

Lecture et représentation d'un programme assembleur MIPS

Emel Hajjem ; Victor Lambret

2011

Abstract

Ce rapport présente la spécification, la réalisation et la procédure de recette de notre projet PSESI dont le but est de fournir une représentation d'un programme assembleur MIPS et un ensemble de fonctions de manipulation de cette représentation.

Contents

I	Spécification	4
1	Contexte	4
2	Objectifs	4
3	Organisation	5
3.1	Méthode de travail	5
3.2	Liste des tâches	5
3.2.1	Structures de donnée	5
3.2.2	Implémentation des fonctionnalités de manipulation	6
3.2.3	Parsing	6
3.3	Outils	6
3.4	Planning	6
4	Recette	7
4.1	Documents fournis	7
4.2	Procédure de recette	7
4.2.1	Acquisition d'un programme avec le parseur et restitution	7
4.2.2	Manipulation d'un programme	7
4.2.3	Manipulation d'instructions	8
4.2.4	Calcul de dépendance entre instructions	8
II	Réalisation	8
5	Classe de bases	9
6	Index	10
6.1	Fonction	10
6.2	Bloc de base	11
7	Calcul de dépendance	12
7.1	Read after Write	12
7.2	Write after Read	12
7.3	Write after Write	13
8	Restitution	13
9	Parseur	13
9.1	Départ d'un parseur existant	13
9.2	Le passage au C++	13
9.3	Intégration à la bibliothèque	14
9.3.1	Adaptations	14
9.3.2	Tests	14
9.3.3	Fonctionnement final	14
III	Recette	14
10	Planning	15
11	Documents fournis	15
12	Procédure de recette	16
12.1	Acquisition d'un programme avec le parseur et restitution	16
12.1.1	Méthode de test	16
12.1.2	Résultats	16
13	Manipulation d'un programme	17

14 Manipulation d'instructions	17
14.1 Calcul de dépendance entre instructions	17
15 Conformité à la spécification	18
16 Extensions possibles	18
IV Conclusion	18

Part I

Spécification

1 Contexte

Le but de notre projet est de fournir une représentation d'un programme assembleur MIPS. Aussi, nous lui ajouterons un ensemble de fonctionnalités de manipulation de programme. Celles-ci constitueront une brique de base qui permettra des développements ultérieurs, notamment dans le but de calculer des temps d'exécution ou de mettre en oeuvre des techniques d'optimisation ou d'analyse de code. Cet outil sera étendu puis utilisé dans le cadre des enseignements de M1 d'architecture ou de compilation avancée.

Ce projet sera réalisé en binôme, Victor Lambret et Emel Hajjem et encadré par Karine Heydemann.

Il faut savoir que ce projet demande ou développera des connaissances autour des notions suivantes.

- Le jeu d'instruction MIPS étant donné que l'on va manipuler du code assembleur MIPS.
- Un langage de programmation qui permettra l'implémentation de la solution retenue pour ce projet.
- L'anglais technique pour la rédaction des commentaires et de la documentation.
- Des techniques et outils de parsing, qui permettront de lire un fichier assembleur.
- La structure d'un programme assembleur et des notions relatives à la compilation.

2 Objectifs

L'objectif du projet est de récupérer un code assembleur MIPS écrit à la main ou produit par un compilateur gcc, via un parseur, et de le représenter en mémoire grâce à des structures de données. Les manipulations de code suivantes sont aussi souhaitées.

- Modification d'un opérande.
- Ajout et suppression d'instructions.
- Calcul de dépendance entre deux instructions.
- Calcul des blocs de base d'une fonction.

Enfin l'outil réalisé doit être capable de reproduire le code initial ou modifié dans un fichier.

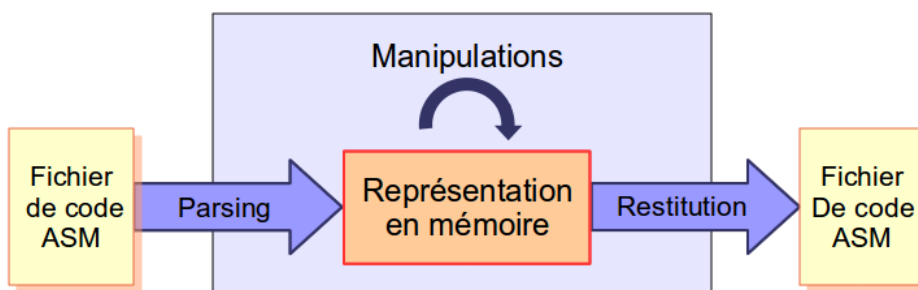


Figure 1: Schéma de fonctionnement de la bibliothèque

C'est dans cette perspective que nous allons réaliser une bibliothèque, afin qu'un développeur puisse inclure facilement les fonctionnalités que nous proposons dans son programme.

Il faut donc :

- créer un parseur qui lira le code assembleur initial dans un fichier,
- définir et implanter des structures de données appropriées pour la représentation du programme,
- définir et implanter les opérations de manipulation,
- fournir une bibliothèque permettant de récupérer la représentation de la manipuler et d'en générer un fichier.

3 Organisation

3.1 Méthode de travail

Notre projet consistant à développer un même outil à deux avec une interdépendance entre les tâches à réaliser, nous allons utiliser un système de gestion de versions pour une collaboration plus aisée. De plus, nous allons adopter des méthodes de travail inspirées de l'Extreme Programming^{[http://fr.wikipedia.org/wiki/EXtreme_Programming]}, consistant à :

- privilégier les solutions les plus simples pendant le développement et optimiser à la fin du projet en fonction du besoin,
- découper le travail en cycles de développement courts de manière à intégrer au plus vite les changements dans l'archivage,
- valider le bon fonctionnement des différentes parties de la librairie par des tests unitaires,
- définir et utiliser un ensemble de règles communes de codage (nommage, commentaire...).

Toute réalisation de code suivra le schéma suivant :

1. Écriture de tests.
2. Écriture du code et de la documentation de la classe associée.
3. Validation par un passage réussi des tests.

En particulier nous nous répartirons le travail entre l'écriture des tests et du code.

3.2 Liste des tâches

Nous pouvons découper notre travail en trois grandes tâches pour réaliser nos objectifs.

3.2.1 Structures de donnée

Pour écrire ces structures de données nous avons choisi de les implémenter grâce au langage C++.

Mais avant de passer à la programmation, il faut dans un premier temps définir ces structures. Pour mieux les visualiser, nous allons les représenter dans un diagramme UML.

Comme nous voulons offrir la possibilité de reproduire le code, il faut conserver toutes les informations. Nous avons donc décidé de voir un programme comme un ensemble de lignes. Parmi ces lignes, seules celles correspondant aux instructions nous intéressent. Toutefois, pour des développements ultérieurs, il est utile de pouvoir accéder à toutes les informations (toutes les lignes) et donc de conserver celles-ci. Dans le but d'analyse de code ultérieure, on souhaite connaître pour chaque instruction son code opérateur et pouvoir le modifier, sa nature (opération arithmétique ou logique, opération mémoire ou branchement) ainsi que ses opérateurs. Les opérandes sont soit des registres, des immédiats des expressions ou des labels. Les labels sont aussi importants, ils indiquent notamment le début des fonctions et des certains blocs de base. Ainsi, nous avons identifié les entités suivantes pour lesquelles nous développerons des structures de données.

- Programme : entité correspondant à un ensemble de lignes, parmi lesquelles nous identifions celles correspondant au début de fonction en les conservant afin d'avoir un accès direct aux fonctions.
- Ligne : entité correspondant à une ligne du code asm d'origine, pouvant être une directive, une déclaration, un label ou une instruction.
- Label : entité composée d'un nom.
- Instruction : entité composée d'un code opération et d'opérandes. Nous conservons son format et déduirons son type du code opération.
- Opérande : entité correspondant à un registre (son numéro), un label (chaîne ou identifiant), un immédiat (valeur numérique) ou une expression.
- Découpage en sous-tâches :
- Diagramme : réalisation du diagramme UML précis des classes utilisées.
- Classes de base : réalisation des classes Programme, Ligne, Instruction, Label et Opérande.

3.2.2 Implémentation des fonctionnalités de manipulation

Après avoir défini nos structures de données, nous pourrions passer à la définition et à l'implémentation des différentes fonctions qui vont permettre de manipuler le code initial.

Découpage en sous-tâches :

- Calcul de dépendance : réalisation de la fonctionnalité de calcul de dépendance entre deux instructions.
- Index : réalisation de la fonctionnalité de calcul des blocs de base et de fonctions d'accès indexées aux lignes du programme par les labels, blocs de base ou fonctions.
- Restitution : réalisation de la fonctionnalité permettant d'écrire la totalité ou une partie du programme sous forme d'un fichier d'instructions assembleur.

3.2.3 Parsing

Notre projet nécessite un parseur qui fera l'analyse lexicale et syntaxique du code assembleur MIPS donné en entrée. L'écriture d'un parseur étant une tâche qui prendrait beaucoup de temps, nous allons réutiliser un parseur écrit par Mr Pirouz acceptant en entrée des fichiers assembleur MIPS produit par gcc. Nous aurons à comprendre, puis nettoyer celui-ci avant de modifier les actions afin de produire la représentation du programme avec nos structures de données.

Découpage en sous-tâches :

- Nettoyage : nettoyage du parseur existant pour en conserver les règles d'analyse.
- Parsing : construction de l'entité programme à partir du parsing d'un fichier assembleur.

3.3 Outils

Notre projet doit pouvoir être utilisé et adapté facilement, nous allons donc le développer grâce à des outils standards et libres :

- g++ pour compiler la librairie et le code de test(C++),
- gccpour mips installé via crosstools pour compiler les programmes d'exemples écrits en C vers de l'assembleur MIPS,
- make pour la gestion du projet,
- un gestionnaire de version comme git,
- Doxygen pour générer la documentation en anglais à partir de commentaires bien écrits dans le code du projet.

3.4 Planning

Nos tâches étant assez dépendantes entre elles, nous nous sommes répartis le travail pour le démarrage du projet. Il nous est assez difficile de donner une estimation précise du temps que prendra chaque tâche, c'est pourquoi les tâches suivantes seront attribuées en fonction de l'avancement.

L'ordre de réalisation des tâches sera dicté par les dépendances que nous avons identifiées dans ce schéma :

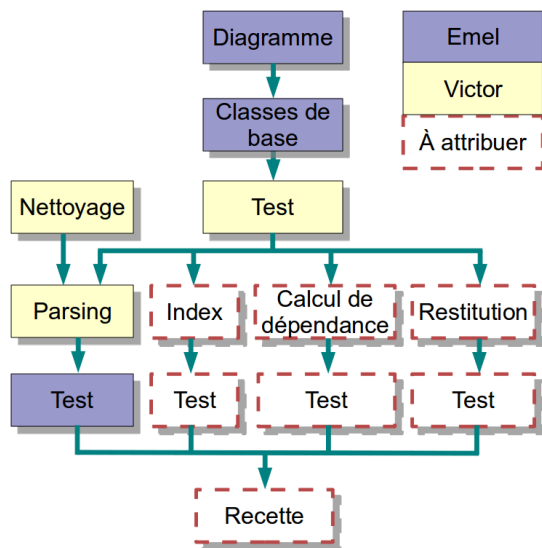


Figure 2: Planning

4 Recette

4.1 Documents fournis

A la fin de notre projet, nous produirons les documents suivants.

- Le diagramme de classes UML du projet.
- Le code de la bibliothèque de base.
- Le code des classes de tests unitaires et de l'ensemble des outils de test.
- La documentation de la bibliothèque générée par Doxygen.

4.2 Procédure de recette

La librairie produite sera validée par deux procédures de recette. Tout d'abord une recette continue composée de test unitaires, de nos revues de code et de la validation des étapes par notre encadrante.

La seconde partie de la recette sera basée sur des scénarios testant chacun un ensemble de fonctionnalités.

4.2.1 Acquisition d'un programme avec le parseur et restitution

Nous allons écrire trois petits programmes en langage C, typiquement de parcours de tableau, de multiplication de matrices ou de tri rapide. Ces programmes seront compilés vers de l'assembleur avec gcc puis représentés en mémoire en utilisant le parseur avant d'être restitués en assembleur. Nous vérifierons que les programmes restitués sont identiques aux programmes d'origine, en particulier qu'ils peuvent toujours être compilés et produire le même code exécutable.

4.2.2 Manipulation d'un programme

A partir de ces trois même programmes d'exemple, nous effectuerons les opérations suivantes.

- Affichage des informations générales du programme comme le nombre de lignes et d'instructions.
- Calcul du découpage en blocs de base et en fonctions.
- Écriture dans un fichier des séquences de lignes de ces blocs de base et fonctions dans des fichiers différents.

4.2.3 Manipulation d'instructions

```
1 # Code avec des Nor
2 add    $5, $0, $0
3 or     $6, $5, $0
4 nor    $4, $5, $6
```

Nous allons démontrer les possibilités de manipulation, insertion et suppression de lignes en transformant la séquence d'instructions suivante :

```
# Code avec des Nor # ligne 1
add    $5, $0, $0 # ligne 2
or     $6, $5, $0 # ligne 3
nor    $4, $5, $6 # ligne 4
```

en effectuant les opérations suivante :

- les lignes 1 et 2 seront supprimées,
- les opérandes et l'opérateur de la ligne 3 seront modifiés pour obtenir la ligne 1bis,
- l'opérateur de la ligne 4 sera modifié pour obtenir la ligne 5bis,
- les lignes 2, 3, et 4bis seront ensuite insérées dans le programme.

et nous obtiendrons le programme suivant :

```
lui $4, 0xFFFF # ligne 1
ori $4, $4, 0xFFFF # ligne 2
xor $5, $4, $0 # ligne 3
xor $6, $4, $0 # ligne 4
and $4, $5, $6 # ligne 5
```

4.2.4 Calcul de dépendance entre instructions

En partant de cette séquence d'instructions :

```
# Exemple de programme utilisé pour la recette
1 lw $4, 0($0)
2 lw $4, 0($8) # Write after Write avec 1
3 addi $5, $4, 10 # Read after Write avec 2
4 sw $4, 0($8)
5 sub $4, $10, $5 # Write after Read avec 4 et 3
6 addi $31, $31, 4 # instruction indépendante
```

Nous effectuerons les opérations suivantes :

- Calcul et affichage des dépendances entre instructions.
- Inversion de deux instructions indépendantes.
- Production du fichier assembleur résultat.

Part II

Réalisation

Une fois les spécifications établies, nous sommes donc passés à la réalisation de notre bibliothèque. L'objectif du projet étant de lire un code assembleur MIPS écrit à la main ou produit par gcc, via un parseur, de le représenter en mémoire et d'effectuer dessus des manipulations, nous avons défini trois grandes

tâches : définition et implantation des structures de données, l'implémentation des fonctionnalités de manipulation et le parsing. Nous vous présentons dans cette partie la réalisations de ces tâches.

5 Classe de bases

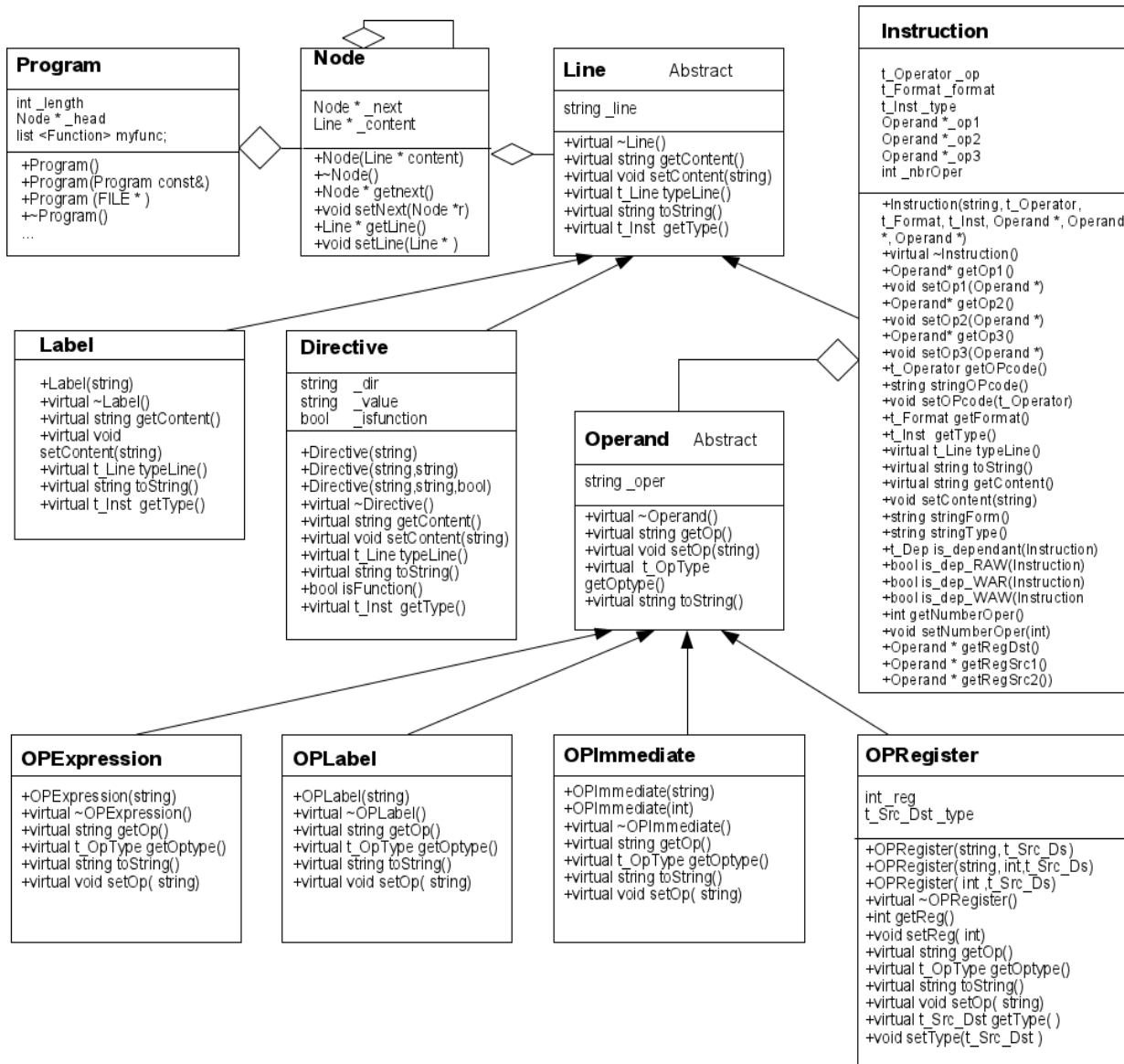


Figure 3: Diagramme UML

Les structures de données représentent le programme contenu dans un fichier assembleur (données, code). Ces structures vont être réparties en classes représentant les différents objets qui caractérisent un code assembleur. Nous avons donc construit nos classes de bases de manière hiérarchique. Afin de mieux visualiser cette hiérarchie, nous avons établi le diagramme UML ci-dessous qui représente l'imbrication des classes.

Tout d'abord nous avons le programme que nous avons choisi de voir comme une suite de lignes qui peuvent être de différents types : labels, instructions et directives. Les instructions sont des lignes particulières, elles ont chacune un code opérateur, des opérandes, un format de codage (R : Register, I : Immédiat, J : Jump, O : Other) et un type (BR : Branchement, MEM : Mémoire, ALU : Arithmétique et logique, OTHER : autre correspondant aux instructions relative au système.)

- Le programme est vu comme une liste chaînée de lignes. La classe Node correspond aux noeuds de la liste chaînée. Dans ce programme, nous pouvons supprimer ou ajouter des lignes. Nous pouvons aussi connaître le nombre de ligne du programme, l'afficher et chercher une ligne en particulier.
- La classe Line correspond à l'information contenue dans un noeud de la liste chaînée. C'est une classe abstraite qui permet aux sous-classes Instruction, Directive et Label d'hériter de ses caractéristiques. Chaque ligne a

pour attribut une chaîne de caractère. Nous pouvons modifier cette ligne, récupérer la ligne ou son type (instruction, label ou directive).

- Les directives correspondent aux lignes de déclaration de zone (.data, .text) de données, de fonction. La directive peut être suivi d'un champs. Cette classe peut identifier le début et la fin d'une fonction(.ent, .end), ce qui est utile lors du calcul des fonctions.
- Les labels sont les étiquettes utilisées pour désigner des données ou instructions dans le code.
- Les instructions correspondent au code proprement dit. Ce sont, pour nous, les plus importantes. Une instruction contient un code opérateur, des opérandes, son type et son format. Dans cette classe nous pouvons récupérer ses caractéristiques mais nous ne pouvons modifier que son code opérateur et ses opérandes car le type et le format sont déduit de l'opérateur et des opérandes.
- Le code opérateur est représenté par un type énuméré, qui lui même représente toutes les mémoriques assembleur.
- Les opérandes sont des objets dérivant de la classe abstraite Operand. Il y a une classe par type d'opérandes possibles : registre, immédiat, expression ou label. Pour toutes ces classes nous pouvons modifier l'opérande et la récupérer, elle et son type (registre, label, expression ou immédiat).

6 Index

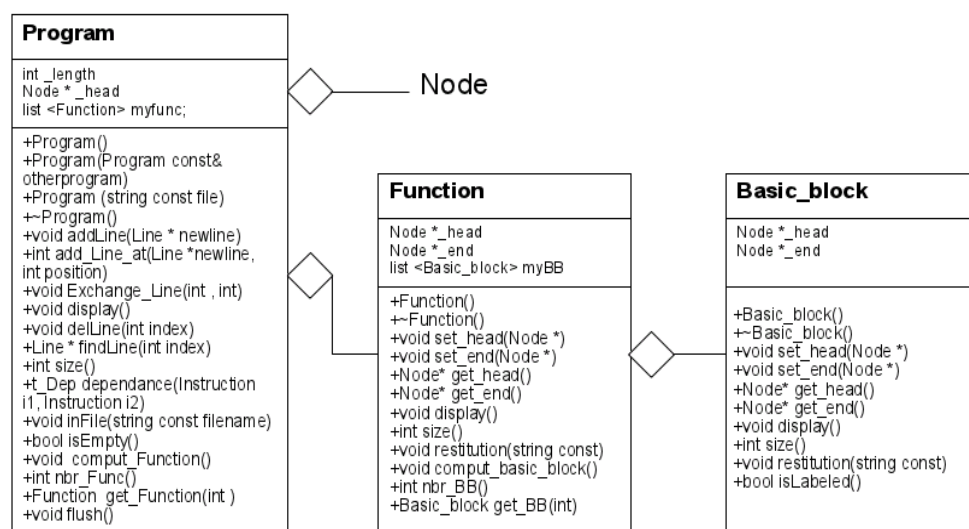


Figure 4: Diagramme UML index

La fonctionnalité d'indexation permet d'accéder aux fonctions d'un programme et d'accéder aux blocs de bases d'une fonction.

6.1 Fonction

Un programme peut contenir plusieurs fonctions. Nous avons donc implémenté une fonctionnalité permettant de délimiter ces fonctions. Le calcul des fonctions se fait dans la classe Program qui a une vue d'ensemble de tout le code assembleur, donc de toutes les fonctions définies. On conserve ces fonctions dans une liste stockée dans le programme correspondant. La classe Function représente une fonction du programme. Elle contient un pointeur sur la ligne de début et la ligne de fin de la fonction. La classe offre la possibilité d'afficher le contenu de la fonction, de restituer le code lui correspondant dans un fichier et de calculer le nombre de lignes qu'elle contient.

La méthode permettant de calculer les fonctions parcourt la liste chaînée, correspondant au programme, affecte au noeud de début de la fonction le noeud du programme contenant la directive d'entrée à la fonction .ent et affecte au noeud de fin de la fonction le noeud du programme contenant la directive de sortie de la fonction .end. Ensuite, elle rajoute cette fonction à la liste des fonctions du programme, et continue pour trouver la prochaine fonction.

Voici l'algorithme en pseudo code du calcul des fonctions d'un programme :

```
compute_function()
begin
Function function
  if (program.is_not_Empty()) then
    Node current_node<-program.head while (current_node) then
      /* la directive de fin correspondante est .end*/
      if (current_node = directive_of_end_function) then
        function.end <-current_node
        program.add_list_function ( function)
      end if
      /* la directive de début correspondante est .ent*/
      if (current_node = directive_of_beginning_function) then
        function.head <- current_node
      end if
      if (current_node.next_node = program.end) then
        break
      else current_node <- current_node.next_node
      end if
    end while
  end
end
fin
```

6.2 Bloc de base

Dans chacune des fonctions d'un programme il y a plusieurs blocs de bases. Ces blocs de bases sont des séquences d'instructions contenant un seul point d'entrée et un seul point de sortie. La classe Basic_block représente un bloc de base. Elle contient un pointeur sur la ligne de début (une instruction ou un label) et la ligne de fin du bloc. Cette classe offre la possibilité d'afficher le contenu du bloc, le restituer dans un fichier et calculer le nombre de lignes qu'il contient. Le calcul des blocs de bases se fait dans la classe Function, qui va conserver ces blocs dans une liste.

La méthode qui permet de calculer les blocs de base va chercher dans la fonction les noeuds correspondant au point d'entrée et de sortie d'un bloc. Le point d'entrée d'un bloc est soit un label, soit la ligne suivant un delayed slot. Le point de sortie est soit l'instruction précédant un label, soit le delayed slot suivant un branchement. Dans notre cas le delayed slot est de 1. Après avoir trouvé le point de sortie d'un bloc on ajoute le bloc à la liste des blocs contenu dans la classe Function.

Voici l'algorithme en pseudo code du calcul des blocs de bases dans une fonction:

```
compute_Basic_block()
begin
  Node current_node <- function.head
  block.head <- function.first_node_label_or_instruction
  while (current_node != function.end) then
    if (current_node = instruction)
      if (current_node = branch) then
        current_node <- current.Nem_next_node /*N delayed slot*/
        block.end <- current_node
        function.add_list_block ( block)
        if (current_node.next_node != function.end &&
            current_node.next_node._next_node != function.end)
          block.head <- current_node.next_node
          /*the node next the Nth delayed slot*/
          current_node <- current_node.next_node /*block.head*/
        else break
      end if
    end if
  end while
end
```

```

end if

if (current_node.next_node = label) then
    block.end <- current_node
    function.add_list_block ( block )
    block.head <- current_node.next_node
end if

if (current_node.next_node = function.end) then
    block.end <- current_node
    function.add_list_block ( block )
    break
else current_node <- current_node.next_node
end if
end while
end

```

7 Calcul de dépendance

Nous avons réalisé la fonctionnalité qui permet de calculer la dépendance entre deux instructions. Pour cela nous avons implémenté des méthodes correspondant à chacune des dépendances possibles dans la classe Instruction:

- RAW: détection des dépendances lecture après écriture entre deux instructions i1 et i2, c'est à dire la lecture d'un registre par i2 après son écriture par i1.
- WAR: détection des dépendances écriture après lecture entre deux instructions i1 et i2, c'est à dire l'écriture dans un registre par i2 après sa lecture par i1.
- WAW: détection des dépendances écriture après écriture entre deux instructions i1 et i2, c'est à dire l'écriture d'un registre par i2 après l'écriture par i1.

Ces méthodes testent si la dépendance est vérifiée entre l'instruction courante et celle donnée en paramètre. Nous avons aussi implémenté une méthode permettant de calculer la dépendance entre deux instructions particulières dans un programme.

Voici les algorithmes pour chacun des types de calcul des dépendances dans la classe instruction:

7.1 Read after Write

Cette fonction renvoie vrai si il y a une dépendance de type WAR entre l'instruction courante et i2 passé en paramètre, sinon retourne faux.

```

RAW(Instruction i2)
begin
    if (self.RegDest && i2.RegSrc1 && self.RegDest==i2.RegSrc1)      return true
    if (self.RegDest && i2.RegSrc2 && self.RegDest==i2.RegSrc2)      return true
    return false
end

```

7.2 Write after Read

Cette fonction renvoie vrai si il y a une dépendance de type WAR entre l'instruction courante et i2 passé en paramètre, sinon retourne faux.

```

WAR(Instruction i2)
begin
    if (i2.RegDest && self.RegSrc1 && i2.RegDest==self.RegSrc1)      return true
    if (i2.RegDest && self.RegSrc2 && i2.RegDest==self.RegSrc2)      return true

```

```

        return false
end

```

7.3 Write after Write

Cette fonction renvoie vrai si il y a une dépendance de type WAW entre l'instruction courante et i2 passé en paramètre, sinon retourne faux.

```

WAW(Instruction i2)
begin
    if (self.RegDest && i2.RegDest && self.RegDest==i2.RegDest)    return true
    return false
end

```

8 Restitution

Nous avons réalisé la fonctionnalité qui permet de restituer un programme, une fonction ou un bloc de base sous forme d'un fichier d'instruction assembleur. La restitution se fait dans une méthode de chacune des classes associée. Ces méthodes prennent en paramètre le nom du fichier dans lequel on veut écrire. On parcourt la structure, récupère la chaîne de caractère de chaque ligne et on l'écrit dans le fichier.

9 Parseur

9.1 Départ d'un parseur existant

Nous sommes partis d'un outil existant écrit par Mr Bazargan Sabet utilisant un parseur pour construire une représentation en mémoire d'un programme assembleur mips. La partie parsing était basée sur *lex* comme analyseur lexical et *yacc* comme analyseur syntaxique. Nous souhaitions récupérer les règles de découpage en mots clés ainsi que la grammaire et remplacer tous les actions du parseur par des appels aux constructeurs de notre bibliothèque de base.

Dans un premier temps nous avons cherché à isoler le code de ce parseur du reste du simulateur afin de le compiler avant de l'adapter à notre besoin. Cette étape nous a fait conserver uniquement 3 éléments :

- les règles dans deux fichiers *lex* et *yacc*,
- une bibliothèque asm contenant les fonctions utiles pour les actions *yacc*,
- une bibliothèque utl de fonctions de manipulations de données pour le passage de données entre *lex* et *yacc*, ou entre actions *yacc*.

Dans un premier temps nous avons donc commenté les actions existantes pour n'avoir qu'un simple affichage du nom de la règle parcourue afin de tester la seule compilation du code et comprendre les différentes règles de parsing.

9.2 Le passage au C++

Pour que notre parseur fasse appel à des constructeur C++ dans les actions nous avons eu besoin de compiler le fichier yacc avec *g++*. A priori *g++* pouvant compiler du C nous avons d'abord cherché à lui faire compiler le code passant avec *gcc*. Avant d'y arriver nous avons rencontré deux types de problèmes :

- *g++* est plus exigeant sur les types et n'accepte pas l'ancienne syntaxe C pour déclarer les paramètres des fonctions. Nous avons donc mis à jour les entêtes et les corps des fonctions des bibliothèques et corrigé les incohérences de type dans le code.
- L'usage de C++ dans le fichier yacc a posé un problème long à résoudre : le mécanisme de dialogue entre Lex et Yacc ne marchait plus. Les deux logiciels étant des projets différents, et le support du C++ évoluant suivant les versions, l'information ne fut pas facile à trouver. Lors de cette étape *git* fut très utile pour créer une branche par solution essayée.

Une fois ces deux points résolus, nous avons pu utiliser C++ librement.

9.3 Intégration à la bibliothèque

9.3.1 Adaptations

Le parseur étant initialement prévu pour de la simulation, son comportement n'était pas adapté pour notre besoin de représenter le code d'un programme :

- Les règles initiales transformaient les données au plus tôt pour les mettre dans des structures de données dédiées à la simulation, souvent sous forme binaire. Nous avons cherché à faire remonter les données jusqu'aux règles appelants les constructeurs. Pour préserver les données comme du texte autant que possible, nous avons placé les chaînes de caractères (par exemple les labels) dans un tableau associatif dynamique puis fait remonter un identifiant numérique, la valeur étant ensuite récupérée par le constructeur via cette clé.
- Les expressions étant évaluées, il aurait été long de chercher à les conserver sous forme de chaîne de caractères, nous les avons donc conservées sous leur forme numérique, perdant en lisibilité sur le code final. Par exemple `.mask 0x40000000,-4` sera transformé en `.mask 1073741824,-4` car la notation hexadécimale est évaluée au plus tôt comme sa valeur numérique décimale.
- L'analyseur lexical gérait plusieurs versions du jeu MIPS. Nous avons décidé de tout accepter dans notre parseur afin de simplifier le test des règles et de reconnaître le plus de codes possibles.
- Afin d'éviter d'avoir à gérer les pseudos-instructions qui auraient complexifié la bibliothèque, par exemple pour le calcul de dépendance, nous avons choisi d'étendre les macros. Par exemple, la macro `li $2,2` sera transformé en la suite d'instructions `lui $1,0 ; ori $2,$1,2` par le parseur.

Comme tous les constructeurs de la classe `Line` sont utilisés dans les actions du parseur nous avons ainsi bénéficié d'un premier retour critique sur leurs signatures. Dans certains cas nous avons modifié nos constructeurs pour gagner en simplicité et en sûreté d'utilisation. Par exemple au départ nous n'avions qu'un seul constructeur pour la classe `Instruction` et l'on passait plusieurs paramètres comme le type d'instruction ou le nombre d'opérandes. Afin de gagner en sûreté et en lisibilité nous avons préféré utiliser plusieurs constructeurs, un pour chaque nombre d'opérandes possible et déduire les autres informations à partir de l'opérateur ou des opérandes.

9.3.2 Tests

En parallèle de l'écriture d'une action pour une règle du parseur, nous avons complété un fichier `complete.s` permettant de tester tous les cas couverts par la règle. Ceci nous a permis de vérifier exhaustivement qu'entre l'acquisition du programme par le parseur, sa représentation dans les classes de base et sa restitution vers un fichier assembleur il n'y avait aucune erreur ou altération parasite.

Une fois ceci validé nous avons testé le parseur avec le fichier assembleur `primes.s` produit pour le premier test de la recette. Nous avons relevé de nombreuses erreurs venant d'une mauvaise utilisation des instructions par `gcc`, par exemple, un opérateur au format R utilisé au format I (`slt rd, rs, imm`). Dans un premier temps nous avons cherché comment désactiver ce comportement mais la chaîne `gcc` dispose de peu d'options pour modifier l'assembleur produit par son compilateur. Afin de pouvoir lancer les tests, nous avons donc ajouté de nouvelles règles reconnaissant ces formats invalides et mais construisant l'instruction valide correspondante (`slti` à la place de `slt` pour notre exemple).

Nous avons également eu besoin de nouveaux tokens dans le fichier `lex` pour gérer de manière plus générique les directives, les labels ou des syntaxe de `gcc` pour les expressions comme le `%hi(imm)`. En créant de nouveaux tokens nous avons pu les récupérer comme du texte sans toucher aux tokens déjà existants pour éviter les effets de bord.

9.3.3 Fonctionnement final

Le parseur est utilisé par un des constructeurs de la classe `Program` qui lui permet de s'initialiser à partir d'un fichier.

Part III

Recette

10 Planning

Nos tâches étant assez dépendantes entre elles, nous nous sommes donc répartis le travail pour le démarrage du projet. Ensuite, les tâches ont été attribuées en fonction de l'avancement de chacun. Voici le schéma représentant la répartition des tâches.

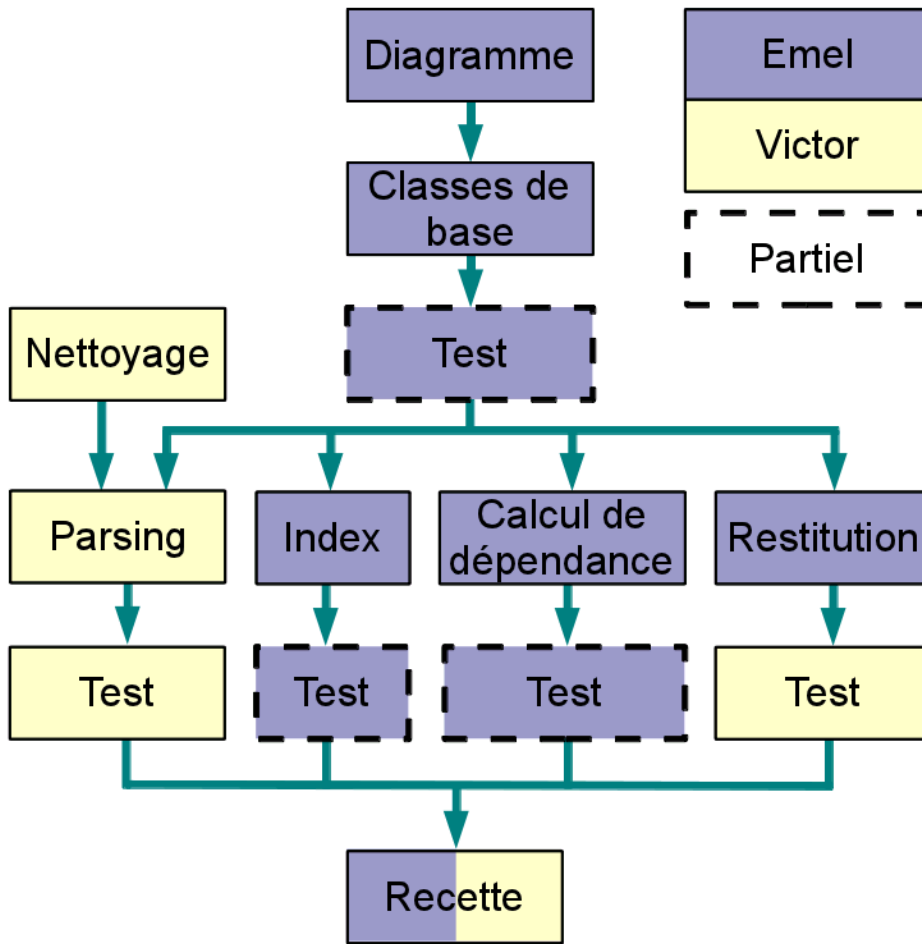


Figure 5: Planning final

Par rapport à la répartition initiale nous avons mal évalué la charge de travail de la partie parsing qui s'est avérée bien plus importante que prévue alors que nous pensions la terminer en quelques semaines. Suivant notre principe d'attribuer le travail à faire suivant ce qui est déjà fini Emel s'est retrouvé à réaliser toutes les fonctionnalités de la bibliothèque de base tandis que Victor a essentiellement travaillé sur le parsing, modifiant les classes de bases pour le besoin du parseur. Nous avons toutefois travaillé en équipe, en particulier pour la détection de bugs, l'évolution des fonctionnalités, la définition des classes et leur agencement.

11 Documents fournis

A la fin de notre projet, nous avons fourni les documents suivants à notre encadrante.

- Le diagramme de classes UML du projet.
- Le code de la bibliothèque de base.

- La documentation de la bibliothèque générée par *Doxygen*.

Par ailleurs, l'intégralité de notre projet peut être récupérée sur la page github du projet <http://github.com/VLambret/libASM>.

12 Procédure de recette

La librairie produite a été validée par deux procédures de recette. Tout d'abord une recette continue composée de nos revues de code et de la validation des étapes par notre encadrante.

La seconde partie de la recette est basée sur des scénarios testant chacun un ensemble de fonctionnalités.

12.1 Acquisition d'un programme avec le parseur et restitution

12.1.1 Méthode de test

Le but de ce test est de mesurer les changements produits par notre outils sur un programme entre son acquisition par le parseur et sa restitution. La comparaison des deux codes sources assembleurs en entrée et en sortie n'est pas pertinent à cause des changements induits par le parseur. Par exemple pour les registres *\$zero*, *zero* et *\$0* sont tous transformés en *\$0*. La solution retenue consiste donc à ne pas comparer les sources mais les codes objets compilés.

Pour cela nous avons écrit un script *diffobj.sh* qui effectue cette comparaison binaire. Voici son fonctionnement :

- Dans un premier temps il utilise *od* pour convertir les codes objets en fichiers texte contenant un mot de 32 bits en hexadécimal sur chaque ligne.
- Ensuite il utilise *objdump* pour obtenir un dictionnaire traduisant les mots binaires en code assembleur.
- Enfin il compare les mots hexadécimaux avec le programme *diff* et ajoute au résultat la traduction assembleur des mots concernés en utilisant le dictionnaire.

Lors des tests nous remplaçons les programmes prévus initialement afin de couvrir plus de cas différents :

- *test1a.c* : Un programme avec quelques opérations basiques pour tester une base minimale,
- *test1b.c* : Ce programme compte les nombres premiers dans un intervalle, il introduit des sauts conditionnels et des appels de fonction,
- *primes.c* : Ce programme affiche les nombres premiers dans un intervalle, il ajoute les appels à la bibliothèque standard.

Ces programmes sont compilés avec *mipsel-linux-gnu-gcc* version 4.5.2 pour le *MIPS R3000*.

12.1.2 Résultats

Le lancement des tests s'effectue via la commande *make test1*. Cette règle construit les fichiers assembleurs puis appelle le script *diffobj.sh* pour effectuer la comparaison. Le premier succès est que les codes restitués se compilent sans problèmes majeurs. La comparaison des codes objets montre par contre des différences dont voici les causes :

- Nous avons choisi d'étendre les macros dans le parseur, mais gcc ne choisit pas forcément les mêmes instructions. Par exemple pour la macro *li* (load immediate) gcc optimise dans le cas d'un immédiat de 16 bits en n'utilisant qu'une seule instruction ce que nous n'avons pas fait dans le parseur.
- Ce nombre différent d'instructions implique un décalage des adresses pour les sauts.

D'après nos tests nous pouvons donc dire que si nous n'avons pas une égalité des fichiers binaires produits ceux-ci sont par contre équivalents.

13 Manipulation d'un programme

Nous avons écrit un programme *test1.cpp* qui prend en entrée un fichier assembleur et qui effectue avec succès les opérations suivantes sur les trois fichiers assembleur produit précédemment :

- Affichage du programme et du nombre de lignes que contient celui-ci.
- Calcul du découpage en blocs de base et en fonctions.
- Écriture des séquences de lignes de ces blocs de base et fonctions dans les fichiers *test1.txt* et *test2.txt*.

14 Manipulation d'instructions

Nous démontrons les possibilités de manipulation, insertion et suppression de lignes en transformant la séquence d'instructions suivante présente dans le fichier *recette.s* :

```
# Code avec des Nor # ligne 1
add      $5, $0, $0 # ligne 2
or       $6, $5, $0 # ligne 3
nor      $4, $5, $6 # ligne 4
```

Le programme *test2.cpp* effectue les opérations suivante pour modifier le code :

- parse le fichier *recette.set* affiche le code ci-dessus,
- supprime les lignes 1 et 2 ,
- modifie les opérandes et l'opérateur de la ligne 3 pour obtenir la ligne 1bis,
- modifie l'opérateur de la ligne 4 pour obtenir la ligne 5bis,
- insère les lignes 2, 3, et 4bis dans le programme.

nous obtenons le programme suivant qui est l'affiché avec succès dans la console:

```
lui $4, 0xFFFF # ligne 1
ori $4, $4, 0xFFFF # ligne 2
xor $5, $4, $0 # ligne 3
xor $6, $4, $0 # ligne 4
and $4, $5, $6 # ligne 5
```

14.1 Calcul de dépendance entre instructions

Ce test est réalisé par le fichier *test_04.cpp*. Le programme initialise un programme à partir du fichier *test4.s* que voici :

```
# Exemple de programme utilisé pour la recette
1 lw      $4,      0($0)
2 lw      $4,      0($8)  # Write after Write avec 1
3 addi    $5,      $4,10   # Read after Write avec 2
4 sw      $4,      0($8)
5 sub     $4,      $10,$5   # Write after Read avec 4 et 3
6 addi    $31,     $31,4    # instruction indépendante
```

puis affiche l'ensemble des dépendances entre les instructions. Une vérification manuelle nous confirme que les calculs sont corrects.

```
      : 0   1   2   3   4   5
Ligne 0 : ———WAWRAWRAWWAWnone
Ligne 1 : -WAW——RAWRAWWAWnone
Ligne 2 : -WAR-WAR——none-WARnone
Ligne 3 : -WAR-WARnone——WARnone
Ligne 4 : -WAWWAWRAWRAW——none
Ligne 5 : nonenonenonenonenone——
```

Nous échangeons ensuite la première et la dernière instruction puis nous affichons de nouveau l'ensemble des dépendances. La vérification manuelle ne permet pas de relever d'incohérences.

```

      : 0   1   2   3   4   5
Ligne 0 : ——nonenonenonenonenone
Ligne 1 : none———RAW-RAW-WAWWAW
Ligne 2 : none—WAR———none—WAR-WAR
Ligne 3 : none—WARnone———WAR-WAR
Ligne 4 : none—WAWRAW-RAW———WAW
Ligne 5 : none—WAWRAW-RAW-WAW——

```

Le programme est restitué dans *tmp/test4.s*, les deux instructions sont bien échangées.

15 Conformité à la spécification

Malheureusement, lors de la réalisation de la bibliothèque certains problèmes ont ralenti l'implémentation et ont provoqué des changements par rapport aux spécifications fournis.

- Fonctionnalité d'indexation: l'indexation sur les labels fut écartée volontairement car jugée inutile par notre encadrante.
- Du fait du temps passé sur le parseur nous n'avons pas écrit les tests unitaires prévus pour les classes de base, qui furent remplacés par des tests fonctionnels et une vérification humaine moins chronophage.

16 Extensions possibles

Si notre bibliothèque est parfaitement utilisable en l'état, nous avons trouvé des axes d'améliorations possibles pour des développements ultérieurs :

- Rendre la bibliothèque paramétrable, par exemple sur le nombre de delayed slot ou sur les instructions acceptées.
- Actuellement l'opérateur d'une instruction est géré via un type énuméré et un tableau de caractéristiques associé. En gérant ces informations dans une classe, on aurait une manière uniforme de manipuler les opérateurs et le reste de la bibliothèque et plus de facilité à ajouter des informations grâce à l'objet.
- Supprimer les bibliothèques asm et utl qui rendaient des services utiles au parseur d'origine écrit en C mais perdent en intérêt pour notre bibliothèque qui peut les obtenir plus facilement par la bibliothèque standard C++ ou s'en passer. Plus globalement, reprendre le parseur pour lui faire manipuler uniquement du texte afin de rendre la restitution de programme plus fidèle à l'original.

Part IV

Conclusion

L'objectif de notre bibliothèque est rempli : un utilisateur peut récupérer notre bibliothèque et en installant uniquement des logiciels libres standards la compiler et l'utiliser pour son usage. La conception objet permet d'adapter ou d'étendre facilement les services rendus par notre bibliothèque et de l'utiliser facilement pour concevoir un nouvel outil s'appuyant sur nos classes.

Ce projet fut l'occasion de travailler en équipe sur un projet de développement concret tout en approfondissant nos connaissances sur le MIPS, le fonctionnement des outils de compilation comme sur l'usage d'outils collaboratifs.