchements

Dans cet exercice vous devrez convertir un code en langage C fourni en assembleur.

Le code est le suivant :

```
int i;
for (i=0; i<=10; i++){

	a = d + i + e - b;
	if ((b - 4000) < ( c - 500 )) {
		c = c + 500;
		if (b > c) {
			b = b + 1000;
		}
	}
	else {
		b = b - e + i;
		d = d + 500 + a;
	}

}
```

Question 1 : Écrire la version assembleur x86 du code ci-dessus dans le fichier `exo3.s`. Pour ce faire, modifiez le fichier `exo3.s` et utilisez la commande

```
make
```

pour compiler votre exécutable. Lancez l'exécutable avec la commande suivante pour vérifier votre résultat.

```
./exo3
```