Université de Lille 1
Institut Universitaire de Technologie en informatique

Rapport de projet tuteuré
Tuteur : M. Michaël HAUSPIE

CodeWar

Grégoire SALINGUE - Steeve VANDECAPPELLE

Villeneuve d'Ascq, Année 2008-2009

Remerciements

Nous remercions M. Hauspie, notre tuteur qui a su à la fois nous laisser une grande marge de manoeuvre dans la conception de ce projet, et nous recadrer lorsque nous effectuions de mauvais choix techniques.

Nous remercions M. Chlebowsky qui nous a aidé à représenter notre conception du décodage d'un mot de la manière la plus claire possible.

Nous remercions les autres binômes du groupe M, qui, lors de discussions toujours plus animées et passionnantes autour de ce projet, nous ont aidé à mûrir nos idées.

Enfin nous remercions tous les relecteurs de ce rapport, qui auront largement contribué à ce qu'il soit de la meilleure qualité possible.

Résumé

Notre projet tuteuré réalisé lors du semestre 3 de notre formation, traite de CodeWar, un jeu mettant aux prises 256 processeurs virtuels. Ces 256 processeurs fonctionnent à partir de programmes assembleurs, soit codés par le joueur et traduit en binaire par un compilateur fourni par notre tuteur, soit codés par défaut. Pour programmer ce jeu, nous avons dû tout d'abord concevoir la dynamique de celui-ci, mais aussi le fonctionnement d'un processeur, pour terminer par l'interprétation du binaire assembleur en une série d'instructions exécutables.

Pour réaliser ce travail, nous avons dû nous organiser en équipe, mener des recherches sur le fonctionnement d'un processeur, ou encore sur l'utilisation du langage C. Cette expérience est avec le stage à venir, le point d'orgue de notre formation, et nous a posé des défis, tels que le simple fait d'apprendre à travailler ensemble, que nous nous sommes efforcés de relever.

Dans ce rapport, nous traiterons de cette expérience, en essayant de vous la faire partager, de la manière la plus claire possible.

Abstract

Our tuthored project realised during the third semester of our training is about CodeWar, a game opposing 256 virtual processors. Those 256 processors working from assembly programs, either coded by the player and translated to binary by a compilator provided by our tutor, or coded by default. To program this game, we had first to design it's dynamic, then the processor working process, to end by turning the assembly-bynary into an executable instructions serie.

To fulfill this work, we had to organize into a team, deal with some research about the processor working process, or about the C programming language. This work, is with the coming stage, the highlight of our training, and set us with problems, like the simple fact of learning how to work together, that we tried our best to face.

In this report, we'll deal with this experience, trying to share it with you, by the clearest way as it's possible to do.

Table des matières

Introduction

Notre projet tuteuré du Semestre 3 intitulé CodeWar est un jeu mettant aux prises 256 processeurs, faisant chacun fonctionner un programme assembleur, écrit ou non par l'utilisateur. Pour concevoir ce jeu, nous avons dû simuler le fonctionnement d'un processeur dont nous avions la documentation technique fournie par M. Hauspie.

L'une des difficultés de ce projet aura été de nous approprier cette documentation. En effet, bien que rédigée de manière à ce que le lecteur puisse finalement comprendre un sujet complexe, elle reste difficile à comprendre si elle n'est pas étudiée point par point.

Pour rédiger ce rapport, nous aurions pu nous poser les questions suivantes : comment simuler le comportement d'un microprocesseur ou encore comment interpréter un fichier binaire en un comportement bien défini ?

Cependant ce projet étant à la croisée d'un certain nombre de points techniques, nous sommes amenés à nous poser une question plus large : comment mener un développement de manière efficace autour d'une documentation technique ?

Dans le but d'y répondre, nous traiterons dans un premier temps de comment nous avons structuré notre travail, au niveau de l'organisation au niveau humain et conceptuel. Dans un second temps nous parlerons de ce qui a constitué la majeure partie de notre travail, la conception du processeur et de sa dynamique. Enfin, nous parlerons de la manière dont nous avons chercher à aboutir notre travail.

1 Bases du projet

Cette partie a pour but de décrire de quelle manière nous avons organisé notre équipe et la réponse que nous avons apportée à CodeWar.

Nous commencerons donc par vous présenter l'outil grâce auquel nous avons pu mettre en commun notre travail et autour duquel nous avons articulé tout le développement de ce projet.

Nous vous présenterons ensuite, pas à pas, le jeu et poserons les bases de notre développement, en définissant les structures de données que nous avons utilisées tout au long de ce projet.

1.1 Présentation de notre organisation

L'intérêt d'un tel projet étant d'apprendre à travailler et raisonner en équipe, nous avons organisé notre travail de manière à toujours pouvoir mieux travailler ensemble. Tout d'abord grâce à une communication efficace au sein de l'équipe, et même au sein du groupe du projet, ensuite grâce à l'utilisation d'un outil pour partager notre travail.

1.1.1 Communication au sein du groupe

Au cours de ce projet nous avons cherché à communiquer de manière efficace. Cette communication était facilitée par le fait que nous avions les mêmes emplois du temps, ce qui nous a permis de discuter de notre travail assez souvent lors de notre temps libre.

Même sans cet aspect des choses nous profitions de la séance de suivi de projet pour faire des points sur l'avancement du travail, et de nous fixer des objectifs à réaliser.

Nous avons aussi travaillé de concert avec certains autres binômes du groupe M, en nous exposant mutuellement notre vision des choses, et en débattant autour de la meilleure solution à adopter. Ce travail sur la communication nous a aidé à mieux comprendre les choses et n'a pas forcément eu pour résultante que nous proposions tous la même conception de solution.

En ce sens nous pensons que ce travail sur notre communication nous a tiré vers le haut nous et les autres binômes qui y ont participé, nous tenions donc à le préciser ici.

1.1.2 Subversion

Subversion est un outil de mise en commun du travail, qui a pour principe de faire stocker à l'équipe de développement toutes les modifications apportées au projet au fur et à mesure du développement de celui-ci. Ce stockage a ici été assuré sur un serveur mis à disposition par M. Hauspie.

Cet outil a plusieurs avantages, à commencer par celui d'offrir une sécurité non négligeable au développeur, comme nous l'avons tous deux expérimenté au cours de ce projet. En effet, nos deux machines étant tombées en panne tour à tour, le fait de stocker nos données sur un serveur distant, nous a permis de pouvoir les récupérer très facilement, ce qui n'aurait pas été aussi évident si nous

avions travaillé sans cet outil.

Outre l'aspect sécurité, Subversion nous a permis de développer plus facilement ce projet, en nous permettant de travailler par exemple en même temps sur un même fichier, ou tout simplement de pouvoir facilement récupérer une mise à jour, ou une ancienne version de notre programme.

Nous avons donc organisé notre développement autour de cet outil, de nos réunions hebdomadaires et de la documentation technique fournie par M. Hauspie.

1.2 Présentation du jeu

CodeWar est un jeu opposant des programmes assembleurs fonctionnant sur des processeurs dont nous allons expliciter le fonctionnement au cours de la partie 2 de ce rapport.

Les 256 processeurs, répartis sur une grille torique[1] de 16 x 16 cases, possèdent leur mémoire propre, et vont s'attaquer entre eux, en allant écrire dans leurs mémoires, de manière à aller prendre le contrôle d'un maximum d'autres processeurs.
Lors d'un tour de jeu, on considère que tous les programmes qui s'exécutent le font de manière simultanée, ce qui, comme nous allons le voir dans la partie 1.5 qui traite de la manière dont nous allons écrire dans la mémoire, va nous obliger à conserver un certain nombre d'informations.

Ce jeu oppose deux joueurs, qui vont (grâce au compilateur fourni par M. Hauspie) pouvoir charger en mémoire un programme assembleur sous forme de fichier binaire.

Le contrôle d'un processeur est matérialisé par un emplacement mémoire à aller écrire avec une valeur propre à chaque processeur : sa couleur. Ainsi, à la fin d'un nombre de tours de jeu déterminé par l'utilisateur, le processeur associé à la couleur la plus présente a gagné.

Voici par exemple un programme qui pourrait être chargé en mémoire par un joueur :

```
MOVE #1,R0    ; On met l'adresse mémoire de la couleur dans un registre
MOVE (R0),R1  ; On met le contenu de l'adresse contenue dans R0 dans R1
MOVE R1,(R0)  ; On met le contenu de R1 à l'adresse R0
              ; (Cette instruction est stockée en mémoire à l'adresse 18)
JMP #18       ; On saute à l'adresse 18
```

Cet exemple de programme représenterait un programme qui défend : il recopie en effet à chaque tour sa couleur à l'emplacement de celle-ci.

Le fonctionnement du jeu en lui-même est assez basique, mais il requiert qu'on crée des structures de données particulières, de manière à pouvoir accéder de manière efficace les 256 processeur du plateau de jeu. C'est ce que nous allons faire dans la section suivante, en définissant ce qu'est un processeur au niveau de la mémoire.

[1] Ce point sera traité dans la section 1.3.2 de cette partie

1.3 Définition du plateau de jeu

1.3.1 Architecture d'un processeur

Chaque processeur est un processeur 16 bits [2], ses registres[3] et les instructions qu'il lira en mémoire auront donc cette taille pour standard. Ils possèdent chacun 256 octets[4]de mémoire RAM et 9 registres.

Registres

Un processeur possède 8 registres, dans cette partie étant donné que nous nous intéressons uniquement à la représentation d'un processeur en mémoire, leur rôle ne nous intéresse pas ici, et sera traité dans la partie 2 du rapport, qui traite de la dynamique du processeur.

La seule chose qui nous intéresse ici est de savoir qu'ils existent, et qu'il faut les représenter en mémoire.

La taille d'un registre est de 16 bits, ce qui, dans le langage C (et sur les machines du département), correspond au type de données short int. Il est donc possible de regrouper les 8 registres au sein d'un tableau[5]de short int nommé r, de manière à pouvoir y accéder plus facilement. La déclaration de ce tableau en langage C est la suivante :

short int r[8] ;

Mémoire RAM

La mémoire d'un processeur est constituée de 256 octets de RAM. Dans le langage C, l'octet est représenté par le type char (charactère). Nous avons donc décidé de représenter ces 256 octets de mémoire RAM par un tableau de 256 char d'où la déclaration suivante :

unsigned char ram[256] ;

Structure processeur

La mémoire RAM et les registres appartenant à la même entité, un processeur, il est maintenant possible de se définir un type de donnée, dans lequel nous pouvons accéder aux informations précédemment définies. A noter la présence d'une entité nommée cycles, dont le rôle sera décrit dans la partie 2.1.2 traitant du timer.

```
typedef struct {
  unsigned short int r[9]; /* tableau de registres */
  unsigned char ram[256];  /* tableau de RAM       */
  unsigned int cycles ;
}processeur;
```

[2]bit : Unité élémentaire d'information pouvant prendre uniquement deux valeurs, le 0 ou le 1

[3]registre : entité qui va recevoir ou proposer des données suivant le contexte

[4]octet : suite de 8 bits

[5]tableau : type d'organisation qui permet de regrouper un ensemble de données au sein d'une même variable

1.3.2 Description du plateau

Le jeu se joue avec 256 processeurs, répartis sur une grille torique de 16 x 16 cases. La grille étant torique, le voisin du processeur de l'extrémité droite du plateau, sera celui de l'extrémité gauche. Ainsi, sur la grille représentée ci-dessous, le processeur A est voisin de B, et voisin de C. Le plateau ainsi défini n'a à proprement parler pas de limite.

A														C
B														

Chaque case du plateau représentant un processeur, nous avons défini un tableau à deux dimensions (cf glossaire) de processeurs pour représenter le plateau de jeu.

processeur[16][16] grille ;

Maintenant que nous avons défini tout ce dont nous avions besoin pour représenter les données de nos processeurs, le problème de pouvoir se repérer dans ces données se pose à nous. En effet sans système d'adressage des processeurs ou de la mémoire il est impossible d'aller lire ou écrire dans notre mémoire. Ce qui nous amène au sujet de la sous-partie suivante.

1.4 Construction des adresses

La mémoire du processeur et celle de ses voisins peuvent être adressées de deux manières :

- Par adressage direct, grâce à une adresse codée sur 16 bits. La description du mot représentant cette adresse est donnée en annexe A.1.
  Cette méthode permet à un processeur d'aller écrire dans la mémoire d'un processeur de la grille.
- Par un adressage via registre, codé sur 8 bits.
  Cette technique ne permet au processeur l'utilisant que d'adresser sa mémoire.

Partant de ce constat nous avons décidé de nous baser sur l'entité la plus complexe pour construire notre système d'adressage en mémoire : l'adressage direct.

Pour la représenter, nous avons besoin de regrouper deux systèmes d'adressage :

- Un adressage sur la grille de jeu via des coordonnées relatives x et y.
- Un adressage mémoire qui permette de cibler nos 256 octets de mémoire.

Nous avons donc défini une structure comprenant deux valeur x et y qui servent à définir les coordonnées du processeur sur la grille, et un unsigned[6]char permettant de stocker des entiers compris entre 0 et 255 pour adresser la mémoire. Ce qui nous donne la structure suivante :

```
typedef struct {
  unsigned char x,y;
  unsigned char adresse_memoire;
}adresse;
```

1.5 Gestion des écritures

Nous avons lors des sous parties précédentes vu la manière de laquelle nous représentions la grille de jeu, puis celle dont nous allions l'adresser. Nous allons maintenant voir comment nous allons aller modifier les données contenues dans les processeurs de la grille.

Il est important de préciser que lors d'un tour de jeu, nous devons simuler la simultanéité des actions des processeurs de la grille.

Nous commencerons par dire qu'il y a deux types d'écriture possible dans un processeur :

- Une écriture au sein d'un registre.
  Un processeur étant le seul à pouvoir écrire dans un registre on y accède de manière directe, et sans contrôle particulier.

- Une écriture dans la mémoire RAM.
  C'est le point que nous allons développer ici.

Listes d'adresses

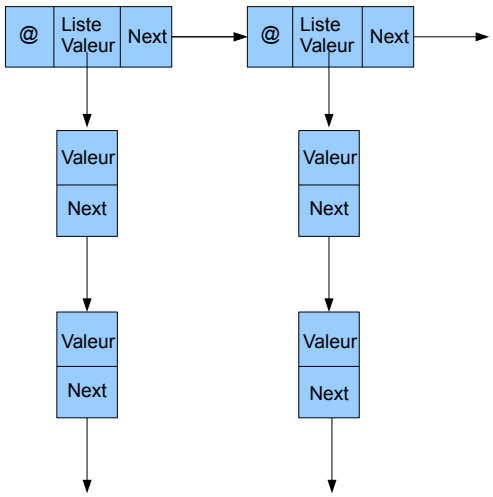
Lors d'un tour plusieurs processeurs peuvent chercher à aller écrire au même endroit en mémoire, ce qui génère un conflit. Pour gérer ce conflit nous avons dû sauvegarder toutes les écritures du tour courant avant de pouvoir les appliquer.

Pour organiser cette sauvegarde, nous avons raisonné à l'envers. En effet, en pensant à une écriture, on se dit qu'elle est associée à un processeur, donc à une adresse. Nous avions dans un premier

[6]unsigned : non signé

temps opté pour la solution d'une liste d'écritures, chacune associée à des adresses. Il est cependant plus efficace de gérer une liste d'adresses associée à une liste d'écritures.

En effet il est plus rapide de faire un parcours des valeurs à écrire par adresse, que de repérer les adresses, écriture par écriture, pour gérer un éventuel conflit. Nous avons donc adopté le mode de fonctionnement conforme au schéma suivant :



Ainsi à chaque fois qu'on cherche à ajouter une écriture, on suit la dynamique suivante :

Algorithme 1 Ajout d'une écriture à une adresse donnée

ENTRÉES: Liste d'adresse liste, Adresse ad, Entier valeur
SORTIES: liste d'adresse complétée

Si ad est déjà dans la liste **Alors**
  On ajoute valeur à la liste de valeurs associée à cette adresse
Sinon
  On ajoute l'adresse à la liste d'adresses
  On ajoute la valeur à écrire à la liste de valeurs associée à cette adresse
Fin Si
Retourner début de la liste d'adresse

L'application des écritures elle est un peu plus particulière. En effet lors de cette application nous allons devoir gérer les conflits dûs au fait que deux processeurs ou plus veulent écrire au même endroit en mémoire.

Nous allons ici considérer que les signaux électriques en essayant d'écrire au même endroit en mémoire vont s'additionner et donner une valeur à écrire qui sera composée de toutes les valeurs qui étaient destinées à être écrites à cette adresse. Ainsi, l'application des écritures se fera de la manière suivante :

Algorithme 2 Application des écritures d'un tour de jeu

ENTRÉES: Liste d'adresse liste
SORTIES: Rien

Tant Que Il reste des adresses dans lesquelles aller écrire **Faire**
On parcourt la liste des valeurs à écrire en les additionnant
On écrit la somme de ces valeurs à l'adresse courante
On passe à l'adresse suivante
Fin tant que

1.6 Dynamique du jeu

Après avoir défini l'ensemble de nos données et la manière d'accéder à celles-ci, nous pouvons, en passant sous silence le contenu de la partie 2 qui traitera de la dynamique du processeur, décrire de manière simple la dynamique du jeu. Le jeu n'est ni plus ni moins qu'une série de tours de jeu, qui se décomposent actuellement en deux temps :

- L'exécution du fonctionnement des 256 processeurs, qui va générer une liste d'adresses à aller écrire.
- L'écriture des valeurs associées à la liste d'adresses.

Conclusion partielle

Nous avons ici posé les bases de notre développement, en définissant les données que nous allions accéder, et la manière d'y accéder. Ceci a constitué chronologiquement la première partie de notre développement.

Nous allons maintenant aborder la seconde partie du projet, qui constitue à nos yeux la principale difficulté de celui-ci : la dynamique du processeur. En effet avant de pouvoir restituer sous forme d'un programme le fonctionnement d'un processeur, nous avons dû comprendre son fonctionnement.

Dans ce but nous avons travaillé encore une fois principalement avec la documentation fournie par M. Hauspie

2 Le coeur du projet : la dynamique du processeur

Lors de cette partie nous exposerons notre conception du fonctionnement d'un processeur. Dans cette optique, nous partirons du cas général, en vous présentant sa dynamique de fonctionnement, ou comment il recherche l'instruction à exécuter. Nous vous décrirons ensuite le décodage de cette instruction, et donc l'exécution de celle-ci.

2.1 Dynamique de fonctionnement du processeur

Le principe de fonctionnement d'un processeur est assez simple : exécuter des instructions[7]en fonction d'un contexte particulier.

Pour créer ce contexte il a à sa disposition trois outils :

- Les registres et adresses réservées qui servent à se repérer dans la mémoire, et activer certaines fonctionnalités.
- Le timer qui permet, s'il est actif de programmer certains traitements à effectuer.
- Les interruptions qui servent à gérer les évènements.

Nous allons vous les décrire dans cet ordre, puis nous vous décrirons la dynamique de fonctionnement d'un processeur.

2.1.1 Registres et adresses réservées

La dynamique du processeur est intimement liée à l'état de certains de ses registres, ou de certaines zones mémoires. Nous allons ici vous décrire ceux-ci.

Registres

Un processeur possède 8 registres, dont 3 sont primordiaux dans le fonctionnement de celui-ci, et 7 seulement sont accessibles au programmeur :

- 5 regitres sans rôle particulier de R0 à R5.
- Le registre de compteur de programme R6 (ou PC) qui indique au processeur la position de la prochaine instruction à lire en mémoire.
- Le registre de pile R7 (ou SP) qui indique au processeur la position en mémoire du sommet de la pile.

Un processeur possède aussi un autre registre, le registre d'état (ou RE), qui lui n'est pas accessible au programmeur. Ce registre sera modifié en fonction du résultat d'une opération. Seuls les 3 bits de poids faible (cf annexe) seront utilisés de la manière suivante :

- Le flag C (Carry flag), le flag de retenue, qui est mis à 1 lorsqu'une opération a généré une retenue.
- le flag Z (Zero flag), qui est mit à 1 si le résultat de l'opération est 0.
- le flag N (Negative flag), qui est mis à 1 lorsqu'une opération a généré un résultat négatif.

[7]La liste des instructions se trouve en annexe C

Ces informations seront utilisées par certaines commandes (les commandes dites “de saut”) qui modifieront leur comportement en fonction de l’état de ces flags.

Adresse réservées

Les adresses réservées sont des adresses qui vont être utilisées par le processeur pour remplir ou activer certaines fonctionnalités. Ainsi la couleur du processeur est stockée dans une adresse réservée, ainsi que les vecteurs d’interruption, qui permettent d’effectuer des traitements spéciaux en réaction à des évènements[8].

Le tableau des adresses réservées est contenu à l’annexe B

2.1.2 Timer

Le Timer est un outil permettant au processeur de programmer un traitement à une date donnée. Il peut être actif ou non, ce qui va modifier la manière de laquelle nous allons exécuter une instruction.

En effet, nous allons devoir contrôler dans la mémoire du processeur si il a été activé ou non, puis exécuter le comportement associé.

Si le Timer est actif, nous devons décompter un nombre de cycles qui permettront d’effectuer un traitement spécial (une interruption), contenu à l’adresse réservée du timer. S’il n’est pas actif, nous nous contentons d’exécuter la prochaine instruction écrite en mémoire.

2.1.3 Gestion des “évènements” ou interruptions

Au sein de cette partie nous allons traiter la manière qu’a le processeur de réagir face à certains évènements. Pour un processeur, un évènement se gère à l’aide d’interruptions.

La notion d’interruptions est d’ailleurs primordiale dans le fonctionnement de l’informatique telle que nous la connaissons actuellement, c’est elle qui permet de faire fonctionner plusieurs processus à la fois.

En effet, pour partager le temps entre les applications, le processeur va utiliser le timer, qui tout les tant de temps, va lancer une interruption. Cette interruption va être utilisée pour traiter la tâche suivante, c’est à dire par exemple traiter à un instant le fonctionnement du logiciel, et à celui d’après la réception d’un appui sur une touche du clavier.

Plusieurs types d’interruption

Mais revenons à nos simples mais non moins passionnants processeurs, qui eux, se limitent à 3 types d’interruption, qui seront levées dans certains cas de figure :

- Celles levées pour cause de décodage d’une instruction illégale
- Celles provoquées par l’expiration du Timer [9]

[8] cf partie 2.1.3

[9] Le timer sera traité dans la partie suivante

– Celles provoquées par l'instruction TRAP.

Comme nous l'avons vu dans la partie 2.1.1 qui traite des adresses réservées chaque type d'interruption est associé à un vecteur d'interruption. On peut donc assez aisément assimiler un type d'interruption à une adresse mémoire à aller retrouver dans un processeur.

En raison du déroulement du jeu, qui impose que les 256 processeurs fonctionnent en même temps, nous avons dû stocker les interruptions dans une liste d'adresses associées à une liste de types de vecteurs d'interruption.

Ces types se décomposent comme ceci :

– le type ILLEGAL : interruptions générées pour cause de décodage d'une instruction illégale.
– le type TIMER : interruptions générées pour cause d'expiration du Timer.
– le type TRAP : interruptions générées par l'instruction TRAP.

Cette conception étant en tout point similaire à la liste décrite dans la partie 1.5 qui traite de la gestion des écritures, nous avons réutilisé les mêmes structures de données, en changeant juste la manière de remplir la liste, et de l'appliquer.

Nous avons considéré qu'une interruption de type TRAP devait prendre le pas sur toute interruption qui aurait été générée jusque là. Lors de l'application des interruptions il va sans dire que celles qui n'auraient pas été appliquées à un processeur sont sauvegardées de manière à pouvoir être appliquées le tour suivant.

Nous utilisons donc les mêmes structures de données que pour gérer les écritures, mais n'appliquons ainsi pas tout à fait les mêmes méthodes. Le remplissage de la liste des interruptions se fera de la manière suivante.

Algorithme 3 Ajout d'une interruption à la liste d'interruptions

ENTRÉES: Liste d'adresses liste, Entier type
SORTIES: liste d'adresses complétée

Si l'adresse à laquelle l'interruption est levée n'est pas dans la liste **Alors**
  On ajoute l'adresse à la liste d'adresse
  et l'interruption à la liste d'interruptions associée à celle-ci
Sinon Si Le type de l'interruption est un TRAP **Alors**
  On ajoute cette interruption au début de la liste d'interruption
Sinon
  On ajoute cette interruption à la fin de la liste d'interruption
Fin Si
Retourner la liste d'adresse

Dynamique du processeur

A ce stade nous pouvons définir une dynamique de fonctionnement pour notre processeur, qui serait la suivante :

Algorithme 4 Dynamique de fonctionnement du processeur

ENTRÉES: processeur
SORTIES: rien

Si Le processeur n'a pas levé d'interruption **Alors**
  Si Le timer n'est pas activé **Alors**
    On exécute une instruction
  Sinon Si le timer n'est pas arrivé à sa fin **Alors**
    On exécute une instruction
    On incrémente le timer
  Sinon
    On lève une interruption de type TIMER
  Fin Si
Fin Si

Nous rappelons que lorsqu'un processeur a levé une interruption, celle-ci est appliquée au niveau du tour de jeu, il est donc normal de ne pas appliquer d'interruption lorsqu'un processeur a levé une interruption.

Maintenant que nous avons défini ce que fait notre processeur, nous pouvons maintenant étudier la manière dont il le fait : l'exécution d'une instruction.

2.2 Exécution d'une instruction

Nous allons maintenant considérer le processeur comme une machine à décoder et exécuter des instructions. Ce décodage aura deux résultantes possibles :

- La réussite et l'exécution du comportement de l'instruction décodée
- L'échec et le déclenchement d'une interruption de type ILLEGAL

Au cours de cette partie nous allons vous décrire comment est réalisé ce décodage. Nous commencerons par expliciter ce qu'est une instruction pour le processeur, puis nous vous présenterons notre conception de ce décodage.

Enfin, en essayant de rentrer un minimum dans le code, nous vous décrirons la manière de laquelle nous avons représenté les commandes, puis les opérandes en machine.

2.2.1 Représentation des instructions en machine

Comme nous l'avons dit dans la partie 1.3.1, notre processeur fonctionne sur une architecture 16 bits. Ceci a pour conséquence que la majorité des instructions qui seront représentées en mémoire,

ainsi que ses registres, auront une taille de 16 bits.

Le langage que nous interprétons est l'assembleur, dont la syntaxe se présente sous la forme suivante :

Instruction Opérande source, Opérande destination

Le programme du programmeur, avant compilation, aura donc un aspect semblable à ce shéma. Après compilation, ce programme sera agencé sous forme de mots binaires structurés de manière à pouvoir être interprétés facilement par la machine, ce qui est moins le cas pour nous autres humains.

La forme des mots à interpréter peut varier selon le type de la commande, il existe 4 types de mots à interpréter :

- Les mots représentant les commandes à 0 opérande
- Les mots représentant les commandes à 1 opérande
- Les mots représentant les commandes à 2 opérandes
- Les mots représentant les MOVE (qui ont eux, une taille de 32 bits)

La description des différentes formes de mots se trouve en annexe à la section A des annexes.

2.2.2 Conception

Pour vous présenter notre conception des choses, nous allons nous appuyer partiellement sur un point de mathématiques vu ce semestre dans le cours de M. Chlebowsky : la théorie des langages.

Pour ceci nous sommes partis du principe que nous disposions de deux grammaires[10] :

- La grammaire des commandes (cf annexe C)
- La grammaire des opérandes (cf annexe D)

Ces deux grammaires forment le langage que nous avons utilisé pour le décodage d'un mot. Chaque commande possédant un mode d'adressage (soit une liste d'opérandes autorisées, nous avons pu regrouper nos commandes de la manière suivante :

Type de commande	Mode adressage source autorisé	Mode adressage destination autorisé
0 Opérande	-	-
1 Opérande	R_N,$-(R_n)$,(R_n),$(R_n)+$	R_N,$-(R_n)$,(R_n),$(R_n)+$
1 Opérande bis	R_N,$-(R_n)$,(R_n),$(R_n)+$,#x	R_N,$-(R_n)$,(R_n),$(R_n)+$,#x
2 opérandess	R_N,$-(R_n)$,(R_n),$(R_n)+$,#x	R_n
MOVE	R_N,$-(R_n)$,(R_n),$(R_n)+$ ou #x,@x	R_N,$-(R_n)$,(R_n),$(R_n)+$, @x ou R_N,$-(R_n)$,(R_n),$(R_n)+$

Si nous construisons l'automate[11]associé à ce langage et au comportement décrit dans le tableau ci-dessus : nous obtiendrons le schéma de l'annexe E qui représente tous les chemins qui mènent à

[10]Grammaire : Une grammaire est un ensemble de mots

une instruction valide.

Imaginons que le mot que nous venons de recevoir est un PUSH (une commande du type 1 opérande). Selon l'automate que nous venons d'établir, si le type de l'opérande source codée dans le mot appartient à la liste des opérandes autorisées pour ce type d'instruction, on valide le PUSH. Sinon, on lève une interruption.

Cette solution a les avantages suivants :

- C'est la commande qui "sait" ce dont elle a besoin pour fonctionner.
- Il est possible de récupérer la valeur de l'opérande en même temps qu'on procède au contrôle sur la validité de celle-ci

Nous payons par contre ces avantages par un grand nombre de fonctions faisant à peu près la même chose au niveau du décodage de l'opérande. Cet aspect est l'une des parties de notre programme que nous pourrions envisager d'améliorer si nous poursuivions ce projet. Nous développerons plus avant ce point dans la section 2.2.4 qui traite du décodage des opérandes.

2.2.3 Décodage d'un mot : les commandes

Comportement d'une commande

Avant de parler du décodage du mot en lui-même, il est important de donner notre définition de ce qu'est une commande :

Une commande est une fonction[12]qui va décrire un comportement d'un processeur de la grille

Ce comportement pourrait être défini de la manière suivante :

Algorithme 5 Dynamique d'une commande

ENTRÉES: mot
SORTIES: rien
Récupération des opérandes codées dans le mot
Si opérandes récupérées avec succés **Alors**
Exécution du comportement de la commande
décrit dans la documentation technique de M. Hauspie) soit :
- Génération d'une écriture mémoire
- ou écriture dans un registre
- Mise à jour du registre d'état suivant le type de la commande (cf à faire ...)
- Incrémentation du compteur de programme PC suivant le type de la commande (de la taille de deux mots si la commande est un MOVE, d'un sinon)

Fin Si

[11] Automate : Méthode pour déterminer si un mot appartient à un langage ou non
[12] fonction : ensemble d'instructions réalisant une certaine tâche

Décodage d'une commande

Les fonctions représentant les commandes peuvent donc être représentées par la ligne suivante :

void ma_commande (processeur grille[][16], adresse *ad) ;

void signifie que la fonction ne va pas retourner de données à l'entité qui l'appelle
ma_commande est le nom de la commande à appeler
processeur grille[][16] est la grille du jeu
adresse *ad est un pointeur[13]vers une structure du type adresse

Après avoir défini ce qu'était une commande pour le programme, nous devons pour décoder un mot, commencer par décoder la commande que ce mot va appeler. En effet, n'importe quel mot aura la structure suivante :

c	c	c	c	c	r	r	r	r	r	r	r	r	r	r	r

c : mot codé sur 5 bits représentant le numéro de la commande à appeler
r : reste du mot

Il est donc assez facile d'isoler un numéro de commande grâce à des décalages de bits et des masquages[14]de la manière suivante :

```
short int numero_commande = mot>>11;       /*on décale le mot de 11 bits
                                             vers la droite*/
numero_commande = numero_commande&0x001F; /*on ne conserve que les 5 premiers
                                            bits du mot obtenu en les masquant*/
```

Grâce à cette méthode nous obtenons le mot suivant qui nous permet de récupérer le numéro de la commande à décoder :

0	0	0	0	0	0	0	0	0	0	0	c	c	c	c	c

c : mot codé sur 5 bits représentant le numéro de la commande à appeler

Pour décoder ce numéro de commande après l'avoir isolé, nous avons dans un premier temps par soucis de performance pris le parti de tester le mot bits par bit. Cette solution était plus performante qu'un switch (cf glossaire), car elle nous permettait de décoder la commande en maximum 5 tests, elle était cependant tellement peu claire que nous avons préféré, sur les conseils de M. Hauspie, changer cette partie de notre conception.

Nous nous sommes alors tournés vers les pointeurs de fonction[15], qui nous ont permis de "stocker" des accès à nos fonctions dans un tableau.

[13]pointeur : type de donnée contenant une adresse mémoire
[14]Masquage : Technique permettant d'isoler certaines valeurs d'un mot pour ne conserver qu'un résultat voulu
[15]pointeur de fonction : Un pointeur de fonction est un type de variable permettant de stocker l'adresse mémoire d'une fonction

En effet comme nous l'avons vu au début de la partie, toutes les fonctions vont avoir la même signature, nous allons donc pouvoir représenter les commandes par un type commande qui serait le suivant :

```
typedef void (* commandes)(processeur liste_processeur[][16],adresse *ad) ;
```

Outre l'aspect très technique qui nous a incité à opter pour ce type de solution, cette représentation des commandes nous a permis de stocker les références à nos fonctions dans un tableau. Plus clairement, ainsi indéxées, les fonctions peuvent être appelées directement, sans test, à l'aide du numéro de commande que nous avons extrait plus haut. En effet la syntaxe suivante :

```
processeur liste_processeur[16][16] ;
adresse ad ;
tableau_de_commandes[numero_commande](liste_processeur,&ad) ;
```

Appelle la fonction de commande numéro **numero_commande** avec les arguments **liste_processeur** et **&ad**. A noter que les numéros de commande sont notés dans la partie C de l'annexe.

2.2.4 Décodage d'un mot : les opérandes

Comportement d'une fonction opérande :

Comme nous l'avons vu précédemment pour fonctionner, une commande va devoir récupérer la valeur des opérandes qui lui sont associées. Chaque type de commande (cf tableau de la section 2.2.2), va être associée à une fonction de récupération de ses opérandes. Celle-ci aura deux fonctions :

- Réaliser le contrôle sur le type des opérandes.
- Récupérer la valeur ou l'adresse des opérandes.

Le comportement de ces fonctions serait le suivant :

Algorithme 6 Comportement des fonctions Opérandes

Si les opérandes à rapatrier sont valides **Alors**
  On rapatrie la valeur (ou l'adresse) des opérandes quel qu'en soit le type
Sinon
  On génère une interruption pour cause d'instruction illégale
Fin Si

Décodage des opérandes

Pour décoder les types d'opérandes, nous avons adopté la même technique que celle adoptée pour les instructions : les pointeurs de fonction.

En effet un type d'opérande est associé à un mot dans lequel est contenu sa valeur, et à un processeur, qui sera modifié en fonction du type de l'opérande (registre pré et post-décrémenté), ce qui nous donne le type de fonction suivant :

typedef unsigned short int (*operande)(unsigned short int mot,processeur *p) ;

Les fonctions de ce type sont stockées dans deux tableaux de fonctions différents :

- Le tableau des opérandes source qui retournera la valeur de l'opérande
- Le tableau des opérandes destination qui retournera l'adresse mémoire de l'opérande à aller écrire

Avantages et limites de la solution adoptée

Lors d'un appel à une fonction de récupération d'opérande, le rapatriement de la valeur ou de l'adresse des opérandes se fera en appelant la fonction correspondant au type d'opérande.

Cette manière de réaliser les choses nous permet au niveau de la commande de nous abstraire totalement du type de l'opérande. Au niveau des instructions, tout ce qu'on veut savoir c'est si on récupère une valeur que l'on va traiter, ou une adresse à laquelle on va aller écrire, ce que cette vision des choses rend possible. En ce sens, cette solution est bonne.

La limite vient du fait qu'il y a beaucoup de fonctions pour finalement à chaque fois quasiment le même traitement. Ce grand nombre de fonctions nous a "forcés" à réutiliser les pointeurs de fonctions pour rapatrier la valeur ou l'adresse des opérandes. Ce n'est pas forcément la manière la plus efficace de travailler étant donné que ces fonctions ne font pas plus de deux lignes. Quand on sait qu'un appel à une fonction, au niveau physique, c'est un stockage de plusieurs données en mémoire, on se rend compte que cette manière de faire, c'est beaucoup de temps perdu, pour finalement pas grand chose.

Ainsi si nous avions à poursuivre ce projet, nous essaierions de ne faire que deux fonctions (pour la source et la destination) qui fonctionneraient pour tous les types de commande, ce qui serait beaucoup plus efficace, en terme de clarté pour le programmeur, et de temps d'exécution pour la machine.

Conclusion partielle

Nous avons lors de cette partie fini d'expliciter le fonctionnement du coeur du projet tel qu'il nous était demandé. A ce stade, le jeu peut fonctionner en mode texte.

Nous avons abordé cette conception en deux temps, nous d'abord cherché à comprendre ce qu'était un processeur, avant de concevoir son fonctionnement. Nous l'avons vu cette conception n'est pas parfaite, elle a cependant le mérite de répondre, nous espérons assez clairement, à un problème somme toutes assez complexe.

Après avoir terminé ce projet, nous avons manifesté la volonté d'aller plus loin dans la logique de celui-ci. Ce qui nous amène à notre ultime partie, qui traite de ce que nous avons fait pour rendre notre programme plus accessible.

3 Les aspects additionnels

Pour traiter cette partie de notre développement, nous sommes partis dans l'optique de rendre notre projet plus accessibles à tous les utilisateurs, quel que soit leur niveau en informatique, et qu'ils utilisent un système d'exploitation ou de type UNIX, ou windows.

Dans ce but nous avons développé une petite interface graphique pour faire fonctionner notre notre projet. Notre développement ayant été réalisé uniquement sur une plateforme de type UNIX, nous avons intégré de la cross-compilation[16]à notre programme, pour qu'un utilisateur employant un système d'exploitation de type Windows puisse compiler ce projet sans problèmes.
Nous traiterons ainsi dans un premier temps de la conception de cette interface graphique, puis nous présenterons les outils et méthodes utilisées pour réaliser notre cross compilation.

3.1 Interface graphique

Avant de nous lancer dans l'interface graphique en langage C, il a fallu nous poser des questions auxquelles le sujet ne demandait pas de répondre. Pour concevoir cette interface nous avons dû modifier en partie certains bouts de notre programme initial. Dans le but de réduire ce travail de refonte, nous avons cherché à concevoir cette interface graphique de manière à la dissocier au maximum du modèle c'est à dire le jeu et ses règles.

Le principe a dès lors été simple : enlever les affichages que nous avions programmé dans la version textuelle de notre jeu, et modifier certains passages de notre code initial, de manière à ce que le modèle puisse communiquer avec la vue et le contrôleur (que sont en fait l'interface graphique).

Nous avons de plus voulu gérer les entrées d'informations nécessaire au chargement des deux processeurs : programmes binaires, couleurs, positions sur la grille.

3.1.1 Le choix de GTK+

Nous nous étions renseignés, sur les différentes bibliothèques graphiques existant en langage C, et deux d'entre elles nous avaient intéressé : GTK+, et SDL.

Après quelques réflexions sur le sujet, lectures de forums traitant de ce sujet, et consultation de notre tuteur de projet, nous avons choisi GTK+.

GTK+ est un ensemble de librairies écrites en C et développées pour le logiciel de dessin GNU GIMP (GIMP Tool Kit). GTK est, entre autres, au coeur de GNOME, un environnement graphique utilisé sous Linux.

[16]Cross-compilation : méthode de programmation permettant de compiler un programme sous différentes formes d'architectures

```
Librairies Disponibles pour GTK+ :
Glib : Permet d'accéder à divers outils nécessaires à la programmation en GTK+ ;
GDK  : Permet la programmation bas niveau de dessin sur les fenêtres
       (modification de la couleur, changement de police de caractères) ;
GTK  : Donne accès aux objets graphiques (widgets) permettant la programmation
       de l'interface utilisateur (boutons, fenêtres).
```

3.1.2 Le choix de Glade

Nous avions d'abord essayé de programmer notre interface uniquement a l'aide de GTK+, cela posait cependant plusieurs problèmes :

- La programmation d'une interface graphique comporte un grand nombre d'obets graphique, ce qui est très long à coder.
- Nous ne maîtrisions pas assez GTK+ pour arriver à un résultat concluant rapidement par nous même

Tenant compte du fait que nous n'arriverions pas à finir cet aspect du projet si nous nous entêtions dans cet voie, nous avons changé de méthode, et nous sommes tournés vers un utilitaire nommé Glade.

Glade permet de générer facilement le squelette de l'interface graphique, et de le ranger dans un fichier XML. Nous avons pu lire ce fichier grâce à une librairie nommée "libglade" qui permet de lire dynamiquement dans ce fichier. En simplifiant le fonctionnement de la libglade, on pourrait dire que le fichier XML est simplement lu, et les éléments graphiques contenus à l'intérieur sont mis en mémoire pour pouvoir être accédés par le programmeur.

Grâce à cette méthode nous avons réduit considérablement la quantité de code à produire, ainsi que la difficulté associée au fait de devoir maîtriser la partie génération de l'interface graphique de GTK, ne nous restait "plus" qu'à arriver à récupérer les éléments graphique de notre fenêtre, ainsi que les évènements qui peuvent leur être associés et mettre à jour l'interface graphique, pour que nous ayons un programme dynamique.

3.1.3 CodeWar en mode graphique

Exploitation du code XML généré par Glade

Comme nous l'avons dit précédemment la création des éléments graphiques via Glade était faite à part de notre programme. Il nous fallait donc un moyen de communiquer à celui-ci les éléments dont nous avions besoin, et en lancer l'affichage.

Pour cela nous avons utilisé un objet nommé Builder, dont le but etait simple, faire la liaison entre le code XML et le Code C. Une fois le Builder relié au fichier XML, une simple fonction nous permet à l'aide de son nom et du constructeur qui lui correspond de récupérer un élément précis présent dans la fenêtre graphique conçue à l'aide de Glade. Il n'y a plus qu'à se concentrer sur l'essentiel : les évenements qui se produisent quand on clique sur tel ou tel composant.

Les Signaux

Le principe est plutôt simple à comprendre : une application de type GUI[17] réagit à partir d'évènements, c'est ce qui rend une fenêtre dynamique. Il s'agit ici de gérer des évènements : interception des touches du clavier, des déplacements de la souris, du joystick, clique sur un composant.

Cette gestion c'est faite à l'aide de deux éléments simples, le signal qui peut être assimilé à un évènement (par exemple un clique de souris), et le slot qui est la fonction appelée lorsqu'un évènement c'est produit (par exemple la fonction de fin de programme).

Ainsi pour rendre notre interface graphique, nous avons dû connecter des évènements à des fonctions particulières, qui décriraient le comportement du programme lors de l'évènement déclencheur (par exemple afficher une autre fenêtre pour effectuer une saisie préliminaire, ou encore démarrer le jeu).

Intégration graphique sur le coeur du programme

Le plateau de Jeu

Pour modéliser le plateau de jeu nous nous somme demandés comment représenter au mieux les processeurs, nous avons choisi de les représenter par leur couleur, qui était contenue à l'adresse mémoire 1, cette valeur représentant l'appartenance d'un processeur, à tel ou tel autre joueur.

Codé sur 15 bits cette couleur devait nous servir à créer un composant graphique intégrant une couleur, nous avons d'un commun accord décidé de faire appel aux GtkColorButton, dans le but d'une future mise à jour qui pourrait éventuellement permettre de traiter N processeurs avec N programmes à charger.

Chargement des données

Le chargement des paramètres nécéssaire à CodeWar sont faits au fur et à mesure de l'apparition de nos fenêtre. Le but construire pas à pas les informations voulue.

- Couleur, coordonnées, programmes binaires des processeurs.
- Nombres de tours de jeu.

Ces paramètres graphiques sont transmis via une structure que nous avons créé, ceci a cause d'une limite de GTK+ qui interdit de passer plus de un paramètre aux slots. Les signatures des slots de signaux GTK+, sont en effet du type :

void <nom_de_fonction> (ObjetGraphique, paramètre) ;

Où Objet graphique représente l'objet connecté à la fonction (passé en paramètre automatiquement par GTK+), et paramètre le paramètre que vous souhaiter passer(voir annexe pour le raccord

[17] Application GUI : Application offrant une interface graphique à l'utilisateur

de la fonction au signal du composant graphique). Nous avons donc créé une structure de données permettant de faire le lien entre toutes les fonctions, en utilisant un seul paramètre. dont voici la signature :

```
typedef struct demarage{
   coordonnees xy;          /*Coordonnées du processeur sur le plateau de jeu*/
   unsigned int couleur;    /*Couleur du processeur (variable à chargé dans RAM[1])*/
   gchar *cheminAssembleur; /*Chemin du fichier binaire à chargé dans le processeur*/
}demarage;
```

Mise à jour du plateau

Pour la mise à jour du plateau nous avons été confrontés à deux petit problèmes, que nous avons résolus avec un peu de réfléxion, de recherches, à l'aide de nos discussion inter-groupe ainsi que de notre tuteur.

Il faut savoir que les GtkColorButton utilisent 3 paramètres qui nous intéressent pour afficher une couleur : une composante de rouge, de vert, et de bleu chacune de ces composante étant en fait un entier compris entre 0 et 65535. Or le soucis, est que nous enregistrons une valeur comprise entre 0 et 255 pour la totalité du code de couleur (correspondant à 5 bits par couleur, soit 15 bits au total = 1 bit non utilisé).

Il nous fallait d'une part, un moyen de passer d'un entier à une composante de 3 entiers, et d'autre part, un moyen de passer d'un entier codé sur 5 bits a un entier codé sur 16 bits et vis-versa. Nous nous somme ainsi penchés sur une structure de type "champ de bits" (cf Annexe F) jointe à un entier. Cette structure nous permet donc de selon notre besoin de prendre la valeur de l'entier ou la valeur des 3 champs rouges vert ou bleu.

```
typedef{
 union{
   unsigned short int tout ;
   struct{
   unsigned int x : 1 ;
   unsigned int r : 5 ;
   unsigned int v : 5 ;
   unsigned int b : 5 ;
 } composed ;
}couleur_processeur;
```

Ainsi lorsque l'on rentre une valeur dans l'entier "tout" la valeur de chacuns des champs définis précédement changent.

Pour la mise à jour du plateau, il ne restait plus qu'à créer une fonction qui permet de relancer l'affichage des composants graphiques de la fenêtre principale.

Nous terminerons cette partie en vous renvoyant à l'annexe G.5 qui présente un aperçu de notre application.

3.2 La compilation croisée

Pour que notre code soit compilable sur plusieurs architectures différentes, nous avons dû jouer sur l'étape préprocesseur de la compilation.

Il n'y a dans notre programme que peu d'instructions qui posent des problèmes de compatibilité entre Windows et Linux, nous les avons donc encadré dans des blocs de macros de ce type

```
#ifndef WIN32
   /*instructions qui compileront sous systèmes UNIX*/
#else
   /*instructions qui compileront sous système Windows*/
#endif
```

Cette manière de faire nous a permit de différencier les codes qui pouvaient poser des problèmes de compatibilité. Il nous est cependant resté un problème plus épineux, celui du Makefile et de la compilation avec GTK sous windows. En effet la version texte de notre programme compile avec cette méthode sans problème sous les deux architectures sur lesquelles nous avons voulu le rendre compilable : UNIX et WIN32 (linux et windows). La version graphique a posé quand à elle quelques problèmes étant donné qu'il faut bien préciser au compilateur ou aller chercher les bibliothèques de GTK.

Ce problème nous a amené à vouloir utiliser l'utilitaire CMake qui permet de générer des Makefile en fonction de l'architectures et des bibliothèques utilisées. Nous n'avons cependant pas eu le temps nécessaire pour arriver à utiliser cet outil. Cet aspect vient s'ajouter à la liste des choses que nous aurions bien aimé faire si nous avions à poursuivre ce projet.

Conclusion partielle

En conclusion de cette partie nous pouvons noter que pour réaliser cette partie de notre travail, nous avons dû nous remettre en question. Tout d'abord en cherchant à se rapprocher des besoins de l'utilisateur, qui sont que le programme fonctionne sur son système d'exploitation préféré, et qu'il soit facile à prendre en main.

Nous avons ensuite dû chercher les outils qui nous permettraient de satisfaires ces besoin, et apprendre à les utiliser, la plupart du temps par nous même. Cette partie du travail, qui symbolise bien le fait que le développeur ne doit pas rester sur ses acquis mais toujours chercher à faire évoluer son bagage technique, était un réel défi.

Conclusion

En conclusion de ce rapport, qui représente l'aboutissement d'un semestre de travail, et en quelques sortes de nos deux ans de formation à l'IUT, nous pouvons dire qu'il est une bonne synthèse de notre formation, et peut nous éclairer sur ce qui nous attend au cours de notre carrière de développeur.

En effet, développer, c'est structurer les choses, tout comme nous l'avons fait dans la première partie de notre travail. C'est structurer une équipe, en s'organisant, en communiquant en utilisant toutes les ressources disponibles, qu'elles soient humaines ou matérielles. C'est structurer une conception, en posant des bases simples et abstraites à un raisonnement toujours plus concret mais difficile à mettre en place.

Développer, c'est analyser les choses, de manière poussée, en utilisant une documentation, en comprenant des fonctionnements, de quelques type qu'ils soient. Si ce projet avait porté sur le fonctionnement d'un autre objet que le processeur, nous n'aurions pas adopté une démarche différente : d'abord comprendre puis formaliser les choses pour ensuite les restituer sous forme d'une conception, et enfin, d'un programme.

Développer c'est une perpétuelle remise en question, c'est être à l'écoute des besoins de l'utilisateur final, d'un environnement, qui peut inspirer une idée, et débloquer une situation verrouillée ou encore imposer de nouvelles contraintes qui changeront notre manière de concevoir les choses. C'est se demander quel outil utiliser, et comment l'utiliser. C'est enfin rester critique vis à vis du travail effectué et toujours se demander comment mieux faire.

Nous sommes satisfaits de ce travail, qui, s'il n'est malheureusement pas parfait, nous a permis d'apprendre un certain nombre de choses sur nous et notre manière de travailler. Cette expérience nous sera certainement très utile au cours de notre stage, et à plus long terme au début de notre vie professionnelle.

Annexes

A Types de mots

A.1 Mot représentant un adressage direct

L'octet de poids fort (b_{15} à b_8) représentant les coordonnées relatives du processeur cible sur le plateau de jeu, l'octet de poids faible (b_7 à b_0), lui, représentant l'adresse à aller accéder en mémoire. Le mot représentant l'adresse se décompose de la manière suivante :

b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
X	X	X	X	Y	Y	Y	Y	@	@	@	@	@	@	@	@

A.2 Mot représentant une instruction à 0 opérande

Le format d'une instruction à 0 opérande est le suivant :

c	c	c	c	c	0	0	0	0	0	0	0	0	0	0	0

Les 5 bits de poids fort (c) représentent le code de l'instruction les autres bits sont mis à 0.

A.3 Mot représentant une instruction à 1 opérande

Le format d'une instruction à 1 opérande est le suivant :

c	c	c	c	c	t	t	t	v	v	v	v	v	v	v	v

Les 5 bits de poids fort (c) représentent le code de l'instruction.
Les 3 bits suivants (t) représentent le type de l'opérande
Les 8 bits de poids faible désignent en fonction du type de l'opérande à représenter :

A.4 Mot représentant une instruction à 2 opérandes

Le format d'une instruction à deux opérandes est le suivant :

c	c	c	c	c	r	r	r	t	t	t	v	v	v	v	v

Les 5 bits de poids fort (c) représentent le code de l'instruction
Les 3 bits suivants (r) sont le niveau du registre de destination
Les 3 bits suivants (t) représentent le type de la source
Les 5 bits suivants (v) sont la valeur de l'opérande. En fonction du type, ces bits désignent :

- Le numéro du registre entre 0 et 7
- La valeur immédiate

A.5 Mot représentant un MOVE

L'instruction MOVE, permettant des choses plus complexes nécessite 32 bits, soit 4 octets pour être codée

Mot 1	c	c	c	c	c	h	l	t1	t1	t1	t2	t2	t2	r	r	r
Mot 2	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v

Les 5 bits de poids représentent le code de l'instruction.
Les deux bits suivant (h et l) permettent de différencier un MOVE, un MOVE.l, et un MOVE.h. Pour un MOVE les deux bits sont à 1, pour un MOVE.l le bit l, pour le MOVE.h le bit h.
Les 3 bits suivants définissent le type de la source
Les 3 bits suivants définissent le type de la destination
Les 3 derniers bits du premier mot servent à coder un numéro de registre qui sera utilisé soit par l'opérande source, soit par l'opérande destination
L'intégralité du second mot représente une valeur qui est utilisée soit en source, soit en destination suivant le contexte.

B Tableau des adresses réservées

Adresse 8 bits	Taille	Signification
@1	16 bits	Couleur du processeur. Les 16 bits contiennent les composantes rouge, vert et bleu. 5 bits par composante : xrrrrrvvvvvbbbbb.
@3 à @9	-	Vecteurs d'interruptions
@3	8 bits	Adresse de la routine d'interruption d'instruction illégale.
@4	8 bits	Adresse de la routine d'interruption du timer.
@5	8 bits	Adresse de la routine d'interruption générée par un autre processeur.
@A à @D	32bits	Configuration du timer

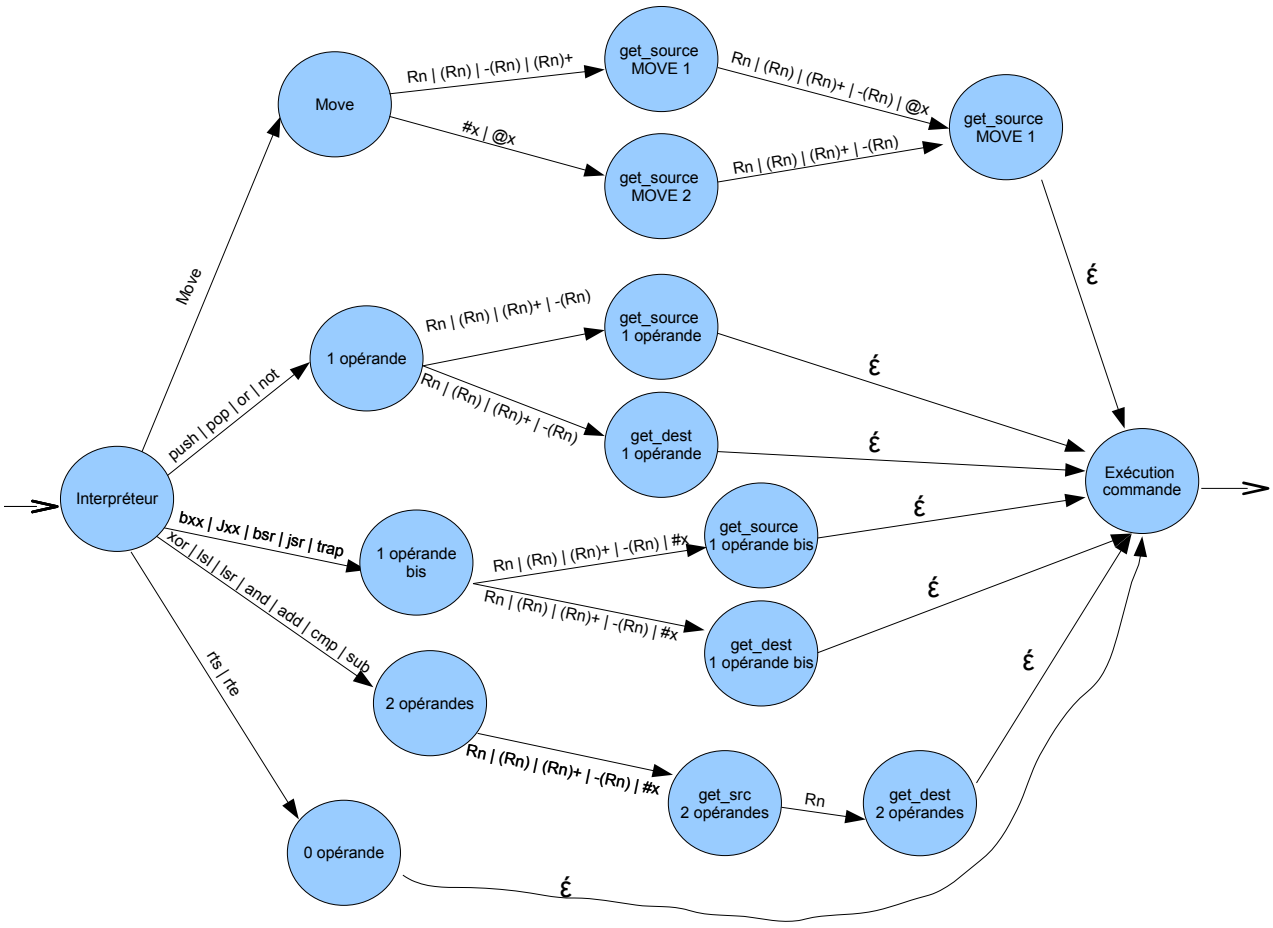
C Liste des instructions ou Grammaire des commandes

Instruction	type	Code décimal	Code Hexadecimal	Instruction	type	Code décimal	Code Hexadecimal
MOVE	move	00	00	BLE	1 opérande bis	16	10
PUSH	1 opérande	01	01	BGE	1 opérande bis	17	11
POP	1 opérande	02	02	BRA	1 opérande bis	18	12
ADD	2 opérandes	03	03	BSR	1 opérande bis	19	13
CMP	2 opérandes	04	04	JCC/JGT	1 opérande bis	20	14
SUB	2 opérandes	05	05	JCS/JLT	1 opérande bis	21	15
LSL	2 opérandes	06	06	JEQ	1 opérande bis	22	16
LSR	2 opérandes	07	07	JNE	1 opérande bis	23	17
AND	2 opérandes	08	08	JLE	1 opérande bis	24	18
0R	2 opérandes	09	09	JGE	1 opérande bis	25	19
XOR	2 opérandes	10	0A	JMP	1 opérande bis	26	1A
NOT	1 opérande	11	0B	JSR	1 opérande bis	27	1B
BCC/BGT	1 opérande bis	12	0C	RTS	0 opérande	28	1C
BCS/BLT	1 opérande bis	13	0D	TRAP	1 opérande bis	29	1D
BEQ	1 opérande bis	14	0E	RTE	0 opérande	30	1E
BNE	1 opérande bis	15	0F				

D Grammaire des opérandes

Mode d'adressage	Syntaxe	code	Commentaire
Registre	Rn	0	n∈[0,7]
Valeur immédiate	#v	1	v : valeur entière sur 16 bits
Adressage direct	@x	2	x : adresse mémoire sur 16 bits
Adressage indirect[18]	(Rn)	3	n∈[0,7]
Adressage indirect post incrémenté	(Rn)+	4	n∈[0,7] la valeur du registre est incrémentée après l'adressage
Adressage indirect pré décrémenté	-(Rn)	5	n∈[0,7] la valeur du registre est décrémentée avant l'adressage

E Automate



F Champs de bits et unions

Dans le langage C, le grain le plus fin pour représenter la mémoire à l'aide des types de base est l'octet, qui est donné par le type char. De manière à pouvoir descendre en dessous de cette limite, il

est possible d'utiliser une structure particulière, qui permet de diviser une variable en plusieurs "sous-variables" qui mesureront la taille en bit voulue par le programmeur.

Le programmeur a alors une contrainte, il devra allouer au minimum un espace mémoire d'une taille de 32 bits, qui sont dans le cas des machines du département, la taille des registres du processeur.

Nous pourrions alors imaginer la structure suivante :

```
typedef struct{
   int champ1 : 16;            /*Le champ 1 a une taille de 16 bits*/
   int champ2 : 13;            /*Le champ 2 a une taille de 13 bits*/
   int champ3 : 3;             /*Le champ 3 a une taille de 3 bits */
}couleur_processeur;
```

G Graphisme

G.1 Le Builder

Algorithme 7 Creation et utilisation du GtkBuilder

/*on crée le builder*/
p_builder = gtk_builder_new () ;
Si builder n'a pas pour valeur NULL **Alors**
/*Chargement du XML dans p_builder*/
gtk_builder_add_from_file (p_builder, <chemin_du_fichier_XML>, &p_err) ;
Fin Si

G.2 Récuperer les éléments graphiques

Algorithme 8 Récupération des GtkWidget

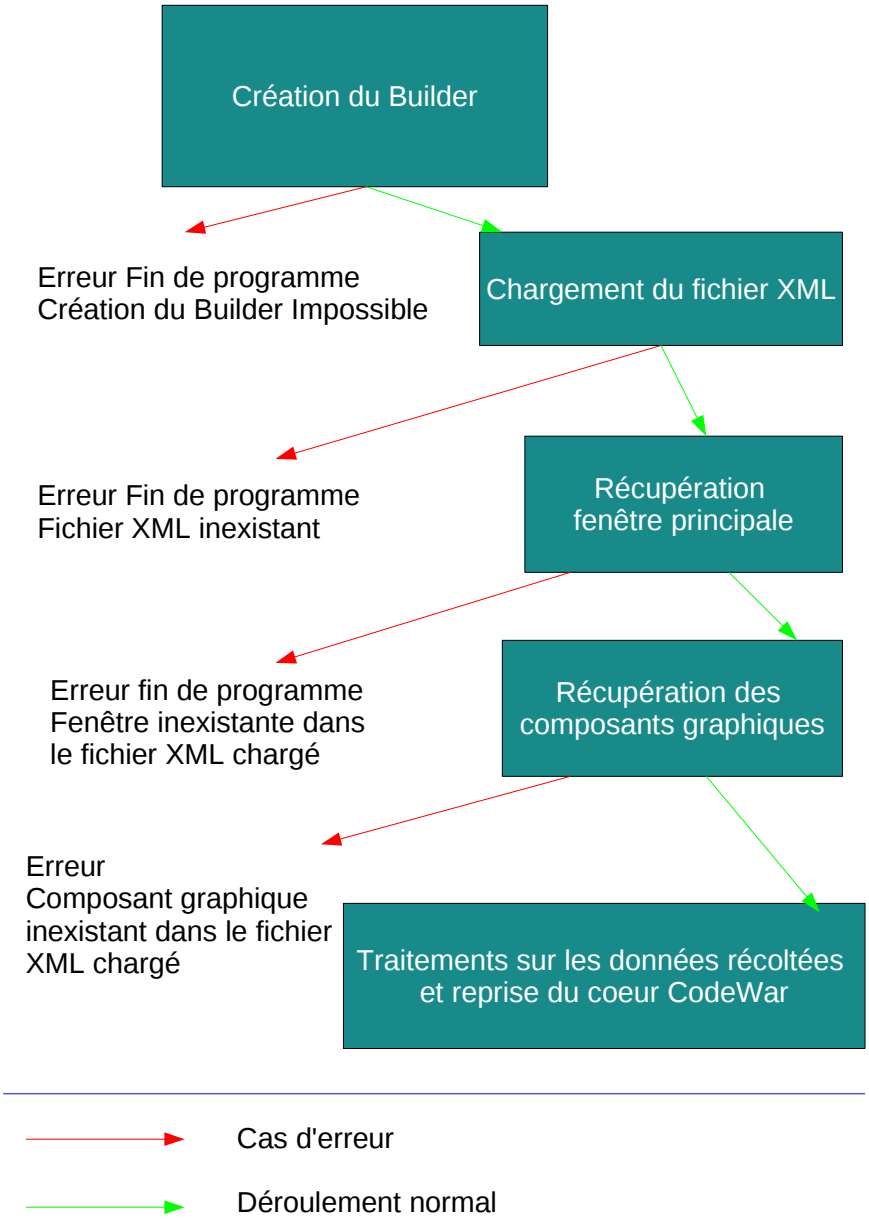
/*on peu récupérer un élément graphique via son nom et son Builder de fichier XML*/
<éléments_créés> = (<type_de_l_élément *) gtk_builder_get_object ((GtkBuilder*)unBuilder, <nom_de_l'élément_dans_le_XML>) ;

G.3 Association d'un signal à un slot

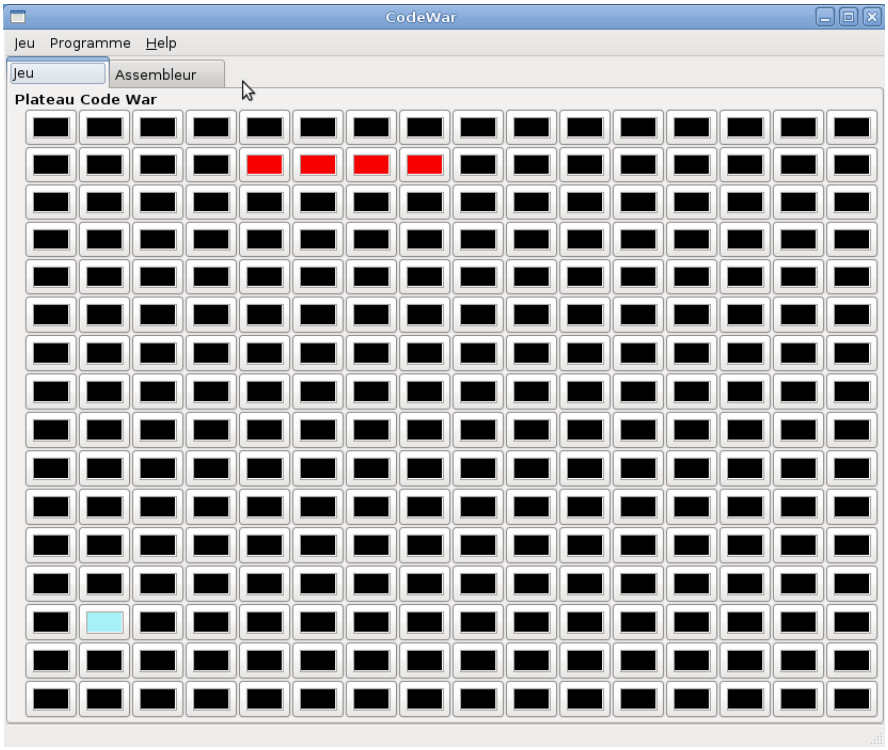
```
/*on peu récupérer un élément graphique via son nom et son Builder de fichier XML*/
        g_signal_connect (
   gtk_builder_get_object (<Le_Builder>, <"Titre_du_Composant">),
   <"type d'evenement de l'objet">, G_CALLBACK (<fonction>), pWindow
   );}
```

Où cliqued indique le type d'événement de l'objet :
Les évenements principaux :
cliqued : Pour les cliques sur les boutons
selected : Pour les selections sur les items de menus
destroy : pour le signal de fermeture de la fenêtre où de destruction du composant
color-set : Pour les changements d'état sur les couleurs des GTKColorButton
value_changed : Pour les changement de valeurs sur les GtkSpinButton
selection_changed : Pour les changement de Fichiers sur les bouton de selections de fichiers

G.4 Dynamique de l'interface graphique



G.5 Aperçu du résultat



Résumé

Notre projet tuteuré réalisé lors du semestre 3 de notre formation, traite de CodeWar, un jeu mettant aux prises 256 processeurs virtuels. Ces 256 processeurs fonctionnent à partir de programmes assembleurs, soit codés par le joueur et traduis en binaire par un compilateur fourni par notre tuteur, soit codés par défaut. Pour programmer ce jeu, nous avons dû tout d'abord concevoir la dynamique de celui-ci, mais aussi le fonctionnement d'un processeur, pour terminer par l'interprétation du binaire assembleur en une série d'instructions exécutable.

Pour réaliser ce travail, nous avons dû nous organiser en équipe, mener des recherches sur le fonctionnement d'un processeur, ou encore sur l'utilisation du langage C. Cette expérience est avec le stage à venir, le point d'orgue de notre formation, et nous a posé des défis, tels que le simple fait d'apprendre à travailler ensemble, que nous nous sommes efforcés de relever.

Dans ce rapport, nous traiterons de cette expérience, en essayant de vous la faire partager, de la manière la plus claire possible.

Abstract

Our tuthored project realised during the third semester of our training is about CodeWar, a game opposing 256 virtual processors. Those 256 processors working from assembly programs, either coded by the player and translated to binary by a compilator provided by our tutor, or coded by default. To program this game, we had first to design it's dynamic, then the processor working process, to end by turning the assembly-bynary into an executable instructions serie.

To fulfill this work, we had to organize into a team, deal with some research about the processor working process, or about the C programming language. This work, is with the coming stage, the highlight of our training, and set us with problems, like the simple fact of learning how to work together, that we tried our best to face.

In this report, we'll deal with this experience, trying to share it with you, by the clearest way as it's possible to do.