Rapport
Projet Informatique : Assembleur MIPS

Livrable 4

Année Universitaire 2018 - 2019

BRITTON Giovani - Giovani-Crasby.Britton-Orozco@grenoble-inp.org
WRIGHT Gilles - gilles.wright@grenoble-inp.org

Professeur encadrant : Katell Morin-Allory - katell.morin-allory@imag.fr

Filière : SEI

I. Démarche Générale

Le livrable 4 du projet informatique a pour but la génération du code binaire et du fichier .o. Le code final doit pouvoir prendre en entrée un fichier au format .s contenant le code en langage assembleur mips, et générer en sortie un fichier .o au format ELF. POur cela, nous disposons de la bibliothèque ELF, qui permet de générer le fichier .o à partir du code binaire de nos différentes sections obtenues aux livrables précédents.
Nous avons d'abord dû commencer par compléter le contenu de notre livrable trois, en vérifiant toutes les opérandes des instructions, et en vérifiant le contenu de la table de relocation. Nous avons aussi par la suite remplacé toutes les opérandes au format de symbole par leur correspondance en adressage ou en décalage.
Pour la génération du code binaire, il a fallu créer un dictionnaire contenant le binaire associé à chaque instruction. Pour la section data, nous avons uniquement convertis les opérandes en binaire.
Afin de générer le fichier .o il faut respecter les contraintes de la bibliothèque ELF. Pour cela, on a stocké les données binaires correspondant aux sections text et data, dans un format adapté. De plus il a aussi fallu trouver certaines information nécessaire au fichier elf pour générer le .o.
Finalement, nous avons aussi affiner l'affichage à l'écran du contenue des différentes sections, étant données que nous ne sommes pas parvenue à compléter le livrable quatre pour générer le fichier .o. Nous avons aussi libérer l'espace mémoire occupé par toutes les listes que nous avions utilisées.

II. Modélisation du problème

Tout d'abord, concernant la fin du livrable 3, pour gérer la vérification des opérandes on a utilisé la fonction void extraction_des_operandes(Instruction* inst, ListeG Symb). Cette fonction vérifie la validité du type et de la tailles des opérandes opérandes de l'instruction pointées par inst, en comparant le type de l'opérande (trouvé lors de l'analyse lexicale), au type attendu pour l'instruction (cette information est contenue dans la structure de l'instruction, et récupéré pendant l'analyse grammaticale, lorsque l'existence de l'instruction est vérifiée). Si une des opérandes n'est pas du type attendu ou que sa taille n'est pas valide, une erreur est détectée et le programme s'arrête. Pour ce qui est de la relocation, nous utilisons les fonctions void relocationInst(ListeG Inst,ListeG* Symb,int ligne,ListeG* RelocInst) et void relocationData(ListeG Data,ListeG* Symb,int ligne,ListeG* RelocData) pour réaliser respectivement la relocation des instructions et des données data. Ces fonction ont un fonctionnement global similaire : lorsqu'une relocation est nécessaire, c'est à dire lorsqu 'un symbole est utilisé comme opérande à la place d'un immédiat, d'un absolue ou pour une directive .word, on regarde si le simbole existe, on le rajoute

sinon, et on rajoute un élément au tables de relocation des données data ou des instructions. Ces fonctions sont appelé dans la fonction principal qui est void rel(ListeG* Instruct, ListeG Data, ListeG* Etiquette, ListeG* RelocInst, ListeG* RelocData), et qui réalise la relocation.

Concernant la génération du code binaires, la fonction principale est void gen(ListeG Inst, dico_bin tab[], int tailledico, int tailletext, int tailledata, unsigned int texttab[]). Cette fonction fait appel à la fonction int genInstruction(inst_poly* bin,ListeG Inst, dico_bin tab[], int tailledico) qui génère le code binaire associé à l'instruction pointé par Inst, à l'aide d'un dictionnaire contenant le code func ou opcode associé à chaque instruction. Le code binaire ainsi créé est stocké dans l'union des structures pour qui contiennent le binaire en fonction du type R, I ou J de l'instruction. Cette fonction est passé dans une boucle parcourant la liste des instruction afin d'obtenir le code binaire de l'ensemble de la section text, qui sera stocké dans le tableau texttab, pour être utilisé par la bibliothèque elf. La génération du code binaire pour la section data se fait quand à elle grâce à la fonction void gendata(unsigned int* binairedata,ListeG Data, unsigned int datatab[]) qui prend en entré la liste des data et remplis le tableau datatab avec le code binaire. nous avons eu des difficulté pour passer le code binaire de la section data en big endian, et envisageons de le réaliser en écrivant le code binaire d'abord dans un fichier texte avant de remplir le tableau, pour n'avoir que des code de 32 bits pour les passer en big endian.

Enfin, pour la génération du fichier .o à l'aide de la bibliothèque elf, l'idée était de récupérer les tableau contenant le code binaire des sections data et text, pour faciliter le traitement du binaire et la génération du fichier .o avec les fonctions déjà implémentées. Nous somme cependant pas arrivé à terminer cette étape.

III. Organisation du travail

Dans un premier temps nous nous somme répartie le travail restant sur le livrable trois, pour le terminer au bout de la première semaine. Ainsi Giovanni traitais la vérification des opérandes des instructions tandis que Gilles s'occupait du remplissage de la table de relocation. Par la suite nous avons pris du temps pour découvrir le fonctionnement de la bibliothèque ELF afin de générer le fichier .o. Nous avons ensuite traiter le la génération du binaire, ainsi que le passage des donnée à la bibliothèque ELF. Ces deux dernières étapes n'ont cependant pas pus être achevés. Pour le déroulement du travail sur différentes partie du code, nous utilisons tous les deux le logiciel de gestion de versions de codes décentralisés Git, en travaillant chacun sur une branche séparées, et en fusionnant le code une fois validée.

La répartition des tâches s'est faites comme décrit dans le tableau suivant.

Tâches	Semaine 1	Semaine 2	Semaine 3
Traitement des opérandes des instructions	X	X	X
Remplissage et affichage des tables de relocation	X		
Analyse du fonctionnement de la bibliothèque ELF		X X	X
Génération du binaire pour les instructions et les data			X
Passage des données binaire à la bibliothèque ELF			X
Corrections apporté sur le code des livrables précédents	X	X	X
Réalisation du rapport			X X

Légende :

réalisé par : Giovani X Gilles X

Figure 1 : Répartition et organisation du travail

IV. Méthode de gestion des erreurs

Pour la gestion des erreur, le code est réalisé pour ne pas s'arrêter lorsqu'il y a une erreur dans l'analyse lexicale. Cela est fait de même dans l'analyse grammaticale. Lorsqu'une erreur est rencontrée, un warning est retourné, indiquant le numéro de la ligne ou il y a un problème, et indicateur d'erreur passe à un. Ensuite, la fin de cette ligne n'est pas traitée pour que cette erreur ne se répercute pas sur l'analyse des lignes suivantes. A la fin de l'analyse lexicale ou grammaticale, un test réalisé pour vérifier si l'indicateur d'erreur est à un. Si c'est le cas, alors on sort du programme avec un code d'erreur. Pour la relocation et la génération du binaire, dès qu'une erreur est rencontrée, le programme s'arrête en renvoyant à l'écran le message d'erreur. Cela est fait notamment pour ne pas générer du binaire d'un code qui ne correspondait à rien. En effet, si l'opérande d'une instruction n'existe pas, ou qu'une instruction n'existe pas, il est impossible de générer le binaire.

V. Tests réalisés

Pour tester le programme que nous avons écrit, nous avons utilisé quatre fichier de tests. Celui qui était proposé dans le sujet, à savoir miam_sujet.s, que nous avons modifié afin de traiter certains cas spécifique censé retourner des erreurs, ou devant être pris en compte lors de la compilation. Nous avons utilisé un deuxième fichier instruction_test.s, qui recense à peu près toute les instructions que nous devons prendre en compte dans ce projet, avec les différent type d'adressage. Enfin, nous avons fait un petit fichier test.s, ne contenant que quelques instructions, afin de comparer le code binaire obtenu finalement, à celui attendu.
Pour tester les nouvelles fonctions écrites à chaque fois, au lieu de les mettre dans uns fonction test, nous les implantations directement à la suite du code , en affichant à l'écran les entrées et les sortie de la fonction grâce à des printf(). Cela permettait de vérifier le bon fonctionnement de la fonction, et de gagner du temps sur le passage des opérandes.

VI. Problèmes rencontrés et travail restant à faire

Au cours de cette partie, nous avons rencontrés plusieurs difficultés. Tout d'abord, la fonction pour la vérification des opérandes des instructions nous a pris un temps considérable. En effet, la délimitation entre la première passe où les pseudo-instructions étaient remplacées, d'avec la deuxième passe où les opérande de type étiquette étaient remplacées par le décalage et les adresses nous pris du temps à être cerné.
Il y a eu un problème aussi pour la génération du binaire au niveau des pseudo-instructions. Pour les pseudo-instructions remplacées par deux instructions (pour Lw et Sw en particulier), le passage de la deuxième instruction en binaire à générer une erreur de segmentation lors de la lecture de l'instruction que nous ne somme pas parvenus à résoudre. De fait, le code binaire retourné pour ces instruction est incorrecte. Concernant la section data, la génération du binaire pour la réservation de mémoire des données non initialisée nous a aussi posé problème. Afin de résoudre cela, nous avons pensé remplir un fichier texte avec le code binaire généré. Ainsi nous pourrions mettre le nombre de zéros correspondant à la taille de l'opérande de la directive .space. De plus cela nous permettrait de réaliser le swap sur le code binaire des directives de tailles inférieur à trente deux bit, afin de passer le code en big endian. Mais la mise en oeuvre de cela n'a pas été fait par manque de temps.
Le travail ayant été réalisé de façon séparé, nous n'avons pas passé les tableau de binaire à la bibliothèque ELF par manque de temps. En effet, les fonctions pour

réaliser cela n'ont pas pu être toutes débugger et testés. La compréhension de la bibliothèque ELF nous ayant pris beaucoup de temps.
Par conséquent nous ne générons pas de fichier .o.

VII. Conclusion

En conclusion, ce projet nous a permis de cerner toutes les étape de la compilation d'un code assembleur, afin d'obtenir le fichier .o relogeable. Nous sommes parvenus à réaliser l'analyse lexicale, puis grammaticale, en prenant en compte la relocation, afin d'aboutir à la génération d'un code binaire.