Émulateur MIPS

1 Rendu

À la fin de ce tutoriel, vous devez soumettre une archive qui respecte la architecture suivante :

```
rendu-tp-login_x.zip
|-- login_x/
    |-- AUTHORS*
    |-- README
    |-- MyMiniMips.sln*
    |-- MyMiniMips*
        |-- Everything but bin/ and obj/
        |-- CPU.cs
        |-- ALU.cs
        |-- Instruction.cs
        |-- Program.cs
```

Bien sûr, vous devez remplacer *"login_x"* par votre propre login. Tous les fichiers marqués d'un astérisque sont obligatoires

N'oubliez pas de vérifier les points suivants avant de soumettre votre travail :

- Le fichier `AUTHORS` est au format usuel (une *, un espace, votre login et un retour a la ligne).
- Pas de dossier `bin` ou `obj` dans votre projet.
- **Le code doit compiler !**

2 Introduction

Le processeur (ou CPU de l'anglais Central Processing Unit) est le composant de l'ordinateur qui exécute les instructions machine des programmes informatiques.

Autrement dit c'est la partie de votre ordinateur qui va exécuter les instructions et traiter les données des programmes.

Le CPU de votre ordinateur est très probablement un Intel x86/64

2.1 Objectifs

L'objectif de ce TP est simple : créer un émulateur MIPS 32 bits.

- émuler c'est 'chercher à imiter', dans notre cas nous cherchons à imiter le comportement d'un microprocesseur MIPS).
- un microprocesseur est un processeur dont tous les composants ont été suffisamment miniaturisés pour être regroupés dans un unique boîtier.
- Une instruction machine est ce qui compose tout programme sur votre ordinateur, c'est l'ordre le plus basique que peut comprendre et exécuter un ordinateur.

3 Cours

3.1 MIPS

MIPS (Microprocessor without Interlocked Pipeline Stages) désigne un type de microprocesseur qui est facile à comprendre et à mettre en place.

C'est aussi un type de microprocesseur qui appartient à la catégorie des microprocesseurs possédant un jeu d'instruction machine réduit (RISC : Reduced Instruction Set Computing). C'est à dire que le nombre d'instruction machine différentes est faible.

De plus, les instructions sont très facile à décoder (elles sont en binaire et dans notre cas possèdent une taille de 32 bits) et ont un comportement simple (ajouter, multiplier, ...).

Pour récapituler :
— Un processeur exécute des instructions ;
— MIPS ne comporte que peut d'instructions différentes ;
— Une instruction machine est écrite en binaire et devra être décodée.

3.1.1 Registre

Un registre est une petite zone de mémoire interne au CPU. Il s'agit de la mémoire la plus rapide de l'ordinateur car la plus proche du CPU.

Ces registres servent au stockage de valeurs importantes ou intermédiaires avec lesquelles le processeur est en train de travailler.

Les registres sont des composants importants d'un microprocesseur, d'une part pour leur rapidité, d'autre part car une partie importante des instructions machines agissent sur eux.

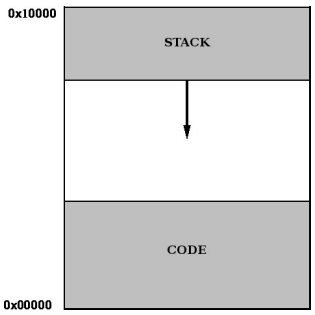
Dans le cas de MIPS, ce microprocesseur comprend 32 registres dont chacun possède une taille de 32 bits.

En MIPS, il n'existe que trois registres spéciaux :
— Le registre 0 ou $zero qui contient toujours la valeur 0 ;
— Le registre 31 ou $ra qui contient l'adresse de retour de la fonction courante ;
— Le registre 29 ou $sp qui contient l'adresse de la pile.

Pour récapituler :
— Les registres sont des petits bouts de mémoire très rapide ;
— En MIPS ils sont au nombre de 32 et chacun possède une taille de 32 bits ;
— Les registres important à la réalisation de notre émulateur sont le registre ***$zero*** qui contient toujours ***0***, le registre ***$ra*** qui contient l'adresse de retour d'une fonction ainsi que le ***registre 29*** qu'il faudra initialiser au top de la stack.

3.1.2 Mémoire

Nous allons supposer une vision simpliste de la mémoire pour notre émulateur :



Dans les adresses basses (autour de 0x00000) se situent les instructions machines. Au dessus (comprendre tout en haut) se situe la pile (qui fonctionne comme une pile normale) qui grandit vers le bas.
Les données sont stockées sur la pile et leur adresse est indiquée par le pointeur de pile (***$sp***).

3.2 Instruction Machine

Les instructions sont encodées en binaire et stockées dans la mémoire comme n'importe quelle donnée.

Ce qui veut dire non seulement qu'elles peuvent être lues mais aussi modifiées.

Les instructions machines en MIPS correspondent à des actions simples comme ajouter, multiplier ou soustraire deux registres et mettre le résultat dans le premier.

Elles peuvent aussi correspondre à des instructions qui permettent de modifier le flot du programme (par exemple sauter des instructions)...

...ou encore des instructions pour stocker un registre sur la pile ou modifier une valeur dans la mémoire (comme par exemple une instruction).

Une instruction en MIPS possède une taille fixe de 32 bits (soit un int en C#) ou tout simplement 4 bytes et, peut se décomposer en différents champs.

Chaque champ indique quelque chose à propos de l'instruction (l'action à effectuer, les paramètres de l'action, ...).

En MIPS il existe 3 types d'instructions :
- Les Instructions de ***type R*** : Les instructions qui agissent (principalement) sur des registres.
- Les Instructions de ***type I*** : Les instructions qui utilisent des constantes ;
- Les Instructions de ***type J*** : Les instructions de saut ;

3.2.1 Les Instructions de type R

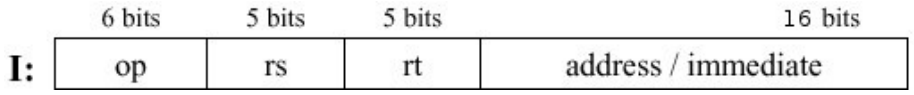
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R:	op	rs	rt	rd	shamt	funct

Les Instructions de ***type R*** se décomposent selon le schéma ci-dessus.
On reconnaît une instruction de ***type R*** au fait que son champ ***op*** (ou opcode) est a ***0***. Afin de déterminer l'action a effectuer il faut regarder le champ ***funct*** qui contient le numéro de l'action.

Il existe un tableau qui associe le numéro de l'action avec l'action, qui est fourni en lien à la fin du TP.

Enfin dans le schéma plus haut les champs ***Rs, Rt et Rd*** correspondent respectivement aux registres source, ***target*** et destination de l'action. Le champ ***shamt*** ne sera pas utilisé dans ce tp, et ne sera donc pas développé par soucis de clarté.

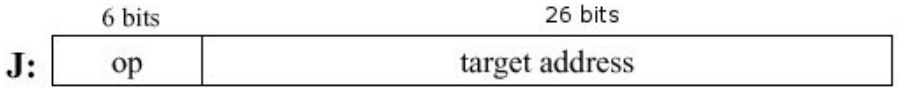
3.2.2 Les Instructions de type I



Les Instructions de ***type I*** se décomposent selon le schéma ci-dessus. Le champ ***op*** (opcode) correspond à l'opération à effectuer, ***Rs et Rt*** correspondent respectivement aux registres ***source et target***.

Enfin, ***address / immediate*** correspond à une constante (un entier ou une adresse dans la mémoire) sur laquelle on veut effectuer des actions.

3.2.3 Les Instructions de type J



Les Instructions de ***type J*** se décomposent selon le schéma ci-dessus. Le champ op correspond à l'opération à effectuer.

Ces instructions sont au nombre de deux :

— ***Jmp*** : permet de sauter dans la mémoire.

— ***Jal*** : Effectue la même action que ***Jmp*** mais sauvegarde l'adresse dans ***$ra*** avant de sauter sur le stack. C'est l'instruction que l'on utilise pour appeler une fonction.

4 Exercices

4.1 Partie 1 : Le CPU

Dans cette première partie vous allez réaliser l'objet CPU. Son rôle sera de mettre en relation les différents objets.

Le CPU contient :

— Un accès a la RAM que vous représenterez par un tableau de 64 KiB c'est a dire 64 * 1024 bytes ;

— 32 registres d'une taille de 32 bits chacun. Vous noterez qu'un entier en C# possède une taille de 32 bits.

— Une valeur à part appelée le ***Program Counter*** qui indique l'adresse de l'instruction courante. Celui-ci commence a 0 et est s'incrémente a chaque instruction exécuté.

De plus vous réaliserez le constructeur et la méthode suivante :

```
public CPU(string fileName);
public void LoadFile(string fileName);
```

Le fichier file contient les instructions MIPS sous forme binaire. Vous devez donc copier le contenu du fichier dans la RAM a partir de l'adresse ***0x000000***.

Le constructeur s'occupera d'initialiser les attributs ainsi que d'appeler la fonction ***LoadFile***.

Hint : Un tableau de bytes sera utile...

4.2 Partie 2 : Les Instruction

Vous créerez et compléterez la classe Instruction :

```
public class Instruction {
    int Op    {get; private set; }
    int Rs    {get; private set; }
    int Rt    {get; private set; }
    int Rd    {get; private set; }
    int Funct {get; private set; }
    int Imm   {get; private set; }
    int Addr  {get; private set; }

    public Instruction(int ins);
}
```

On ne s'occupe pas ici du type d'instruction. Considérez toutes les possibilités et remplissez les champs en fonction. Selon le type de l'instruction, son utilisation sera simplement différente par la suite.

```
0x00004020
// op: 0x00
// rs: 0x00
// rt: 0x00
// rd: 0x08
// funct: 0x20
// imm: 0x4020
// addr: 0x4020
```

4.3 Partie 3 : ALU

L'ALU est le composant physique de votre ordinateur qui s'occupe d'effectuer les opérations mathématiques.

Vous implémenterez donc la classe ALU suivante :

```
public class ALU {
        public ALU(CPU cpu);
    public void Exec(Instruction i);
}
```

Notez que vous aurez à implémenter la fonction ***Exec*** tout au long de cette partie.

4.3.1 1 : Les syscalls

Vous implémenterez tout d'abord l'instruction ***syscall***, qui est spéciale :

Cette instruction permet de faire appel à des fonctions du système ; par exemple afficher un entier sur la console :

```
addi    $a0, $t0, 0 # move value of t0 in a0
addi    $v0, $0,  1 # displays value in a0
syscall
```

Lorsque l'instruction ***syscall*** est exécutée (sur une architecture MIPS) le système prend la main, regarde dans le registre $v0 et exécute la fonction associe à ce numéro. Par exemple ici, si la valeur est ***1***, la fonction affichera sur la console la valeur contenue par ***$a0***.

Vous devrez donc maintenant implémenter la fonction suivante :

```
public void Exec(Instruction i);
```

Si elle prend une instruction de type ***syscall***, elle exécutera les actions suivantes en fonction de la valeur du registre ***$v0*** :

— 1 : ***$a0*** = Entier a afficher
— 4 : ***$a0*** = adresse de la String terminée par un caractère null
— 5 : Lire un entier et le mettre dans ***$v0***
— 10 : Exit
— 11 : Afficher le caractère contenu dans ***$a0***

Complétez en conséquence la fonction suivante :

```
public void Exec(Instruction i);
```

4.3.2 2 : Les instructions de type R

Dans cette section vous devrez être capable de gérer les instructions suivantes :
— ***add : Opcode*** 0x20, R[rd] = R[rs] + R[rt]
— ***addu : Opcode*** 0x21, R[rd] = R[rs] + R[rt] (addition non signée)
— ***sub : Opcode*** 0x22, R[rd] = R[rs] - R[rt]

N'oubliez pas que ***l'Opcode*** pour les instructions de ***type R*** correspond à son champ ***Funct***.

Complétez en conséquence la fonction suivante :

```
public void Exec(Instruction i);
```

4.3.3 3 : Les instructions de type I

Dans cette section vous devrez être capable de gérer les instructions suivantes :
— ***addi : Opcode*** 0x08, R[rd] = R[rs] + SignExtImm
— ***addiu : Opcode*** 0x21, R[rd] = R[rs] + SignExtImm (addition non signée)
— ***ori : Opcode*** 0x22, R[rd] = R[rs] | ZeroExtImm
— ***beq : Opcode*** 0x04, if (R[rs] == R[rt]) PC = PC + 4 + BranchAddr
— ***bne : Opcode*** 0x05, if (R[rs] != R[rt]) PC = PC + 4 + BranchAddr

Complétez en conséquence la fonction suivante :

```
public void Exec(Instruction i);
```

Notez ici que l'immédiat que prend une instruction de type I doit être ***SignExtended*** ou ***ZeroExtended***. En effet, comme l'immédiat ne peut être que codé sur 28 bits, il faut l'étendre afin qu'il atteigne les 32 bits.

Un simple ***cast*** effectuera une ***Zero Extension***. Une extension de signe doit être capable de récupérer le bit de signe (le 28eme bit) et de le "copier" sur les 6 bits restants.

4.3.4 4 : Les instructions de type J

Enfin vous devrez prendre en charge les deux instructions suivantes :
— ***j*** **: *Opcode*** 0x02, PC = JumpAddr
— ***jal*** **: *Opcode*** 0x03, R[31] = PC + 8; PC = JumpAddr
Attention : En raison du fait que votre RAM ne fait que 64K vous ne devez prendre que les 16 premiers bits de l'adresse. Le reste de l'adresse peut et **doit** être ignoré.

4.4 Partie 5 : Debug

Votre fonction ***Exec*** est presque finie; dans le but de vous assurer que vous décodez bien les bonnes instructions, nous vous demandons d'afficher sur la console les instructions que vous exécutez avec le format suivant :

```
[my_minimips] Executing pc = 0x00000000: 0x34040064: ori r4, r0, 100
[my_minimips] Executing pc = 0x00000004: 0x34020004: ori r2, r0, 4
[my_minimips] Executing pc = 0x00000008: 0x0000000c: syscall
Hello World!
[my_minimips] Executing pc = 0x00000004: 0x3402000a: ori r2, r0, 10
[my_minimips] Executing pc = 0x00000008: 0x0000000c: syscall
```

4.5 Partie 6 : Run

Il ne nous reste plus qu'à tester votre émulateur, c'est la partie où vous allez lier le code que vous venez d'écrire.

Vous devez écrire la fonction suivante :

```
public void Run();
```

Dans la classe CPU, cette fonction ira chercher l'instruction courante (en utilisant le PC), la décodera grâce a la classe Instruction et la livrera a l'ALU qui exécutera l'instruction.

Enfin elle incrémentera le PC de 4 puis recommencera ce cycle.

4.6 Bonus

Voici une liste non exhaustive d'instructions et/ou de techniques que vous pouvez implémenter :
— FPU : une floating point unit qui permet de gérer les floats (ainsi que les instructions assembleurs avec);
— Compilation JIT
— ...

5 Documents

http ://inst.eecs.berkeley.edu/c̃s61c/resources/MIPS_Green_Sheet.pdf
http ://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html

The bullshit is strong with this one