LIVRABLE 4
Colin EPALLE, Eléonore BARGES

LIVRABLE 4

Pour ce dernier incrément nous avons dû dans un premier temps terminer certains point de l'incrément 3, sur lesquels nous reviendrons brièvement, avant d'aborder à proprement parler le dernier incrément.
Puis nous expliciterons les choix que nous avons fait lors du codage ainsi que les objectifs que nous avons atteints dans cette 4ème et dernière étape.
Enfin, nous terminerons sur les tests que nous avons effectués et la façon que nous avons eu de suivre l'évolution du projet bout par bout.

I. La fin de l'incrément 3

Pour parachever l'incrément 3, il nous restait plusieurs points à finir de coder et nous avons fait certains choix en termes de codage.
- Ce fut notamment le cas pour le stockage des arguments de type Base_offset. Nous avons finalement choisi de les stocker dans deux arguments : un argument de type Immédiat et un autre de type de Registre, ce qui nous a permis par la même occasion de vérifier l'existence du registre correspondant et la taille de l'Immédiat.
- Nous avons pu corriger certaines maladresses de codage des pseudo-instructions et assurer la gestion des sauts relatifs, ce qui avait été omis.
- Nous avons cru comprendre récemment que la table des symboles devait contenir la liste des symboles par ordre d'apparition des symboles et non par ordre de définition, ce que nous avions initialement codé. Nous avons contourné ce problème en réalisant une fonction, sort_symb_table, qui permet de renvoyer une table où les symboles sont rangés par ordre d'apparition des symboles à partir de la table des symboles que nous avions initialement crée (simplement en parcourant à nouveau la liste des lexèmes et en remplissant la nouvelle table des symboles à partir des informations de l'ancienne à mesure que les symboles sont trouvés). Cependant certaines personnes disent que c’est pas grave si c’est l’ordre de définition car c’est comme ça que ça était présenté. La variable reorder à la ligne 140 de main.c permet de choisir le type d’organisation de la table des symboles. Par défaut elle est à FALSE, donc par défaut la table des symboles est par ordre de définition.
Cette approche nous permet de conserver les deux manières de ranger les symboles.

II. Choix sur la façon d'aborder l'incrément 3

1. Le dictionnaire et la traduction en binaire

Le premier objectif que nous nous sommes fixé était celui de stocker dans le dictionnaire sous une forme facilement interprétable les informations utiles à la traduction en binaire, laquelle est spécifique pour chaque instruction. Nous avons souhaité respecter la contrainte qui voulait que sur chaque ligne du dictionnaire se trouvent le même nombre de termes, afin de conserver notre lecture du dictionnaire simultanée au stockage des informations dans une liste_dictionnaire (cf: la ligne de code suivante :
while (fscanf (fp, "%s %d %s %s %s %d %s %s %s %s %s", inst, &nb_arg, arg1, arg2, arg3, &opcode, c_b_1, c_b_2, c_b_3, c_b_4, c_b_5) != EOF)).
Ainsi, nous avons finalement choisi de stocker les informations sous la forme suivante :
INSTRUCTION nb_arg Type1 Type2 Type3 opcode bits(21->25) bits(16->20) bits(11->15) bits(6->10) bits(0->5)
Dans une telle structure, chaque champs qui ne doit être rempli par rien est remplacé par NONE pour les Type et par _ pour les champs de bits. Les champs

"opcode" et "bits(a->b)" prennent tantôt une valeur en décimal correspondant à la valeur en binaire équivalente, tantôt les valeurs "a1", "a2" ou "a3", ce qui se réfère à l'argument respectivement 1, 2 ou 3.
Ces informations sur le codage en binaire sont stockées dans une liste dictionnaire afin de pouvoir être récupérées dans la fonction instr_in_binar par la fonction look_for_instr_and_return_binar_info, juste au moment de la traduction en binaire des instructions et de leurs arguments.

Concernant le codage en binaire, nous avons choisi d'utiliser des masques et des décalages, que nous avons trouvés plus faciles à manipuler, lesquels sont utilisés dans les fonctions instr_in_binar et data_in_binar.

2. La réalisation du fichier elf, étapes de la réalisation

Après étude approfondie du fichier initial make_mips_elf, nous avons abordé la réalisation de notre propre fonction de la sorte :
- Nous avons réalisé un tableau de int contenant chacun sur 32 bits la valeur en binaire de l'instruction avec ses arguments.
- Nous avons fait de même pour réaliser un tableau de int contenant les directives, en veillant à mettre chaque .word sur un seul int, et en mettant bout à bout les .byte, les .space et les .asciiz.
- Puis nous avons réalisé la sym_table, un tableau de char* contenant tous les symboles du fichier .s, réalisée à l'aide de la table des symboles, et avons mis à jour la valeur de bss_prog à l'aide de la valeur cumulée de toute les valeurs de .space contenues dans la section .bss.
- Enfin, nous avons calculé les valeurs de size_table et size_instr, qui contiennent le nombre de symboles et instructions pour chacune de ces listes, afin de savoir quelle place réserver pour les tableaux. Nous fîmes la même chose pour déterminer la taille du tableau de data, tout en prenant gare à bien considérer le décalage correspondant à chaque directive.
- Enfin, nous avons réalisé la fonction swamp, laquelle nous permet de passer de little endian à big endian.
- Nous avons testé le code préécrit du main (après avoir renommé la fonction main_init_function) en remplaçant uniquement les champs de data_prog et text_prog. La vérification du bon fonctionnement de ces tests s'est faite à la main et à l'oeil, comparant la version initale, et notre version complétée peu à peu, exécutée pour exemple.s.
- Une fois que nous avons été certains que nos tableaux et notre traduction en binaire et notre swap fonctionnait correctement pour le fichier exemple.s, nous avons réaliser les fonctions make_elf32_reloc et make_syms lequelles permettent de créer les sections Elf32_Rel et Elf32_Sym de façon systématique, quelque soit le fichier .s analysé.

Après ces premiers tests sur l'unique fichier exemple.s, nous avons lancé d'autres tests plus poussés sur d'autres fichiers .s. Nous avons fait une comparaison à l'oeil, grâce à l'outil mips-as et objdump, qui nous permettait d'obtenir un fichier de référence avec lequel nous comparions nos propres fichiers.

Nous nous sommes appliqués à la libération toute la mémoire allouée. Pour cela nous avons utilisé l’outil valgrind (avec l’option –leak-check=full pour faire ça proprement). Cette étape a permis de repérer des erreurs dans le code.

Ainsi, notre programme est actuellement capable de traduire un fichier .s en un fichier binaire executable. Il est, semble-t-il, assez robuste aux erreurs.

III. Tests effectués

Dans notre fichier, plusieurs fichiers de tests sont disponibles. Ils nous ont permis, étape par étape de mettre à l'épreuve la robustesse de notre code.
Ainsi, le test symb_table_ordre nous a permis de vérifier le bon fonctionnement de notre code avec er sans l’usage de la fonction sort_symb_table, et la capacité de notre code à gérer le cas d'étiquettes ayant le même nom que des instructions.
Le test mega_test.s nous a permis de vérifier la bonne détection de différents types d'erreurs concernant les instructions (taille des arguments, type des arguments,...) et les directives : il fut utilisé à la main, pour des raisons techniques d’erreurs soulevées par l’assembleur accessible sur les ordinateurs de l’école, lesquelles ne correspondent pas aux consignes qui nous ont été données.
A l’aide du fichier de test SimpleUnitTest nous avons pu constater que les tests suivants fonctionnent : donnees_simple.s et exemple.s . Le fichier miam_sujet.s semble fonctionner également, au détail près que l’assembleur de l’ordinateur, dans ce cas précis, ne stocke pas les symboles par ordre d’apparition et qu’il ne complète pas par des zéros afin de tomber sur un multiple de 32 bits le tableau en binaire des .data : ainsi le test renvoie une erreur.