Projet Architecture des Ordinateurs

Jeu de Focus en MIPS

Tristan MICHEL
Kenza EL MHAMDI
Maxime LAFONT

2019-2020

Table des matières

Introduction

Pour ce projet final d'Architecture des Ordinateurs nous devions programmer un jeu en MIPS. Il s'agit du jeu Focus qui se joue à 2 utilisateurs, chacun possède des pions de formes différentes. L'objectif est d'empêcher le joueur adverse de pouvoir jouer, pour cela chaque joueur joue en tour par tour en choisisant soit de déplacer une pile de pièces qu'il possède ou de déposer des pièces de sa réserve s'il en possède. Ainsi le seul moyen de gagner est de posséder toutes les cases du jeu et que son adversaire n'ai pas de pièces en reserve.
En début de partie le plateau, une grille de 6x6, est rempli de pièces selon un motif particulier.

[+]	[+]	[∗]	[∗]	[+]	[+]
[∗]	[∗]	[+]	[+]	[∗]	[∗]
[+]	[+]	[∗]	[∗]	[+]	[+]
[∗]	[∗]	[+]	[+]	[∗]	[∗]
[+]	[+]	[∗]	[∗]	[+]	[+]
[∗]	[∗]	[+]	[+]	[∗]	[∗]

Les pièces [+] appartiennent au premier joueur alors que les [∗] sont au second. Au début du tour, le joueur choisit s'il veut déplacer ou déposer des pièces, dans notre cas le choix lui sera proposé seulement s'il peut effectuer les deux actions. S'il choisi de déplacer une pièce alors il choisi une cellule qu'il possède, indique le nombre de pièces qu'il veut déplacer parmi celles qui sont sur cette case. Puis il indique la direction dans laquelle il veut déplacer cette pile de pièces, ainsi on transpose cette pile sur celle de la case d'arrivée en limitant le nombre totale de pièces à 5 à la fin. Ainsi s'il y a plus de 5 pièces on compte les pions en trop qui appartiennent au joueur actuel et on les ajoutent à sa réserve, pour son adversaire les pièces en trop sont retirées du jeu.
Si le joueur choisit de déposer des pions alors il peut déposer autant de pions qu'il a dans sa réserve sur n'importe quelle case du plateau, biensûr ici aussi la limite est de 5 pièces ainsi le même mécanisme sera mis en place.

1 Mise en place du projet

Repertoire Github

Notre équipe étant formée de trois menbres, nous avons commencé par créer un répertoire git afin de pouvoir facilement éditer du code chacun de notre coté sans devoir se coordonner avant pour ne pas écraser l'avancement de quelqu'un d'autre. Ayant pas mal d'expérience avec Github, nous avons contruit ici notre projet. L'avantage était d'avoir une interface complète. En effet, on pouvait facilement voir l'avancement de chaque partie avec le nom des commit qui nous permettait d'indiquer quelle fonction nous avions codé. De plus puisqu'il s'agissait d'un jeu qui repose sur des interactions avec l'utilisateur, il est très important pour nous d'identifier les bug et de les corriger, grâce a Github nous avons pu facilement les inscrire en tant qu'*Issues* pour avoir une liste interactive des problèmes à regler.

Organisation du projet

Pour réaliser ce projet, nous avons commencé par analyser les différentes règles du jeu tout en réfléchissant à la manière dont nous allons tout mettre en place. Au moment de la conception du début de notre code, nous l'avons directement rédigé en MIPS. En voyant notre projet avancer, nous avons décidé de continuer sans passer par le language C. Afin que notre projet soit bien structuré et assez clair, nous avons réparti notre code dans différents fichiers. Chacun relatif à un certain aspect du jeu et regroupant différentes fonctions. Nous allons donc expliciter le contenu de chacun de ses fichiers :

Fichier main.asm

Ce premier fichier contiendra la fonction *main* qui représentera le déroulement du jeu. De plus nous avons fait une seconde fonction qui nous a servi d'environnement de développement pour tester nos nouvelles fonctions avec tous les paramètres possibles. Cet environnement peut être activé en décommantant une seule ligne ce qui facilite le développement de ce projet.

```
#j test             # decommenter pour tester des fonctions

    .globl main
main:
    # fonction main qui sera appelee au lancement du programme pour executer le jeu
```

Fichier fonctions.asm

Ce fichier va contenir l'ensemble des fonctions qui gèreront le contenu invisible du jeu. Que ce soit déplacer les pièces, gérer la réserve des joueurs ou encore mettre en place le plateau en début de partie. C'est aussi dans ce fichier que nous allons stocké nos variables de jeu, soit le plateau ou la réserve des joueurs.

```
    .globl plateau
plateau: .space 72  # stockage : un half pour une case soit 16 bits pour 5 pions =>
    pion sur 2 bit [01] 1 (+) / [10] 2 (*) / [00] vide
    .globl reserve
reserve: .byte 0, 0
```

Fichier user_interaction.asm

Comme son nom l'indique, celui-ci contient toutes les fonctions qui servent à intéragir avec l'utilisateur. Pour ce fichier la section *.data* servira a charger des phrases pour communiquer avec l'utilisateur. En comptant le nombre de caractères dans les phrases qui suivent nous pouvons ainsi nous déplacer entre elles pour choisir celle que nous voulons, ce qui nous permettra d'automatiser certaines tâches.

```
phrase_error: .asciiz "La valeur indiquee n'est pas valide, veuillez recommencer."
phrase_choix_action: .asciiz "Que voulez-vous faire (0: bouger une pile / 1: deposer des pieces) ? "

phrase_choix_direction: .asciiz ") : ", "Indiquer la direction souhaite ("  # 4 ascii : 5 octets pour la 1er string.
directions: .asciiz "^: 8", "<: 6", ">: 4", "v: 2"  # 4 ascii : 5 octets par string.
```

Fichier graphic.asm

Dans ce dernier fichier nous avons construit toutes fonction nécessaire à l'affichage console. Ces fonctions sont notamment *print_plateau* ou *print_reserve* qui affiche l'état actuel du jeu. Ainsi ici aussi le segment *.data* contiendra des chaines de caractère que nous afficherons dans la console.

```
affichage_case: .asciiz  "   ", "[+]", "[*]", "-->", "<--"  # 4 char ascii par string : 4 octets de decallage
affichage_grille: .asciiz "+-------------------+-------------------+-------------------+
+-------------------+-------------------+-------------------+"
```

Architecture des fonctions

Pour réaliser nos fonctions, nous avons utilisé la même mise en forme pour éviter le chaos, l'anarchie et eviter un soulevement de la population mondiale face a une telle incompréhension et faciliter la compréhension du code

```
1    # TODO
2    .globl fonction
3  fonction:     # $a0 = argument 1, $a1 = argument 2, $a2 = argument 3, $a3 = argument 4
4
5    # $v0 = valeur retourne numero 1
6    # $v1 = valeur retourne numero 2
7
8    sub $sp, $sp, 4      # move stack pointer
9    sw $ra, 0($sp)       # save $ra in stack
10
11   # Coeur de la fonction
12
13   lw $ra, 0($sp)       # get $ra from stack
14   add $sp, $sp, 4      # move stack pointer
15   jr $ra               # go back to caller
```

Ci-dessus vous avez un exemple d'une fonction lambda selon nos règles que nous nous sommes fixés après avoir fait des recherches sur la construction de fonctions en MIPS. Dans cette fonction, on commence par utiliser *.globl* si la fonction est dite publique et peut être appelé par une autre placé dans un autre fichier.
Après on note son nom avec un nom explicite pour indiquer son rôle, puis grace à de nombreux commentaires qui sont toujours placés aux mêmes endroits pour simplifier la lecture, nous indiquons les arguments et les valeurs retournés par cette fonction. Cela nous a donc permi d'avoir une progression asynchrone, à chaque fois qu'on avait besoin d'une fonction il suffisait d'ajouter cet entête avec le mot clé # TODO pour que cette fonction soit écrite par quelqu'un. Pendant ce temps la fonction supérieure pouvait être construite en sachant que la fonction sur laquelle elle repose correspondera parfaitement. Finalement il suffit d'ajouter les lignes 8 à 15 en écrivant notre fonction au milieu si la fonction que l'on contruit en appelle d'autres. Ce jeu d'instruction permet de sauvegarder facilement $ra qui nous permet de retourner à la fonction appelante.

2 Liste des fonctions intéressantes

Afin de faciliter la compréhension de notre code, nous avons établi une liste des fonctions les plus complexes et les plus intéressantes avec leur explication.

move_pieces | drop_pieces

La fonction move_pieces permet au joueur de déplacer une ou plusieurs pièces du jeu d'une case à une autre.

```
    .globl move_pieces
move_pieces:    # $a0 = num case depart, $a1 = num case arrive, $a2 = nb pieces, $a3 =
    num joueur

    # $v0 = le nb de pieces a ajouter a la reserve
    # $v1 = le num du joueur qui gagne les pieces en reserve
```

drop_pieces elle permet de placer des pièces de sa réserve dans une certaine case.

```
    .globl drop_pieces
drop_pieces:        # $a0 = case depot, $a1 = nb pieces, $a2 = num joueur

    # $v0 = le nb de pieces a retirer de la reserve
    # $v1 = le num du joueur
```

Ces deux fonctions intègrent le méchanisme qui limite le nombre de pièces dans chaque case à 5, et qui ajoute les pièces en excès à la reserve. C'est pour cela que ces fonctions retournent le nombre de piece de différence par rapport à la reserve actuelle pour pouvoir la mettre a jour en fonction de ce qu'il s'est passé lors de ce tour.

ask_player_action

Cette fonction est un pilier de notre jeu, elle va permetre de donner le choix de son action au joueur. Mais cela seulement si il peut choisir les deux options, sinon la fonction renvera le bon choix sans demander au joueur.

```
    .globl ask_player_action
ask_player_action:              # $a0 = num du joueur

    # $v0 = Choix du player : 0 pour move et 1 pour drop(de la reserve)
```

ask_player_direction_move

La fonction *ask_player_direction_move* gère l'interaction avec le joueur pour determiner la direction dans laquelle le joueur veut deplacer sa pile. L'interêt de cette fonction est son affichage dynamique qui indique directement à l'utilisateur les différentes directions possibles.

```
    .globl ask_player_direction_move
ask_player_direction_move:      # $a0 = coord case x, $a1 = coord case y, $a2 = nb
    pieces

    # $v0 = Renvoie la direction voulu pour deplacer la pile de pieces du joueur
```

print_plateau

Cette fonction permet d'afficher l'état du plateau de jeu en imprimant une grille de 6x6 remplie des différentes pièces et des flèches qui représentent les possibles choix des joueurs.

```
    .globl print_plateau
print_plateau:              # $a0 = case depart, $a1 = nb directions possible, $a2 =
    cases d'arrives encode sur 6 bit car case

    # Affiche le plateau de jeu avec les pieces qu'il contient
```

print_new_line

Une des fonction les plus petite que nous ayons, mais pourtant l'une des plus pratique, et surement celle que nous avons le plus utilisé est cette petite fonction qui ne fait rien d'autre que d'afficher un retour à la ligne dans la console. Mais l'implementation de cette fonction nous evite à chaque fois de recopier les 3 lignes qui la compose, ce qui au total a bien dû nous faire economiser une bonne centaine de ligne.

```
    .globl print_new_line
print_new_line:             # NULL

    # Affiche un saut de ligne (\n) dans la console
```

main

La fonction principale du jeu dans lequel on execute les différentes fonctions qui nous ont permis de construire le jeu. Dans cette partie nous ne ferrons que appeler les autres fonctions que nous avons précédament construit. Pour eviter toute perte de données et effacement de variables par l'appel d'autres fonctions, nous conservons toutes les données dans la pile ce qui nous permet de facilement les recuperer plus loin et même plusieurs fois, de manière sécurisé. Cette fonction se compose principalement d'une boucle qui sera exécuté tant que personne n'a gagné. Chaque tour commence par tester si un joueur à gagné puis si ce n'est pas le cas, on demande au joueur actuel l'action qu'il veut effectuer en fonction des cas on execute plusieurs fonction pour lui demander quelle case il veut jouer. Après cela on met à jour la reserve en fonction de ce qui a été joué. Finalement on change de joueur et on recommance.

Remarques

Pour ce projet nous avons cherché à créer le code le plus optimisé possible. Sans avoir fait de maquette en C, cela s'est avéré assez difficile, mais rien d'insurmontable. Nous avons ainsi utilisé les raccourcis MIPS avec le registre $at, ce qui nous a permis d'avoir un code beaucoup plus clair notament au niveau des branchement conditionels. Chaque fonction a dû être réfléchie afin d'être la plus efficace possible, de même notre manière de stocker des données à été optimisée pour prendre le moins de volume possible. Finalement réduire autant notre utilisation de mémoire nous a permis de voir de nouveaux algorithmes pour les fonctions que nous voulions qui nous ont encore permi d'optimiser le tout. Nous avons perdu un paquet de temps à tout optimiser mais cela nous a permit d'avoir un programme efficace, en travaillant sur language assembleur il s'agit d'une étape necessaire car ce programme pourrait servir de base pour un autre, ainsi il se doit d'être parfaitement efficace.

Conclusion

La réalisation de ce projet nous a permis non seulement d'assimiler les notions vues en cours mais aussi de les mettre en application. En effet, nous avons conçu le jeu entiérement en MIPS :

— Nous avons d'abord commencer par réfléchir à la possibilité de représenter chacune des variables qui rentre en jeu.

— Ensuite, nous avons pensé à définir les différentes fonctions dont nous aurons besoin.

— Pour finir, nous avons compléter chacune de nos fonctions puis réaliser notre fonction main qui nous a permis de tout tester.

Liens et références

- Cours sur les fonctions en MIPS (https ://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec05.pdf)
- Cours d'Architecture des Ordinateurs
- Liste des instructions MIPS