Blask

Vahan Boghossian Kimbembe Malanda Benjamin Peter
Ilan Schemoul

23 janvier 2023

Table des matières

Chapitre 1

Introduction

Blask est un projet de spécification de **console portable 16-bit** et de son **jeu d'instructions** associé. L'objectif du projet est de permettre la création de jeux 16-bits facilement pour tous. Dans ce dessein, nous avons décidé de concevoir une architecture simple, ouverte et bien spécifiée.

Le projet comprend la conception de l'**architecture du processeur** de notre console ainsi que de divers **outils de développement** tels qu'un **assembleur**, un **désassembleur**, un **débogueur** et un **émulateur**.

La **simplicité** d'utilisation est le maître mot pour ce projet. Un programmeur qui prend en main notre architecture, doit être capable de **facilement et rapidement** pouvoir commencer à **prototyper** des jeux rétro. Afin de pouvoir exécuter ses fichiers, le programmeur doit écrire son programme en **langage Blasm**, **assembler** le **programme** en **binaire** grâce à l'**assembleur**, puis, passer son **binaire** au **débogueur** ou à l'**émulateur** afin d'exécuter son code.

Chapitre 2

État de l'Art

Il existe plusieurs projets similaires au nôtre, certains modernes, d'autres plus anciens mais aucun qui cible vraiment la niche que nous avons souhaité cibler lors de la conception de Blask pour des raisons que nous allons vous présenter.

2.1 Game Boy

La **Game Boy** est l'un des projets qui s'approche le plus des objectifs du projet **Blask**. La **Game Boy** est une console de jeu développée dans les années 80 par Nintendo. Son processeur, le GBZ80 est inspiré des architectures de deux autres processeurs très célèbres, le 8080 d'Intel ainsi que le Z80 de Zilog. Cela aurait pu faciliter le développement de jeux pour la plateforme. Malheureusement, l'architecture de la **Game Boy** était assez fermée et il n'existe pas ou très peu de documentation officielle afin de développer pour la **Game Boy** et aussi très peu d'outils. C'est un problème qui encore à ce jour rend difficile la tâche de développer un émulateur ou des jeux pour la plateforme.

Un autre reproche que nous faisons à la **Game Boy** est en rapport avec les délais. La **Game Boy** n'ayant pas d'instruction pour arrêter le processeur pendant un certain temps, la plupart des jeux attendent un certain nombre de cycles processeur à la place. Cela empêche donc de faire fonctionner la plupart des jeux **Game Boy** sur une machine plus rapide sans limiter la vitesse de la machine artificiellement. Il n'est donc pas possible de profiter de l'augmentation de la vitesse des processeurs en utilisant l'architecture de la **Game Boy**.

2.2 CHIP-8

CHIP-8 est l'autre projet qui s'approche le plus des objectifs du projet **Blask**. **CHIP-8** est une plateforme virtuelle apparue dans les années 70 sur l'ordinateur **COSMAC VIP** pour laquelle il est facile de développer un émulateur (interpréteur en réalité). C'est une plateforme assez limitée avec seulement 4096 octets de RAM disponible et un écran monochrome 64x32. Son jeu d'instructions n'est pas non plus le très optimal compte tenu des ressources limitées de la plateforme et ressemble très peu à une vraie architecture processeur.

De nombreuses variantes et extensions de CHIP-8 furent développées au cours des années pour ajouter des fonctionnalités à la plateforme. Cela entraîna l'utilisation de jeux d'instructions différents selon les ROMs (comprenez jeu ici). Malheureusement, à cause de cette diversité, il arrive souvent que des jeux fonctionnent sur certains émulateurs et pas sur d'autres à cause de petites différences au niveau des jeux d'instructions utilisés.

2.3 Autres Architectures

Nous aurions pu utiliser une architecture existante mais aucune ne nous convenait réellement dans le cadre de notre projet de console. Notre objectif principal depuis le début est de concevoir une plateforme la plus simple possible mais suffisamment puissante pour exprimer la créativité des utilisateurs.

Nous avons donc voulu créer une architecture 16-bit. Une architecture 8-bit aurait été trop limitée tandis qu'une architecture 32-bit aurait été trop difficile à concevoir et rendrait le développement d'outils trop complexe. Une architecture 16-bit facilite énormément le développement d'un émulateur.

Nous avons décidé d'avoir une taille de registres unique pour éviter les problématiques liées au boutisme. Il existe très peu d'architectures ayant cette caractéristique et aucune parmi les architectures les plus populaires.

Nous nous sommes tout de même inspiré d'architectures existantes. Par exemple **RISC-V** nous a enseigné que nous n'avions pas forcément besoin de registre d'indicateurs (flags) ou encore d'interruptions.

Nous nous sommes aussi inspiré de **MIPS** et d'**ARM** afin de réduire le nombre d'instructions tout en maximisant les pseudo-instructions possibles.

Tout cela nous a permis d'atteindre un niveau de simplicité très satisfaisant.

Chapitre 3

Avancement

Les objectifs de notre projet étaient multiples.

Dans un premier temps, nous avons défini une architecture processeur pour notre console. Ensuite, nous avons défini les outils et les fonctionnalités de chacun de ces outils.

3.1 Architecture Processeur

Beaucoup de travail a été effectué sur la spécification de l'architecture. Nous avons décidé d'un ensemble minimal d'instructions qui permet d'exécuter un grand nombre d'algorithmes.

Nous avons choisi d'avoir à notre disposition 16 registres d'une taille de 16 bits.

Nous avons également défini plusieurs types d'instructions.

1. **R-Type Instruction** Ces instructions permettent d'effectuer des opérations entre les registres.
2. **I-Type Instruction** Ces instructions permettent d'effectuer des opérations entre un registre et un immédiat.
3. **B-Type Instruction** Ces instructions permettent de nous déplacer dans le code en changeant de ligne en fonction d'une condition.
4. **Load/Store Instruction** Ces instructions permettent d'échanger des données entre les registres et la mémoire.

3.2 Assembleur

L'**assembleur** est l'outil qui transforme du code en langage assembleur en langage machine binaire. Notre assembleur est constitué d'un **Lexer**, d'un **Parser**, d'un **LST** et enfin d'un **AST**. Notre assembleur va donc transformer notre programme en binaire ligne par ligne.

Celui-ci prend en entrée un fichier écrit dans notre langage assembleur, et en sortie va créer un fichier binaire que l'on pourra donner à l'émulateur.

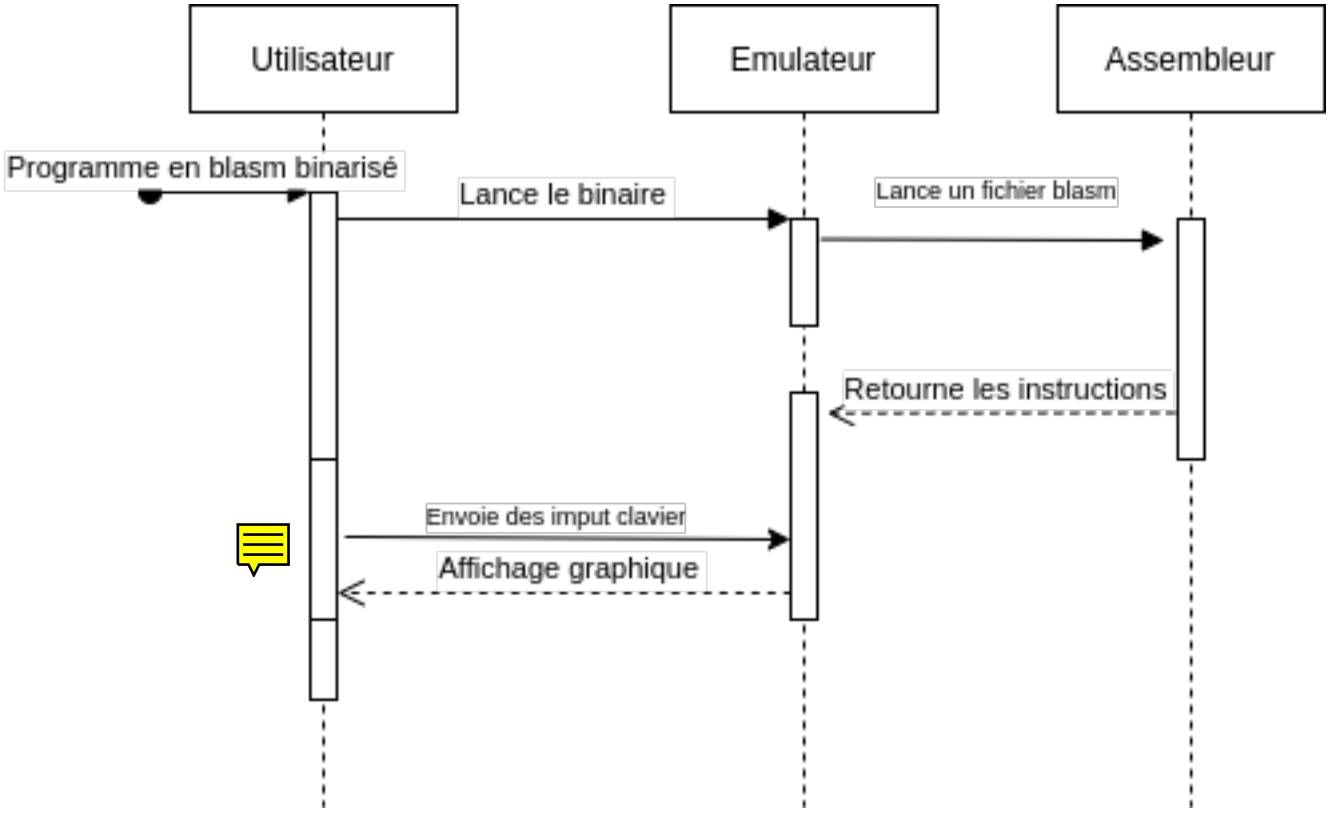

FIGURE 3.1 – Diagramme de séquence pour la création d'un binaire

3.2.1 Langage Assembleur

Notre langage assembleur comporte plusieurs éléments :

- **Instruction** Permet d'indiquer les actions que va réaliser notre cpu.
- **Label** Les labels permettent d'indiquer des adresses qui peuvent être la cible d'instructions de saut.
- **Commentaire** Permet aux développeurs de laisser du texte qui sera ignoré par l'assembleur.

Voici la grammaire de notre langage assembleur sous la forme de McKeeman :

```
File
  Element LineFeed File

Element
  ""
  Label
  Comment
  Instruction

Label
  '@' LabelID

LabelID
  LabelChar
  LabelChar LabelID

LabelChar
  '0' . '9'
  'A' . 'Z'
  'a' . 'z'

Comment
  '#' CommentContent

CommentContent
  ""
  ' ' . '10FFFF' - '000A' CommentContent

Instruction
  Mnemonic Operands
  Mnemonic Operands Comment

Mnemonic
  'a' . 'z'
  'a' . 'z' Mnemonic

Operands
  Operand
  Operand ',' Operands

Operand
  Label
  Number

Number
  '0' . '9'
  '0' . '9' Number

LineFeed
  '000A'
```

3.2.2 Développement

Il existe plusieurs étapes dans le développement d'un assembleur. La première étape est l'analyse lexicale qui consiste a transformer le code sous forme textuelle en liste de symboles atomiques. Cette étape est accomplie et nous a permis de commencer l'étape

suivante qui est l'analyse syntaxique.

Elle consiste à transformer notre liste de symboles obtenus au préalable, en une structure adaptée à notre langage. Dans notre cas puisque notre langage assembleur est assez simple notre structure est un ensemble de lignes qui sont soit un commentaire, soit un label, soit une instruction.

Il s'agit d'analyser la structure du code pour vérifier que toutes les règles du langage sont respectées puis de générer une liste d'instructions machine.

Et enfin la dernière étape est de transformer cette liste d'instructions en code binaire. Cette étape est accomplie dans un module séparé car elle est utilisée dans différents outils de notre projet.

3.3 Émulateur

L'**émulateur** est l'outil qui va nous permettre de lancer nos programmes Blask sur une architecture autre que celle de la console.

L'émulateur a pour rôle d'émuler le CPU de notre machine. Il va donc créer **virtuellement** les **registres** ainsi que la **mémoire** de la console. Il va ensuite prendre en entrée un fichier binaire préalablement compilé, et l'exécuter.

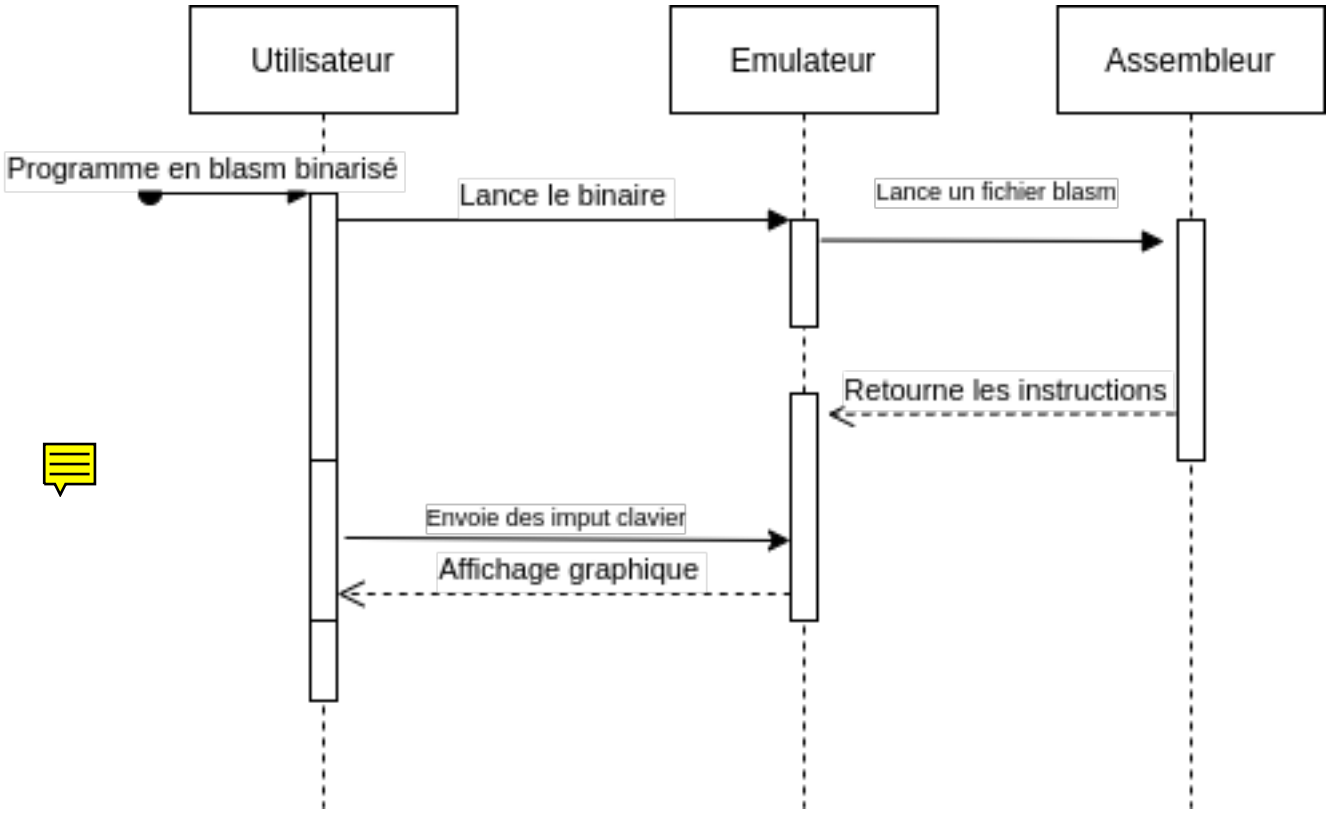

FIGURE 3.2 – Diagramme de séquence pour le lancement d'un binaire

3.4 Testsuite

Pour ce qui est des tests, la plupart sont intégrés dans le code à côté de leur fonctionnalité.

Le Lexer et l'Assembleur sont rigoureusement testés, cependant le reste manque encore de bons tests pour couvrir la plupart des cas d'utilisation.

Il faudrait également rajouter des tests d'intégrations afin de s'assurer de la non-régression de nos outils de développements.

3.5 Débogueur

Nous avons implementé un débogueur qui permet d'éxecuter ligne par ligne les instructions et d'imprimer les registres si requis par l'utilisateur.

```
Type p to print registers n to go to next instruction
n
59
Executing opcode : ADDI with rd 1 rs1 0 immediate 10
n
77
Executing opcode : ADDI with rd 2 rs1 0 immediate 0
n
112
Executing opcode : ADDI with rd 3 rs1 0 immediate 1
p
127
Registers :
Register 0: 0
Register 1: 10
Register 2: 0
Register 3: 1
Register 4: 0
Register 5: 0
Register 6: 0
Register 7: 0
Register 8: 0
Register 9: 0
Register 10: 0
Register 11: 0
Register 12: 0
Register 13: 0
Timer: 127
```

FIGURE 3.3 – Carré rouge

3.6 Fibonacci

Nous avons séparé le projet en trois grands jalons. Le premier jalon était d'avoir un algorithme qui calcule la suite de Fibonacci qui fonctionne sur notre émulateur.

Voici le programme en question :

```
# Input: 1
# Output: 2
@Fibo
  addi 2, 0, 0 # Move 0 into register 2
  addi 3, 0, 1 # Move 1 into register 3

  beq 1, 0, 8 # End function if register 1 is equal to register 0 (
   always 0)

  @Loop
  subi 1, 1, 1 # Decrement counter (register 1)
  addr 4, 0, 3 # Use register 4 as temporary to store register 3
  addr 3, 3, 2 # Add register 2 to register 3
  addr 2, 0, 4 # Store old value of register 3 into register 2

  bne 1, 0, @Loop # Restart loop if register 1 is not 0
```

Nous avons réussi à définir l'architecture de la console, ainsi que les instructions nécessaires pour faire fonctionner l'algorithme pour calculer la suite de Fibonacci. De plus, nous avons aussi complété l'émulateur qui exécute le binaire produit par l'assembleur pour exécuter toutes les instructions nécessaires pour Fibonacci. Nous avons donc complété ce premier jalon.

Nous avons tenté de développer de *strlen* et *memcpy*. Des programmes pour compter la longueur d'une chaîne de caractère et copier des blocs de mémoire spécifique en plus de Fibonacci mais ces programmes ne sont pas entièrement fonctionnel.

3.7 Carré rouge qui se déplace

Pour le deuxième jalon, il nous fallait implémenter la partie graphique ainsi que la mémoire pour pouvoir afficher des pixels à l'écran. Nous avons donc fait un petit programme qui déplace un carré rouge sur tous les pixels de l'écran.

Nous avons donc utilisé la librairie **Raylib** pour gérer les fenêtres via l'émulateur. L'émulateur exécute alors notre script et effectue également le rafraîchissement de l'écran qui affiche les pixels de notre console. Nous avons fixé notre console à 30 images par seconde, ce qui implique l'exécution de notre script ainsi que le rafraîchissement de l'écran 30 fois par seconde.

Afin de pouvoir représenter les pixels de l'écran sur la console, nous avons dédié une partie de la mémoire de la console aux pixels qui seront présents sur l'écran de celle-ci. Afin de ne pas prendre trop de mémoire, nous avons décidé de faire un écran de 32x32 pixels.

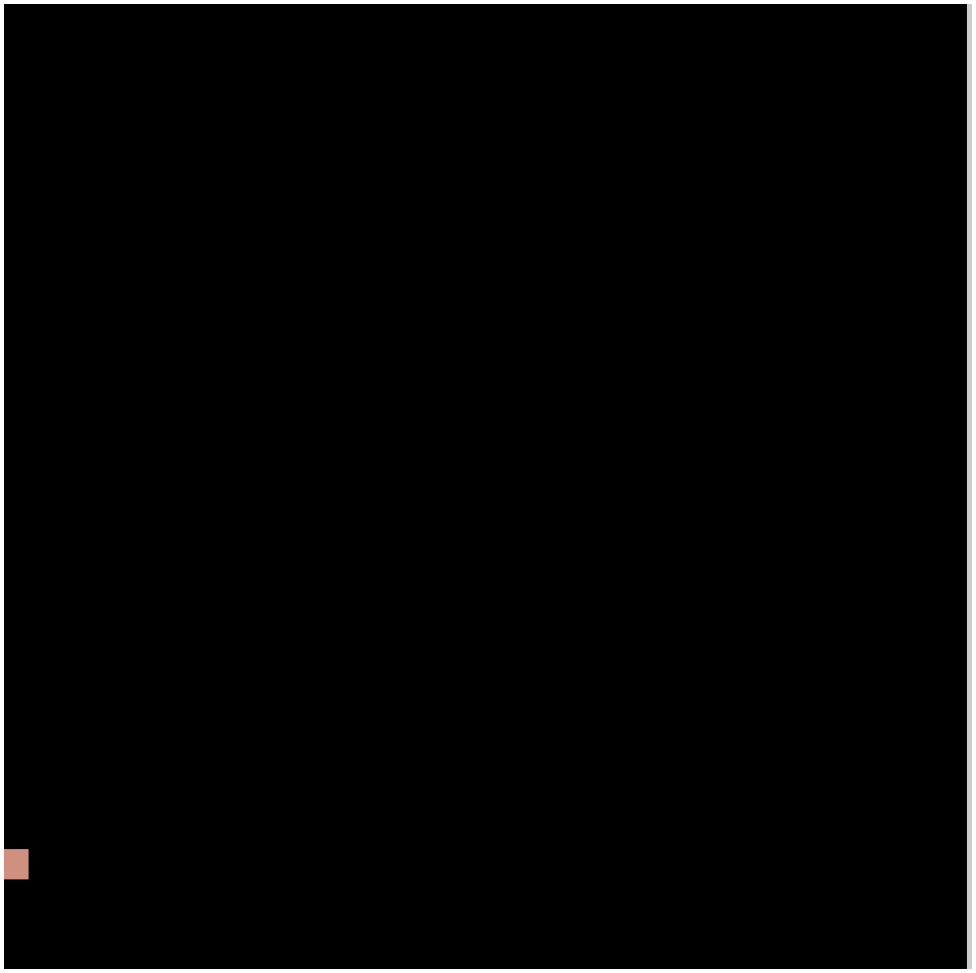

FIGURE 3.4 – Carré rouge

Chapitre 4

Étude de faisabilité

La faisabilité était au centre de nos préoccupations au commencement du projet. Nous nous sommes donc penché sur chaque point qui aurait pu nous empêcher de réaliser notre projet.

4.1 Architecture Processeur

Concevoir une architecture processeur ex nihilo semble être une tâche difficile voire impossible à accomplir dans un délai aussi court et avec aussi peu d'expertise dans le domaine.

4.1.1 Une Architecture Simple

Heureusement, notre objectif était de concevoir l'architecture la plus simple possible. Il y avait donc de nombreuses fonctionnalités qui sont habituellement disponibles sur les processeurs dont nous n'avions pas à nous soucier. En effet, notre architecture n'avait pas besoin de répondre à des problématiques d'interruptions, de registre d'état ou encore de parallélisation. Cela nous a permis de nous concentrer uniquement sur les opérations arithmétiques et logiques.

4.1.2 L'état de l'art

Il existe déjà une multitude d'architectures processeurs dont nous avons pu nous inspirer. Celle qui nous a le plus inspiré est **RISC-V**. Cette architecture était particulièrement intéressante pour nous car elle a très peu de fonctionnalités par défaut.

4.2 Emulateur

Une autre difficulté du projet est l'implémentation de l'émulateur. Celui-ci doit pouvoir émuler correctement l'état du CPU, de la mémoire et des périphériques (écran, entrées/-sorties) liés à notre console.

4.2.1 Une Spécification

Les projets d'émulation prennent souvent longtemps car beaucoup de tests sont nécessaires pour reproduire le comportement de la console initiale. Dans notre cas, nous sommes ceux qui spécifient le fonctionnement exact de notre console. Cela rend l'implémentation plus simple et nous fait gagner beaucoup de temps.

4.2.2 Interprétation

Il existe plusieurs types d'émulateurs. Il y a les émulateurs qui transforme le langage machine de la console en langage machine de la machine hôte. Puis il y a les émulateurs qui simulent l'état de la console et qui interprètent chaque instruction du code machine pour modifier cet état. Afin d'obtenir un résultat au plus vite nous avons décidé de pencher vers la solution d'interprétation dans un premier temps. Même si cette solution est plus lente, cela ne devrait pas se faire ressentir sur les jeux rétro pour lesquels notre console est conçue.

Chapitre 5

Problèmes rencontrés

Nous avons durant notre projet subis de nombreux problèmes. En effet, bien que les objectifs étaient bien défini, nous n'avons pas pu bien communiquer pour s'organiser et avancer efficacement.

La documentation du projet a été écrite très tard ce qui a été compliqué pour les membres du groupe moins impliqué de bien comprendre en profondeur la totalité du projet.

Le manque de communication entre les différent membres a été un des problèmes principal rencontré lors de la réalisation de ce projet. La plupart des membres ne travaillaient pas en groupe et vu que le travail n'avait pas été séparé équitablement, une seule personne s'occupait d'une partie importante du projet dont d'autres personne avait impérativement besoin pour pouvoir tester et donc avancer sur leur parties.

Cette partie en question a de plus pris plusieurs mois a être terminé et la plupart des membres n'était pas au courant de l'avancement de cette dernière. Il a de plus été révélé que la décision de fork la forge git et réécrire complètement cette partie avait été prise sans consulter ou mettre au courant les autres membres du groupe. Cela nous a fait perdre encore plus de temps et a entraîné de nombreuses altercations entre les différents membres.

De plus, le groupe s'est très rarement retrouvé au complet au laboratoire GISTRE durant les heures prévues pour travailler ensemble sur le projet durant l'année. Cela a également mis à mal la motivation de membres vis à vis du projet du à l'absence répété de membres du groupe lors de ces séances.

Chapitre 6

Conclusion

Blask était un projet très ambitieux. Le fait de créer une architecture spécialement pour ce projet est extrêmement formateur et nous a permis de comprendre encore plus le fonctionnement bas niveau.

Nous avons donc après de nombreuses recherches sur les anciennes consoles, fait le choix de partir sur une console que l'on pourrait créer de zéro par nous même. En nous fixant nos propres contraintes et en étant plus libres sur les choix importants pour définir les capacités de notre console.

Nous avons alors commencé par bien définir les composants de la console, comment et pourquoi ceux-ci seraient utiles pour notre architecture. Cette phase a été cruciale pour la conception d'une console à notre goût.

Nous avons ensuite pu implémenter un assembleur ainsi qu'un émulateur avec une grosse partie des fonctionnalités que nous avions prévu.

Nous avons pu valider nos deux premiers jalons. Dans un premier temps la validation de **Fibonacci** nous a permis de confirmer que notre architecture était capable de faire fonctionner des algorithmes de calcul. Puis la validation du carré rouge nous a permis de voir le fruit de notre travail. Il est maintenant possible d'implémenter des programmes graphiques sur **Blask**.

Le projet touche à sa fin et nous n'avons malheureusement pas eu le temps d'explorer tous les sujets que nous voulions (désassembleur, jeux).

Nous restons cependant très satisfaits de cette expérience qui fut très formatrice.