Garnier Nathan
Farel Antoine

Gestion et Implémentation des mémoires de l'émulateur MIPS

Module de registres

Définition des registres

Les registres sont définis à l'aide d'une liste chaînée. Les éléments de cette liste sont définis ainsi :

```
typedef struct Register Register;

struct Register {

  char address_value;
  int32_t value;
  Register *suivant;
};
```

- `"adress_value"` est l'adresse du registre. Nous l'avons définie comme char étant donné qu'un char est défini sur 1 octet et que nous avons que 32 adresses (4 bits).
- `"value"` est la valeur stockée dans le registre. étant définie sur 32 bits, nous utilisons le type `int32_t` défini dans le paquet stdint.
- `"suivant"` est tout simplement le pointeur vers la prochaine valeur de la liste chaînée.

Nous avons ensuite défini tous nos registres directement dans le `.h`, en faisant attention à bien chaîner la liste.
La liste va du registre `zero` :

```
static Register zero = {REGISTER_ZERO_ADDR, 0, &at};
```

Au registre `ra` :

```
static Register ra = {REGISTER_RA_ADDR, 0, NULL};
```

Les valeurs des adresses `REGISTER_XX_ADDR` ont étés définies au préalable pour faciliter la lecture :

```
#define REGISTER_ZERO_ADDR 0x00
...
#define REGISTER_RA_ADDR 0x1F
```

Ainsi, il ne reste plus qu'à définir la liste chaînée que nous utiliserons pour faire nos calculs. Etant donné que cette liste ne sera utilisée que de notre côté, nous pouvons la définir directement dans le `.h`:

```
static Register *head_lst_register = &zero;
```

Fonctions

Nous avons définis deux fonctions :

```
//read_register
//Permet de lire le contenu d'un registre.
//Entrée : (int32_t*) pointeur vers espace
//         memoire qui contiendra la valeur du registre
//         (char) l'addresse du registre
//Sortie : (Int), 0 = succes, 1 = echec
int read_register(int32_t* value, char hex_address);

//write_register
//Permet de lire le contenu d'un registre.
//Entrée : (int32_t) valeur que l'on veut mettre dans le registre
//         (char) l'addresse du registre
//Sortie : (Int), 0 = succes, 1 = echec
int write_register(int32_t value, char hex_address);
```

Les deux fonctions retournent une valeur booléenne qui retourne 0 si tout s'est bien passé, 1 sinon. De manière générale, ce retour nous permet de ne pas avoir de problèmes si la valeur de l'adresse est hors de nos limites fixées (entre 0 et 31).
Les deux fonctions ont en argument l'adresse du registre à lire/écrire en hexadécimal.
La fonction `write_register` possède simplement un `int32_t` en argument, mais la fonction `read_register` possède un pointeur vers `int32_t`. En effet, une fonction ne pouvant pas avoir deux paramètres en sortie, nous avons décidé de passer en argument un pointeur vers `int32_t` où on stockera la valeur contenue dans le registre choisi.

Module de mémoire

Définition de la mémoire

Ici encore, nous avons définis la mémoire sous forme de liste chaînée, dont les éléments sont définis tel que :

```
typedef struct VRAM_bloc VRAM_bloc;
#define TAILLEMAX 4000000000
struct VRAM_bloc {

  unsigned int address;
  int32_t value;
  VRAM_bloc *suivant;
};
```

Tout d'abord, nous avons définis une taille max, taille correspondant à l'adresse 4 Go
`address` : adresse du bloc dans la mémoire. Nous l'avons défini comme `unsigned int` car une adresse est forcément positive. Cela ne change pas grand chose au final sur une machine 64 bits, cependant, un int classique ne pourrait pas accéder aux valeurs excédant 2 147 483 647 sur une machine 32 bits, valeur d'adresse inférieure à notre valeur d'adresse maximale.
`value` : valeur contenue à l'adresse mémoire. Dans l'exercice, un mot est défini sur 4 octets, donc 32 bits. Comme pour les registres nous utilisons donc aussi un `int32_t`.
`suivant` : pointeur vers la prochaine valeur de la liste chaînée

Cela étant défini, nous pouvons créer la liste chaînée que nous utiliserons pour nos calculs :

```
static VRAM_bloc** VRAM_head;
```

Cependant, ici, il s'agit d'un pointeur vers pointeur de `VRAM_bloc`. En effet, étant donné que cette liste sera dynamique, contrairement à l'implémentation des registres, nous ne pouvons pas la déclarer de façon statique en mémoire, nous avons donc une fonction en plus pour l'initialiser (ci-dessous).

Fonctions

Nous avons définis 3 fonctions :

```
// init_VRAM
// Methode permettant l'initialisation de la VRAM
// (allocation du pointeur VRAM_head)
void init_VRAM();

//VRAM_load
//Permet de lire le contenu d'un bloc VRAM.
//Entrée : (uint32_t) adresse du bloc dans la memoire
//         (int) offset du bloc relatif à l'adresse
//         (int32_t*) pointeur vers la variable ou l'on stock le bloc lu
//Sortie : (Int), 0 = succes, 1 = echec
int VRAM_load(uint32_t address, int offset, int32_t* value);


//VRAM_store
//Permet d'ecrire le contenu d'un bloc VRAM.
//Entrée : (uint32_t) adresse du bloc dans la memoire
//         (int) offset du bloc relatif à l'adresse
//         (int32_t) contenu à ecrire dans le bloc
//Sortie : (Int), 0 = succes, 1 = echec
int VRAM_store(uint32_t address, int offset, int32_t value);
```

`init_VRAM()` nous permet simplement d'allouer un pointeur permettant l'initialisation de notre mémoire.
`VRAM_load` et `VRAM_store` retournent une valeur booléenne. Ici aussi, la valeur 1 sera retournée en cas de problème, notamment si la valeur d'adresse est supérieure à 4Go.
`address` est un `uint32_t` pour les mêmes raisons que celles évoquées précédemment.
l'offset est cependant un simple `int`, étant donné que nous pouvons avoir un offset négatif. Nos adresses étant définies par octet, nous avons aussi décidé de définir les offset par octet.

De plus, nous avons pris la décision de “recentrer” les adresses au mot par exemple, si nous avons pour adresse 1 et pour offset 1, nous devrions écrire à la valeur 2, qui se trouve au milieu d’un mot. On lit/écrit alors le mot complet, de l’adresse 0 à 3.
La fonction `VRAM_store` possède simplement un `int32_t` en argument, mais la fonction `VRAM_load` possède un pointeur vers `int32_t`. En effet, une fonction ne pouvant pas avoir deux paramètres en sortie, nous avons décidé de passer en argument un pointeur vers `int32_t` où on stockera la valeur contenue dans le registre choisi.

Remarque

Ici, la définition de notre mémoire comme liste chaînée présente un avantage important, en plus de sa facilité d’utilisation. Nous n’allouons alors pas 4 Go au lancement du programme. En effet, la mémoire est allouée dynamiquement. Nous ne lui affectons littéralement aucune mémoire lors de `init_VRAM()`.
A chaque fois que nous voulons ajouter une valeur en mémoire, deux cas se présentent à nous :

- Un bloc possédant l’adresse à laquelle nous souhaitons stocker une valeur existe dans la liste: Nous écrasons la donnée value de ce bloc. Aucun espace supplémentaire n' est requis.
- Sinon : Nous créons un nouveau bloc, que nous prenons soin d’insérer au bon endroit, dans notre liste chaînée (nous trions notre liste par valeur d’adresse mémoire).

Ainsi, seules les adresses dans lesquelles nous avons explicitement écrit sont définies dans notre liste chainée.