Documentation de l’extension

Equipe GL27

→ *Quentin Cousin*
→ *Christophe Ehret*
→ *Adrien Kaufman*
→ *Berkan Kesman*
→ *Mathis Lavigne*

Sommaire

Introduction

Nous avons choisi pour notre extension de nous intéresser aux machines Game Boy Advance (GBA) qui sont des consoles de jeux vidéo portables sorties en 2001. Ces consoles utilisent un processeur ARM7TDMI et Zilog Z80. Nous ne nous intéressons ici qu'au processeur ARM7TDMI sorti en 1994, le second, plus vieux, étant utilisé à des fins de rétro-compatibilité avec d'anciennes consoles (Game Boy).

Le processeur ARM7TDMI est un processeur de la famille du processeur ARM7 sorti en 1993. Il a la particularité de proposer deux jeux d'instructions différents : ARM et THUMB. Le premier est un jeu d'instruction plus étendu mais moins rapide que le second.

Nous avons choisi cette machine notamment car nous avons la possibilité de tester sur machine réelle. En plus des tests sur émulateurs que nous avons fait, cela nous a permis de vérifier le bon fonctionnement de nos programmes sur une vraie Game Boy Advance.

Ce document a pour objectif de détailler notre démarche dans l'implémentation de l'extension. Dans une première partie, nous explicitons les spécifications de notre extension, notamment en comparaison à ce qui doit être fait pour la machine IMA. Dans une deuxième partie nous synthétisons les informations importantes pour le développement de l'extension que nous avons apprises pendant nos recherches. Dans une troisième partie, nous décrivons nos choix de conception, et comment nous avons décidé d'adapter le projet pour répondre aux problèmes posés par l'ajout d'un langage assembleur dans le compilateur. Dans une quatrième partie, nous expliquant ce qu'a été notre méthode de validation et les résultats de cette validation. Enfin, une dernière partie est consacrée aux liens qui nous ont été utiles pour développer l'extension.

Spécifications

Caractéristiques principales de l'implémentation

La complexité de l'implémentation ne nous permettant pas de fournir un compilateur Decac complet avant la fin du projet, nous avons décidé de nous limiter au "sans objet". De plus, nous avons choisi de supprimer la gestion des flottants. En effet, les registres n'étant que du type entier, et la gestion des flottants étant complexe (notamment le stockage des flottants et l'implémentation des opérations + - * /), nous avons décidé de laisser ce point de côté. Nous n'excluons cependant pas la possibilité d'implémenter ces parties après la soutenance afin d'avoir un compilateur complet pour GBA.

Toute la partie Sans Objet (hormis les flottants) a donc été implémentée. Notamment :

- `print/println` : Affiche des entiers et des chaînes de caractères conformément à la spécification Decac.
- Déclaration de variables entières et booléennes - Evaluation d'expressions entières et booléennes.
- `If Then Else` : Exécute l'une ou l'autre des branches conformément à la spécification Decac.
- `While` : Exécute un code tant qu'une condition est vérifiée conformément à la spécification Decac.
- Vérification des débordements de la pile (stack overflow), erreurs lors de divisions par 0.

Nous avons par ailleurs décidé de faire des fonctions additionnelles propres à la GBA :

- `getKey()` : fonction renvoyant un entier décrivant l'état des boutons de la machine (pressés ou non). C'est en quelque sorte un substitut aux fonctions `readInt()` et `readFloat()` qui, par nature, ne peuvent pas être implémentées telles quelles sur GBA.

- `display(String chaine, int Y, int X)` : instruction affichant une chaîne de caractères à un emplacement donné.

Utiliser ces dernières pour des programmes IMA cause bien entendu une erreur. Une spécification plus précise des ces fonctions est proposée dans la partie suivante.

Fonctions propres à l'implémentation GBA

La fonction `getKey()`

La fonction `getKey()` est une fonction propre à l'implémentation pour Game Boy Advance. Elle retourne un entier entre 0 et 1023 décrivant l'état des boutons de la GBA (pressés ou relâchés). Chaque bit de ce nombre est associé à un bouton : si le bit est à 0, le bouton est relâché, si le bit est à 1, le bouton est pressé. Il y a 10 boutons sur la console,

d'où le fait que ce nombre soit compris entre 0 et 1023. Voici un tableau récapitulant la correspondance Bit/Bouton :

Bits	Bouton
0	Bouton A
1	Bouton B
2	Select
3	Start
4	Droite
5	Gauche
6	Haut
7	Bas
8	Bouton R
9	Bouton L
10-31	Inutilisés (à 0)

Cette fonction possède toutefois ses limites. En effet, il n'est pas possible en Decac de faire des opérations bit à bit (opérateurs & | ~ en C par exemple). C'est pourquoi si l'on presse plusieurs touches différentes simultanément, on ne pourra pas facilement récupérer l'état d'un seul bouton. On peut imaginer que dans une future version nous ajoutions les opérateurs "bitwise".

La fonction `display(String chaine, int Y, int X)`

Cette fonction permet d'afficher une chaîne de caractères aux coordonnées (X, Y). Elle est utile pour faire des jeux ASCII et nous a notamment permis de faire fonctionner le jeu du PONG sur Game Boy Advance. Il est important de noter que la coordonnée Y est devant la coordonnée X dans les arguments de la fonction. C'est une façon de faire commune lorsqu'il s'agit d'afficher des caractères et non des pixels.

Cette fonction n'est autre qu'un changement des coordonnées du curseur puis d'un `print` de la chaîne. `display` change donc aussi les coordonnées du prochain `print`.

Analyse bibliographique

Introduction

Cette partie a pour objectif de faire la synthèse de ce que nous avons appris de la console utilisée lors de ce projet. Les sources sont disponibles dans la partie “Liens” à la toute fin de ce document.

Processeur : l’ARM7TDMI

Général

Le processeur de la Game Boy Advance est un ARM7TDMI cadencé à 16.78 MHz. Il est de la famille des processeurs ARM7 et est sorti en 1994. Celui-ci est de type RISC, c’est-à-dire que son jeu d’instruction est réduit mais qu’il possède un nombre plus élevé de registres (voir la partie sur les registres).

Ce qui différencie l’ARM7TDMI du simple ARM7 est le fait qu’il possède deux jeux d’instruction :

- ARM : ce jeu d’instruction est celui qui possède le plus de similitude avec celui de l’ARM7. Les instructions sont codées sur 32 bits ce qui lui permet d’être très polyvalent.
- THUMB : celui-ci est unique à l’ARM7TDMI. C’est un jeu d’instruction réduit codé sur 16 bits. Certains registres sont moins facilement accessibles dans ce mode.

Le bus de lecture de la ROM de la cartouche (là où est le code du programme) faisait 16 bits, le jeu THUMB permet de multiplier la vitesse d’exécution des programmes en moyenne de 1.6. En contrepartie, les programmes sont plus complexes.

La GBA possède un autre processeur qui est fortement inspiré du Zilog Z80. Ce dernier est exclusivement utilisé à des fins de rétro-compatibilité avec d’anciennes consoles de la même famille (en l’occurrence la Game Boy). Il ne sera pas exploité dans le cadre de ce projet.

Registres

L’ARM7TDMI possède 16 registres de 32 bits chacun simultanément accessibles nommés R0...R15. Il est à noter que le processeur possède entre 6 et 16 modes d'authentification (dépendant de la version du processeur). Pour faire simple, chaque mode possède son propre jeu de registre (même si certains registres peuvent être partagés par plusieurs modes). Ceci permet d’éviter un mauvais partage des registres au sein du système. À titre d’exemple, les interruptions ont leurs propres registres R13 et R14. Au total, le processeur possède 37 registres. Ici, nous n'utilisons qu’un seul mode, c’est pourquoi nous ne détaillons pas cet aspect davantage.

Un autre registre spécial vient s'ajouter aux 16 registres précédents : le registre CPSR (Current Program Status Register). Ce registre conserve l'état des flags du système. Par exemple, si une opération a dépassé la capacité d'un registre, le flag V (Overflow) sera mis à 1. Les flags mis à disposition du programmeur et leur signification sont les suivants :

BIT	FLAG	SIGNIFICATION
31	N - Sign Flag	0 : non signé 1 : signé
30	Z - Zero Flag	0 : différent de zéro 1 : zéro
29	C - Carry Flag	0 : pas de retenue 1 : retenue
28	V - Overflow Flag	0 : pas de dépassement de capacité 1 : dépassement de capacité
27	Q - Sticky Overflow	Non présent sur GBA
26 - 8	Réservé	-
7	I - IRQ disable	0 : interruptions inactives 1 : interruptions actives
6	F - FIQ disable	0 : "Fast-interrupts" inactifs 1 : "Fast-interrupts" actifs
5	T - State Bit	0 : ARM 1 : THUMB
4 - 0	M4-M0 - Mode Bits	Mode d'authentification

Ces flags sont par la suite accessibles dans le jeu d'instruction. Plus d'informations sur l'utilisation des flags est donnée dans la partie "Jeu d'instruction ARM".

Certains des 16 principaux registres ont des rôles spéciaux :

- R15 est le Program Counter (PC). Contrairement à certains processeurs, il est accessible (presque) comme n'importe quel autre registre dans le jeu d'instruction. Il est donc possible de faire `mov R15, #45` pour sauter à l'adresse 45.
- R14 est le Link Register (LR). Il est utilisé lors des branchements vers des sous-routines. Le processeur utilise ce registre pour stocker l'adresse de la prochaine instruction lors d'un appel de routine avec `BL` ou `BLX`. Il faut donc faire attention si cette routine appelle elle-même une sous-routine. Dans ce cas, il faudra au préalable sauvegarder la valeur de R14.

- R13 est le Stack Pointer (SP). Dans le jeu d'instruction ARM, il est possible d'utiliser un autre registre que R13 comme pointeur de la pile. Cependant, la norme est de laisser ce rôle à R13.

Les 13 autres registres (R0 à R12) sont des registres communs et peuvent être utilisés librement.

Jeu d'instruction ARM

Afin de simplifier la génération de code, nous ne nous intéresserons qu'au jeu d'instruction ARM.

Ce jeu d'instruction est assez proche de celui de la machine IMA à quelques différences près. Notamment, chaque instruction est conditionnelle. C'est-à-dire que chaque instruction peut dépendre du CPSR et donc s'exécuter conditionnellement. Par exemple, pour charger une valeur dans un registre seulement si la flag EQ est à 1, on utilise l'instruction étendue `MOVeq`. De plus, pour la plupart des instructions, il est possible de choisir si l'on veut qu'elle mette à jour le CPSR ou non en ajoutant le suffixe `s` à l'instruction. Par exemple, `MOV` ne modifie pas le CPSR mais `MOVs` si.

Les familles d'instructions ainsi que les instructions importantes sont énoncées ci-après. Des explications sont données quand nécessaire. Il ne faut pas oublier que toutes ces instructions peuvent être conditionnelles.

- Gestion de la mémoire :
  - MOV Rd, Op2 — Met une valeur Op2 dans Rd
  - LDR Rd, <Addr> — Charge la valeur présente à <Addr> dans Rd
  - STR Rd, <Addr> — Stocke la valeur de Rd à l'adresse <Addr>
- ALU
  - ADD Rd, Rn, Op2
  - SUB Rd, Rn, Op2
  - CMP Rd, Op2
  - MUL Rd, Rm, Rs
  - MLA Rd, Rm, Rs, Rn — Rd <- Rm*Rs+Rn
- Branchements
  - B Label — R15 <- =Label
  - BL Label — R14 <- R15+4 ; R15 <- =Label
  - SWI Imm24b — Appel une routine système
  - MOV PC, Rn

Op2 peut être :

➢ Un immédiat 8 bits "left-sifté" un nombre pair de fois. Exemple : 256 est un Op2. 257 n'en est pas un.

➢ Un registre shifté ou non, à droite ou à gauche, arithmétiquement ou logiquement (tous les cas sont possibles).

Mémoire

Organisation

La mémoire est partagée entre celle de la cartouche de jeu et celle de la mémoire de la console. Les principaux espaces mémoires (avec leurs principales caractéristiques) sont les suivants :

NOM	TAILLE	BUS WIDTH	WAIT STATES	DESCRIPTION
BIOS	16 Ko	32	1	Code de démarrage + routines système
ROM Cartouche	32 Mo Max	16	5	Code assembleur utilisateur à exécuter
Internal WRAM	16 Ko	32	1	Rapide mais petite. Contient la pile.
External WRAM	256 Ko	16	3	Grande mais plus lente.
VRAM	96 Ko	16	1	Vidéo RAM. Voir partie “Fonctionnement de l’écran”
I/O Port	1023 o	32	1	Ports mappés en mémoire (son, pad, …)

Routines système

Pour pallier la simplicité du jeu d’instruction, la GBA met à la disposition des développeurs un certain nombre de routines système accessibles grâce à l’instruction `SWI`. Une liste détaillée est accessible sur le site GBATEK (voir la partie “Liens”). Voici quelques exemples de routines :

- Une routine de division (code 06) : Permet d’effectuer une division signée. Renvoie le quotient, la partie absolue du quotient et le reste de la division entière.
- Des routines de compression/décompression LZ77 ou RLE.
- Une routine pour calculer l’arc tangente d’un nombre.
- Des routines pour approximer un cosinus ou un sinus (à l’aide de LUT).

La routine de division est la seule dont nous nous sommes servis pour ce projet. Elle a permis d'économiser le temps qu’il nous aurait fallu pour créer nos propres routines de division et de modulo.

Fonctionnement de l'écran

L'écran de la GBA fait 240 pixels de largeur sur 160 pixels de hauteur. C'est un écran 16 bits, pouvant donc afficher jusqu'à 65536 couleurs différentes.

Celui-ci est mappé en mémoire à partir de l'adresse `0x06000000`. Autrement dit, chaque pixel correspond à une adresse mémoire et écrire sur cette dernière provoque quasi instantanément un changement de la couleur de ce pixel. Il existe 6 modes de configuration de l'écran jouant sur le nombre de bits par pixel et sur le nombre d'arrière-plans utilisables. Une description plus précise de ces modes est disponible via les liens à la fin de ce document.

Pour ce projet, nous avons utilisé exclusivement le mode 3. Dans ce mode, chaque pixel est codé sur 16 bits : 5 bits de rouge, 5 bits de vert et 5 bits de bleu (+ un bit inutilisé).

Afin d'initialiser un mode, il faut mettre son numéro (ici 3) dans le port mappé en mémoire à l'adresse `0x04000000`. Adresse qui est, comme on peut le remarquer, le début de l'espace mémoire I/O.

L'utilisation et la configuration de l'écran peut-être assez complexe. Cependant, nous n'avons eu besoin que des informations précédentes pour la génération de code GBA. Nous pensons donc qu'il n'est pas nécessaire de rentrer davantage dans les détails.

Fonctionnement du Pad

Le Pad est le principal mécanisme d'interaction avec l'utilisateur. Il est composé de 10 boutons : A, B, Gauche, Droite, Haut, Bas, Select, Start, L et R.

L'utilisation du Pad est très simple : en effet, il est accessible via un port mappé en mémoire à l'adresse `0x4000130`. Il suffit donc de lire les 2 octets pour connaître l'état des boutons. Un tableau montrant la correspondance bouton/bit est disponible dans la partie `getkey` des spécifications de l'extension.

Ce port est ACTIVE-LOW, c'est-à-dire que le bit associé au bouton est à 0 quand ce dernier est pressé et à 1 quand celui-ci est relâché. Nous avons choisi avec la fonction getKey d'inverser ces bits afin de rendre l'interface utilisateur plus logique. Les bits les plus significatifs qui ne sont associés à aucun bouton sont, eux, à 0.

Comparaison de l'architecture de la GBA avec celle d'IMA

Cette partie synthétique a pour objectif de mettre en perspective les différences entre la machine IMA et la GBA qu'il sera nécessaire de prendre en compte pendant l'implémentation.

- Dans un premier temps, la GBA ne possède pas de registres LB et GB et les registres R13, R14, R15 sont déjà utilisés par la console.

- Le processeur de la GBA ne possède pas d'instruction pour diviser ou calculer le modulo. Heureusement, la console en elle-même propose une routine système qui fait ces deux opérations (accessible via l'instruction `SWI 0x060000`)
- La GBA n'a pas de gestion du tas (heap). Il est donc nécessaire d'ajouter à chaque programme des routines assembleur imitant le fonctionnement de l'instruction NEW.
- La GBA n'a pas non plus de gestion matérielle des flottants. Il aurait donc été nécessaire de reprogrammer cette gestion de zéro si nous avions décidé d'implémenter cette partie. A noter que les registres du processeur ne possèdent pas de type (donc pas de #null, ou de #0.0, seulement des entiers signés).
- La GBA n'a pas d'instructions HALT ou ERROR. Il faut donc trouver un équivalent de ces deux instructions (nous avons choisis de faire des boucles infinies).
- De plus, la Game Boy Advance ne possède aucune instruction pour afficher des nombres ou du texte à l'écran. Il faudra donc gérer l'affichage de caractères depuis zéro (voir la partie "Gestion de l'écran").
- Enfin, la GBA ne possède pas de clavier. Par conséquent, les entrées/sorties doivent être gérées différemment (ReadInt, ReadFloat).

Choix de conception

Sur le compilateur

Modification globale de la structure des packages IMA

Le premier grand choix de conception que nous avons dû faire concerne la modulabilité du projet : nous devions faire en sorte que l'ajout d'une machine et de son assembleur soit la plus logique possible pour le programmeur. Pour cela, nous ne pouvions plus mettre au centre de la partie codegen un objet de type `IMAProgram`. De même, nous devions concentrer la génération de code et ne pas l'éparpiller dans plein de packages différents. Nous avons alors décidé de regrouper la grande partie de la génération de code dans des méthodes du type `codeGenXXXYYY`, où `XXX` peut être par exemple `Bool`, `Expr` ou encore `Inst` et `YYY` est le nom de la machine (`IMA` ou `GBA`). Grâce à cela, le programmeur aurait juste à faire "`grep -r codeGen`" pour savoir où il devait ajouter ses méthodes de génération de code. N'ayant pas pu aller jusqu'au bout de cette idée, il est nécessaire, en plus de cela, d'ajouter dans quelques autres classes quelques instructions de génération de code (notamment dans le `RegisterHandler`).

Un autre problème était de rendre la création des packages propres au langage (`pseudocode` et `pseudocode.YYY.instructions`) la plus systématique possible. Pour cela, nous avons créé un package commun `fr.ensimag.pseudocode` qui regroupe toutes les classes pouvant servir à tous les langages assembleurs (par exemple `Operand`, `Line` ou encore `Label`). Il suffit alors de créer la structure du langage assembleur dans des packages, dans lesquels chaque classe représente une instruction. Pour que la partie gérant la génération de code puisse appeler les bonnes méthodes, il y a quelques contraintes sur ces packages :

- Il doit exister une classe héritant de la classe abstraite `AsmProgram` et fournissant les méthodes dont elle a besoin (notamment `addInstruction`).
- Il doit exister une classe héritant de AbstractRegisterSet et fournissant les méthodes dont elle a besoin (notamment la définition de LB et GB, et une fonction `getR`).
- Il doit exister une classe héritant de AsmProgram et fournissant les méthodes dont elle a besoin (notamment `addInstruction`).
- Il doit exister une classe héritant de `Line` et fournissant les méthodes dont elle a besoin (notamment `addLine`).
- Toute instruction doit hériter d'une manière ou d'une autre de `AbstractInstruction`.
- Toute opérande doit hériter d'une manière ou d'une autre de `Operand`.

Le reste est à la charge du développeur et il est libre d'implémenter son langage comme il le souhaite.. Il est cependant conseillé d'utiliser une structure de classe similaire à celle d'IMA (chaque instruction descend d'une classe d'instruction etc).

Enfin, il fallait rendre l'ajout de code le plus systématique possible. Nous avons notamment dû rendre la classe `DecacCompiler`, l'ajout d'instruction et les accès au registres les plus unifiés possibles. Nous avons pour cela conçu des méthodes abstraites comme `AsmProgram` (dont `IMAProgram` hérite) et `AbstractRegisterSet` qui est utilisé par le `RegisterManager` et qui décrit comment doivent être utilisés les registres de la machine (qui est LB/GB, qui est SP etc).

Gestion des routines externes

La GBA n'offrant pas autant d'abstraction que la machine IMA, il nous a fallu trouver un moyen pour lier les routines (fonctions) que nous avons créées aux programmes assembleur générés par notre compilateur. Pour cela, nous aurions pu recopier "à la main" (e.g à l'aide des méthodes `compiler.addInstruction`) mais pour des soucis de maintenance et de clarté, nous avons décidé de mettre ces routines dans des fichiers externes, inclus dans le fichier généré à l'aide de la directive `.include`. Cette méthode a l'avantage de permettre de modifier le code plus facilement et d'éviter la répétition de code (en opposition avec une autre méthode consistant à recopier le code à chaque fois que l'on en a besoin). Elle a cependant le défaut d'inclure le code même si celui-ci n'est pas utilisé.

Afin de gérer la correspondance "nom de routine/Label", nous avons ajouté au projet (dans le package `fr.ensimag.deca.codegen`) une classe `RoutineManager`. Cette classe est simplement une `HashMap<String, Label>` permettant dans un premier temps d'ajouter une correspondance, et dans une deuxième temps de récupérer un Label à partir du nom d'une routine. Cet objet

Adaptation générale du code

En plus des changements énoncés auparavant, il a fallu adapter le code aux contraintes de la GBA. Par exemple, les chaînes de caractères n'étant pas gérées par la console, il a fallu concevoir le stockage de ces chaînes à l'aide d'un `DataManager`, les `String` étant alors stockées à la fin du programme sous forme d'une suite d'octets.

De même, la gestion des erreurs est aussi légèrement différente, il a donc fallu modifier la façon dont les erreurs étaient affichées et comment on gérait la sortie de programme.

Sur la console

Gestion des registres

Comme expliqué dans la partie "Analyse bibliographique", l'ARM7TDMI possède 16 registres simultanément accessibles de R0 à R15. Certains registres ont déjà des rôles prédéfinis:

- R13 est l'équivalent de SP (mais il est quand même possible de choisir un autre registre dans le mode ARM).

- R14 est le Link Register, il est mis à PC+4 lors d'un appel de routine avec BL ou BLX.
- R15 est l'équivalent de PC.

En plus de ces trois registres de contrôle, il nous faut 2 registres supplémentaires pour GB et LB. Nous avons donc naturellement choisi R12 pour LB et R11 pour GB.

Par conséquent, les registres scratch étant R0 et R1, les registres R2 à R10 servent pour l'évaluation des expressions.

Gestion du tas (heap)

La partie objet n'ayant pas été implémentée, cette partie fait seulement part de nos réflexions par rapport à ce qu'il serait possible de faire pour implémenter le tas.

L'implémentation du tas peut être très sommaire car les spécifications du langage Decac n'exigent pas que l'on doive supprimer les objets du tas au fur et à mesure de l'exécution de programme. L'adresse du prochain élément à mettre sur le tas est donc calculée de manière systématique : l'adresse du précédent élément alloué + sa taille. C'est d'ailleurs exactement le fonctionnement de l'instruction `new` que nous avions commencé à implémenter.

Au niveau de la mémoire, il est possible de mettre le tas soit l'Internal WRAM, soit dans l'External WRAM. Le premier ayant l'avantage d'être plus rapide, et le second d'être plus gros. Une description plus détaillée des types de mémoire est disponible dans la partie "Analyse bibliographique".

Gestion de la pile (stack)

La pile est, d'une certaine manière, déjà implémentée par le processeur. La principale question que nous avons dû nous poser est "où mettre la pile ?". Par défaut la pile commence à l'adresse la plus haute de l'Internal WRAM et descend vers les adresses décroissantes. Cette place nous convient car elle maximise l'espace que nous pouvons utiliser avec la pile. Nous avons donc décidé de la laisser à cette adresse.

Gestion des erreurs à l'exécution

Une liste des erreurs que peut provoquer un programme GBA est disponible dans la partie "Spécification" de ce document.

La gestion des erreurs à l'exécution (Runtime Errors) ne peut pas se faire comme sur IMA. En effet, l'instruction `ERROR` n'existant pas et les chaînes de caractères n'étant pas supportées par défaut par la console, nous avons dû légèrement adapter la génération de code. Le principe reste tout de même similaire : le code des erreurs est inséré à la fin de tout programme GBA.

Comme il n'est pas possible de quitter un programme GBA lors d'une erreur, nous avons choisi de le bloquer, choix que nous avons aussi fait pour remplacer l'instruction `HALT`. La cause des erreurs est, par ailleurs, affichée grâce à la routine `printStr` que nous avons

créée pour pallier le manque d'une instruction `WSTR` (une description du fonctionnement de cette routine est proposée dans la partie "Gestion de l'écran").

Gestion de l'écran

Afficher du texte

Principe

Afin d'afficher du texte, et puisque la console ne gère d'aucune façon l'affichage de caractères, nous avons décidé d'utiliser des sprites. Les caractères supportés sont les suivants :

```
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
```

Chaque caractère disponible est donc représenté par un sprite, c'est-à-dire un tableau de "couleurs", dans lequel chaque case est associée à un pixel du sprite. Afin d'optimiser la place que prend les sprites dans la ROM `.gba`, nous avons aussi décidé de stocker les caractères en monochrome (e.g 1 bit/pixel : il est soit éteint, soit allumé). Pour des caractères de 8x8 pixels, nous divisons ainsi l'occupation dans la mémoire par 16, passant de 128o/pixel à 8o/pixels. Dans notre cas, nous avons 96 caractères : à raison de 16 bits par pixel (l'écran est en effet en 16 bits, voir l'analyse bibliographique) nous passons donc de 12288 octets à 768 octets, ce qui est un gain de place non négligeable pour une ROM de taille fixe qui est de l'ordre de quelques centaines de Kilo octets.

Curseur

Il est essentiel pour une fonction affichant un texte de retenir la position du curseur au fur et à mesure de l'exécution du programme. En effet, afficher le texte systématiquement en haut à gauche risquerait d'effacer un précédent affichage. Il nous a donc fallu implémenter un système de curseur.

Le curseur n'est ni plus ni moins que deux entiers stockés en RAM : l'un décrivant la position X du curseur, l'autre la position Y de celui-ci. Ces entiers vont respectivement de 0 à 29 et de 0 à 17. En effet, nous avons décidé de diviser l'écran en cases de 8x8 pixels. Cette abstraction permet de rendre plus facile la gestion du curseur. Elle enlève néanmoins la possibilité d'afficher un caractère entre deux cases.

Afin de gérer la position du curseur, le fichier stdio.asm propose 2 routines : une permettant d'incrémenter la position du curseur et de revenir à la ligne si jamais nous dépassons la limite en X, l'autre permettant de positionner le curseur à un endroit précis. La première est utilisée directement par la fonction decac `println`, la seconde sert pour la fonction decac `display`.

Algorithme

La GBA ne possédant pas de routine d'affichage de caractère, nous avons dû la programmer. Cette routine est présente dans le fichier stdio.asm, qui est la racine du projet. Celle-ci prend en paramètres l'adresse d'une chaîne de caractères terminant par le caractère 0x00 dans R1, et la position du curseur dans les espaces mémoires présentés ci-dessus. Elle boucle tant que le prochain caractère à afficher est différent de 0x00, tout en mettant à jour la position du curseur.

L'affichage du caractère en lui-même est fait par une autre routine `printChar`. Cette routine prend en paramètres le numéro d'un caractère dans R1 et la position du curseur. Afin d'afficher le caractère, elle calcule la position de son sprite dans la mémoire (debSprites+n*8 avec n, le numéro du sprite), puis elle itère sur les bits du sprite : si c'est 1 on affiche du blanc, si c'est 0 on affiche du noir. À noter que cette routine ne met pas à jour la position du curseur.

Attention : les fonctions d'affichage de caractères NE vérifient PAS la bonne position du curseur. En d'autres termes, si la commande `println` arrive à la fin de l'écran, le texte sera ensuite affiché à l'extérieur de l'écran et donc à des adresses mémoire qui ne font pas partie de la VRAM. De même pour la commande display, l'implémentation en assembleur ne fait aucune vérification quant à la validité des coordonnées passées en Y et en X. Cela n'est pas dangereux la plupart du temps puisque les zones mémoires inférieure et supérieure à la VRAM sont inutilisées, par conséquent une écriture n'aura aucun effet. Mais cela peut être plus problématique au bout d'une centaine d'écriture.

Afficher des entiers

À partir de ce que nous avons vu dans la partie précédente, nous pouvons en déduire une façon assez simple d'afficher des variables entières à l'écran.

En effet, parmi les caractères que nous stockons en mémoire, nous avons les chiffres de 0 à 9, dont les numéros vont, dans l'ordre, de 0x10 à 0x19. La routine d'affichage d'entiers fonctionne selon l'algorithme suivant :

1. R1 est un entier à afficher
2. R2 l'adresse d'une saferam, c'est-à-dire un endroit de la RAM sur lequel on peut écrire temporairement. [R2] signifie "La valeur stockée à l'adresse contenu dans R2".
3. Tant que R1!=0
4. Mettre dans [R2] `0x10+R1%10`
5. On met la partie entière de `R1/10` dans R1
6. Décrémenter R2
7. Fin Tant que
8. Incrémenter R2
9. Mettre R2 dans R1 (le paramètre de `printStr`)
10. Appeler la routine d'affichage de caractères `printStr`.

Gestion du Pad

La Game Boy Advance ne possède pas de clavier pour les entrées/sorties, mais un Pad.

Le Pad possède 10 boutons : A, B, Gauche, Droite, Haut, Bas, Select, Start, L et R. Ces boutons sont les seules entrées (avec On/Off et le son) de la GBA. Les sorties, quant à elles, sont l'écran et le système de haut-parleur dont nous ne nous occupons pas dans ce projet.

Comme expliqué dans l'analyse bibliographique, les données du Pad sont accessibles à une certaine adresse mémoire (sur deux octets). Ces données étant nécessaires pour la fonction `getKey` de Decac, nous avons fait une routine pour les récupérer et les inverser. En effet, ces données sont stockées sous la forme ACTIVE-LOW. Afin de rendre l'approche utilisateur plus intuitive, nous avons donc inversé les 10 premiers bits de ce nombre.

Méthode de validation

Environnement de développement

Toolchain

Afin de vérifier que notre compilateur fonctionne correctement, nous avons besoin d'un environnement de développement capable de créer, à partir d'un fichier `.deca`, un fichier `.gba`, puis d'émuler ce fichier `.gba`. La toolchain de développement Décac pour Game Boy Advance est composée :

1. De l'exécutable `decac`
2. D'un assembleur pour GBA
3. D'un émulateur GBA
4. Éventuellement d'une cartouche GBA spéciale pour tester directement sur la console.

C'est grâce aux trois logiciels que nous avons pu exécuter des programmes déca pour GBA. Et c'est grâce au point 4 que nous avons pu lancer nos programmes sur machine réelle. À titre de comparaison, l'exécutable IMA s'occupait directement des points 2 et 3. Les parties suivantes décrivent quels sont les logiciels que nous avons choisis et comment les utiliser.

Assembleur : `devkitPro`

Un programme assembleur est un programme permettant de convertir un fichier assembleur en fichier binaire exécutable. Son rôle est similaire à celui d'un compilateur.

Pour ce projet, nous avons décidé d'utiliser `devkitPro`. C'est un environnement de développement pour console Nintendo (switch, DSi, GBA…). Un tutoriel d'installation est disponible sur leur wiki (voir la partie "Lien" à la fin de ce document).

Pour assembler un fichier test.asm, on utilise les commandes :

```
arm-none-eabi-gcc -mthumb-interwork -specs=gba.specs test.asm

arm-none-eabi-objcopy -O binary a.out a.gba
```

Puis on émule avec (voir point suivant) :

```
vba a.gba
```

Emulateur : `VisualBoyAdvance-M`

Nous avons choisi d'utiliser `VisualBoyAdvance-M` (VBAM) qui est réputé pour être complet, facile d'accès et utilisable sur Linux. Cet émulateur est compilable à partir du projet Github (voir partie "Lien" à la fin de ce document).

Pour émuler une ROM `a.gba`, on utilise la commande suivante :

```
vba a.gba
```

Linker : EZ-Flash IV

Un EZ-Flash est une sorte de cartouche programmable, permettant de faire exécuter à une console Game Boy Advance des “ROM” `.gba`. Munie d’une carte microSD, il suffit de mettre le programme sur la carte SD pour pouvoir le lancer sur la calculatrice. Grâce à cette cartouche, nous avons pu essayer nos programmes directement sur console GBA.

Tests et validation

Général

Les fonctionnalités Decac était en grande partie les mêmes que sur IMA, nous n’avons pas eu besoin de créer un grand nombre de tests Decac supplémentaires. Nous avons donc pu utiliser ces tests pour vérifier le bon fonctionnement de la génération de code pour GBA. Une attention particulière a tout de même été portée aux fonctions propres à la console, à savoir `getKey` et `display`, présentées dans la partie “Spécifications”.

Ces tests ont pour la plupart été faits sur l’émulateur VisualBoyAdvance-M dont un descriptif est disponible dans la partie précédente. Comme nous l’avions énoncé dans l’introduction, certains ont aussi été testés directement sur calculatrice grâce à un EZ-Flash IV, qui est une sorte de cartouche de jeu programmable (voir ci-dessus).

Automatisation

Afin de gagner du temps lors des tests, et puisque les commandes à taper pour assembler les fichiers assembleur est complexe, nous avons fait un Makefile qui automatise ces tâches.

Ce Makefile est disponible à la racine du projet. Il permet soit de seulement compiler un fichier Decac spécifié dans la variable `TARGET`, soit de le compiler et l’émuler avec la commande `vba`.

Pour compiler un fichier `src/tests/hello.deca` on fait :

```
make compile TARGET=src/tests/hello
```

Cela affichera le code généré par le compilateur et créera un fichier `a.gba` dans le même dossier que le fichier `.decac`.

Pour compiler ET émuler un fichier `src/tests/hello.deca` on fait :

```
make run TARGET=src/tests/hello
```

Résultats de validation

La validation du compilateur pour l'extension dépend de plusieurs paramètres :

- Le compilateur, si on lui indique l'option `--gba`, génère un fichier `.s` qui pourra ensuite être utilisé par un programme assembleur pour générer un fichier binaire compatible avec une console GBA.
- Plus généralement, l'interface utilisateur du compilateur respecte les spécifications imposées par le cahier des charges. Entre autres choses, il génère un fichier assembleur OU affiche une erreur. Cette partie est totalement commune avec les spécifications IMA : c'est pourquoi nous n'avons pas eu besoin de tester plus en détail ce point par rapport à ce qui a été fait pour IMA.
- La correction du programme assembleur est assurée : c'est-à-dire que si le compilateur ne retourne pas d'erreur ou d'avertissement, le programme dans le fichier .s est un programme dont on assure qu'il est équivalent (aux erreurs d'exécution près) à celui écrit en Decac. C'est ce point, en particulier, que nous avons dû tester.
- Lorsqu'un événement inattendu survient lors de l'exécution risquant de compromettre la correction d'un programme, une erreur est affichée (division par 0, Stack Overflow…).

C'est en effectuant les tests présentés dans la partie précédente que nous avons pu nous assurer de la validité de notre compilateur pour l'extension GBA.

Liens

Ces liens ont été consultés pour la dernière fois entre le 24 et le 29 janvier 2021.

Tutoriel présentant les notions basiques de l'assembleur pour GBA
https://patater.com/gbaguy/gbaasm.htm

Tutoriel présentant des notions plus avancées
https://www.coranac.com/tonc/text/toc.htm

Projet GBATek (site documentant de nombreux aspects de la GBA)
https://mgba-emu.github.io/gbatek/

User manual de l'ARM7STDI
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0210c/index.html

Généralité sur le matériel de la GBA
https://fr.wikipedia.org/wiki/Game_Boy_Advance

Wiki de l'environnement de développement `dekitPro`
https://devkitpro.org/wiki/Main_Page

Projet Github de l'émulateur `VisualBoyAdvance-M`
https://github.com/visualboyadvance-m/visualboyadvance-m/