Bollet Jacqueline
Grenier Thierry

Mai 2000

Projet Logiciel de Base

Encadré par Matthieu Martel et Léon Mugwaneza

SOMMAIRE

INTRODUCTION

I – Les choix effectués

II – Répartition du travail

III – Notre travail

a) assembleur

b) Editeur de liens

c) Programme de tri avec listes doublement chaînées

d) Les problèmes … état final (tests).

CONCLUSION

ANNEXES :

LISTE DES INSTRUCTIONS GEREES

TYPE D'INSTRUCTIONS

FORMAT DU FICHIER OBJET

INTRODUCTION

Le but de ce projet était de faire programme d'assembleur – éditeur de liens qui génère un exécutable pour machines de type MIPS et de le tester sur un programme de tri sur liste doublement chaînée circulaire. Ce rapport à pour but d'expliquer le fonctionnement de notre programme ainsi que sa conception.

Nous commencerons par présenter les choix que nous avons pris pour nos conventions et l'organisation générale du programme. Nous exposerons ensuite la répartition du travail. Nous continuerons en rentrant un peu plus dans le détail du programme en expliquant son fonctionnement général et en présentant un algorithme simplifié des principales fonctions. Nous finirons par le listing complet du programme (les tests effectués étant présentés en annexe).

I – Les choix effectués

1) Assembleur en C

L'assembleur ressemble beaucoup à celui indiqué dans le cours. Il effectue deux passes. Dans la première passe, il vérifie la syntaxe, construit la Table des Symboles (TS) ainsi que la section donnée du fichier objet. La seconde passe se charge de construire un fichier temporaire (qui contient la section code du fichier objet à générer) et calcule les valeurs dont a besoin l'entête (voir fichier objet pour plus de détails). Enfin, le fichier objet est obtenu en écrivant l'entête puis la TS et en concaténant le résultat au fichier temporaire créé par la seconde passe. Les fichiers temporaires sont alors effacés.

On peut noter que la première passe se charge aussi de transformer les pseudo instructions en véritables instructions élémentaires (voir Instructions gérées pour plus de détails). La seconde passe transforme toutes les instructions possibles directement en code machine mais ne traduit pas entièrement les instructions ayant une adresse relogeable.

Il est aussi à noter que l'on suppose que le fichier source ne contient aucune macro et ne fait référence à aucune bibliothèque.

Chaque ligne lue depuis le source est stockée dans un tableau de taille 50 caractères (8 maximum pour l'instruction et chaque paramètre plus un espace et des virgules). La fonction de lecture se charge donc de mettre chaque ligne sous la même forme pour faciliter le travail de notre analyseur syntaxique. Les commentaires ne seront bien évidemment pas pris en compte. Il est à noter que certaines règles sont à respecter pour le fichier source :

- Les étiquettes doivent contenir au plus 8 lettres et ne peuvent pas contenir de chiffre.
- Les deux points définissant une étiquette doivent être collés à celle-ci.
- Les immédiats doivent être entrés sous forme décimale. La plage où ils se situent est limitée entre –32768 et +32767. Cette limite a été insérée pour faciliter la prise en compte des immédiat mais il est facile de l'augmenter en insérant au besoin la gestion d'une instruction lui au besoin.

Son appel est du type « assembleur <fic.s> [fic.o] ». Le fichier <fic.s> sera supposé dans le répertoire courant de l'appel et le fichier généré est écrit dans ce même répertoire. L'option [fic.o] n'est pas obligatoire ; si elle n'est pas précisée, le nom du fichier généré est le même que celui du source, seule l'extension changeant. Il est à noter que les noms des fichiers sources et objets ont les même restrictions que les étiquettes : 8 lettres maximum (plus l'extension .s et .o obligatoire).

2) Editeur de liens

Il ressemble beaucoup à celui présenté en cours. Il effectue deux passes. Dans la première, il créé la Table Globale des Références Externes (TGRE) en s'appuyant sur tous les symboles externes et globaux des différentes TS. La seconde passe s'occupe de résoudre les problèmes de relocation, et de construire l'entête qui sera, comme dans l'assembleur, concaténée à un fichier temporaire qui sera effacé.

Son appel est du type «./Edl [-O nom] [-E"label"] [-C@] [-D@] [fic1.o fic2.o …] <ficn.o>». Les options entre [] ne sont pas obligatoires.

- L'option –C permet de choisir l'adresse d'écriture du code. Par défaut, elle sera égale à 0x0400000.
- L'option –D permet de choisir l'adresse d'écriture des données. Par défaut, elle sera égale à 0x10000000.
- Le (ou les) fichier <fic1.o> (ainsi que les autres) devra être dans le répertoire courant.
- L'exécutable sera créé sous le nom <nom> dans le répertoire courant. Par défaut il s'appellera mon_exec.
- Enfin, l'option –E permet de choisir un label qui définit le point d'entrée. Par défaut, l'éditeur cherche un label "main" et le prend comme point d'entrée.

Il est à noter que le point d'entrée doit être déclaré en global dans l'un des modules pour que l'éditeur de liens le trouve.

3) Fichier objet

Au début on pensait « mal » notre fichier .o parce que nous n'avions pas les idées claires. Ce n'est qu'ensuite que nous avons pu le réaliser (tout mettre en binaire).

Il nous a semblé impératif de commencer par définir le format exact du fichier objet puisqu'il est le but de l'assembleur et le point de départ de l'éditeur de liens. Il sera présenté ici comme une suite d'information. L'ordre de cette suite est bien entendu très important et ne sera pas changé.

Entête :

- Une suite de caractère donnant le nom du fichier. Ceci permet de bien vérifier que l'on traite le bon fichier. Ce nom ne doit pas dépasser 8 lettres soit 11 lettres (8 lettres de nom, 2 lettres pour l'extension « .o » et une de plus pour le signe de fin de chaîne « \0 ».Ce qui explique les 11 caractères déclarés.
- Un entier représentant la taille totale de la TS (nombre d'éléments)
- Un entier représentant la taille totale du code (taille en octets).
- Un entier représentant la taille totale des données (taille en octets).

Table des Symboles :

Pour chaque symbole

- Un nom de symbole de taille maximale 8. (On définit alors 9 caractères pour comprendre le « \0 »).
- Un caractère qui définit les champs utiles : 4 bits à 0 d'abord, puis 2 bits pour définir la portée du symbole(10 : local, 01 :global et 00 :externe), ensuite 1 bit pour définir la section (1 :section texte, 2 :section données),et enfin 1 bit pour définir l'état du symbole (0 : non défini et 1:défini).
- Un entier long non signé pour indique l'adresse (relative dans le cas de l'assembleur et absolue dans le cas de la 2ème passe de l'édition des liens.

Section Code :

Pour chaque instruction :

- ❖ Un caractère définissant différents champs : 1bit à d'abord puis 3 bits indiquant le type (tel que : type R =000 ; type I1=001 ;type I2 =010 ;type I3=011 ;type J=100) ,ensuite 2bits d'info de relocation (tel que : définie =00; relogeable/code =01; relogeable/donnée=10; en référence externe =11) ,enfin 2 bits d'info supplémentaires pour indiquer si la relocation est en partie haute (10) ou en partie basse (01).
- ❖ Un entier long non signé donnant tout ce que l'on a pas pu traduire de l'instruction.
- Si l'instruction fait appel a une référence externe, à la place de la valeur de l'adresse, on a le numéro de la référence externe dans le champ ou elle devrait être placée.
- Pour cela on s'est limité à 2^{17} –1 symboles (on a une place de 16 bits pour les instructions de type I et 26 bits pour les instructions de type J : on est donc limité par la borne inférieure : 16 bits)

Section Donnée :

Pour la section données on a utilisé des suites de caractères. On a supposé que pour l'éditeur de liens, les données étaient toutes relogeables par rapport à l'adresse d'implantation de la section data du module concerné. On a interdit ainsi de déclarer des données avec une adresse absolue.
Aussi la section data ne comporte aucune information pour le fichier objet .
L'éditeur de liens se contente de recopier la section à la suite des autres.
Il ne se soucie pas de son contenu, l'assembleur ayant fait le nécessaire pour que les données soient alignées en fin de module (pour le début du prochain module).

Instructions choisies

Nous avons choisi d'implémenter toutes les instructions élémentaires (sauf celles concernant les doubles) ainsi que les pseudo instructions suivantes : la, move, beqz, bge et li. Les directives .at et .noat ne seront pas autorisées. Pour une liste exhaustive des instructions autorisées, voir l'annexe.

II – Répartition du travail

Pour la répartition des tâches à effectuer, nous sommes restés proche de ce que proposait l'énoncé.

Thierry Grenier a fait l'assembleur, ainsi que le module 3 (opérations sur les listes) de partie programmation en assembleur.

Jacqueline Bollet s'est occupée de l'éditeur de liens, ainsi que des trois modules restant sur les listes.

Ceci étant dit, certaines parties ont été effectuées en commun : la définition du fichier objet, la recherche d'idées pour la représentation des instructions (analyse des instructions). Le programme C sur les listes doublement chaînées a aussi été faite en commun pour que chacun ait bien dans l'esprit ce que l'autre va faire.

III – Notre travail

On remarquera que même si les structures des données détaillées pour chaque partie portent des noms différents, elles n'en ont pas moins les mêmes formats. En effet, comme précisé plus haut lorsque nous avons parlé de notre fichier objet, nous avons harmonisé nos formats de données alors.
Pour la correspondance des formats, veuillez voir le format du fichier objet déclaré plus haut.

a) Assembleur

1) Présentation générale

Il se compose sous la forme de 5 modules qui donnent lieu à deux exécutables. Le premier « Instructions » est généré à partir des modules « instruction » et « main ». Le second est généré à partir des modules « assembleur », « analyseur », « passe1A », « passe2A » mais aussi « main ».

« Instructions » sert a généré des fichiers qui contiennent les instructions MIPS classées selon leur type. Nous avons défini les types comme expliqué en annexe. Cette solution a été choisie car, au départ l'assembleur et l'éditeur de liens devaient s'en servir. Le deuxième avantage de cette solution est que l'on peut ajouter autant d'instructions que l'on veut sans avoir à tout recompiler (par exemple, on pourrait facilement ajouter les instructions sur les flottants).
Note: Le nombre de pseudo-instructions traitées étant petit, nous avons choisi de ne pas les représenter dans ce module.

« Assembleur » est le programme d'assemblage proprement dit. Il comporte un analyseur syntaxique (« analyseur ») qui vérifie les lignes du source et les met en forme pour la première passe. S'il reconnaît entièrement une instruction, il l'écrit directement dans un fichier temp (en fait, il écrit où se trouve l'instruction dans les tableaux, ainsi que les valeurs des arguments). Sinon, la première passe (« passe1A ») reprend la main pour finir le travail. La deuxième passe (« passe2A »)part du fichier temp pour écrire le fichier temp2 qui a alors directement le format voulu. L'assembleur (« assembleur ») reprend alors la main pour écrire l'entête dans le fichier objet et d'y concaténer a la suite le fichier temp2 puis le fichier data. Il efface ensuite les fichiers temp, temp2 et data.

Tous ces modules ont besoin du module « main » qui définit toutes les structures utilisées.

2) Structures utilisées

La table des symboles (TS) est représentée par une liste simplement chaînée (Liste). Elle comprend un champ elt de type ENREG et un champ suivant qui est un pointeur sur liste.

Le type ENREG définit le Symbole (voir format du fichier objet)

L'entête déclaré sous une structure S_entete.

Chaque instruction est écrite par une structure Instruction (voir format du fichier objet).

Enfin les dernières structures utilisées sont expliquées dans l'annexe sur les types d'instructions.

Les variables globales sont :

LGMAXLIGNE qui définit la longueur maximale d'une ligne.
ligne (tableau de char) qui est la ligne courante.
compteur (entier) qui contient l'adresse relative par rapport au début de la section courante.
entete (type S_entete) qui contient l'entête
TS (pointeur sur Liste)
instr (type Instruction) qui contient l'intruction courante
R (tableau de type R1) : instructions de type R
I (tableau de type I1) : instructions de type I
i (tableau de type I1) : instructions de type i
p (tableau de type p1) : instructions de type p
J (tableau de type p1) : instructions de type J

3) Algorithme : quelques fonctions détaillées.

i) « instruction »

En fait ce module est le module « Ecrire » repris du projet d'algorithme. Après avoir défini les variables et initialisé les tableaux, il ouvre simplement les fichiers, écrit les structures et referme les fichiers.

ii) « assembleur »

C'est ce module qui contient la fonction main.

❖ main :
- entrée : les arguments de la ligne de commande (argc et argv[])
- retour : rien

Elle appelle verifie_arguments (qui vérifie le nombre d'arguments et le nom des fichiers en argument) en premier lieu.
Chaque opération ultérieure est conditionnée par le fait qu'aucune erreur ne se soit produite entre temps (sauf les ouvertures et fermetures de certains fichiers).
Après avoir ouvert tous les fichiers nécessaires, elle appelle charger_instructions (en fait remplit les tableaux i, I, R, J et p avec les fichiers correspondant).
Elle appelle premiere_passe puis seconde_passe et enfin ecrire_final_final.
Elle ferme enfin les derniers fichiers encore ouverts.

❖ Ecrire_fichier_final :
- entrée : les arguments de la ligne de commande (argc et argv[])
- retour : rien

Elle appelle purger_TS (qui enlève de la TS tous les symboles internes non globaux = tous les symboles dont le $5^{ème}$ bit du champ « donnee » est à 0 : cf. fichier objet) pour calculer la taille de la TS. Elle écrit l'entête, puis la TS et concatène le résultat aux fichiers temp2 et data. Elle efface enfin les fichiers temporaires (data, temp et temp2).

On remarquera que la fonction purger_TS est directement copiée du programme sur les polynômes fait précédemment dans l'année.

iii) « passe1A »

- ❖ première_passe :
  - entrée : rien
  - retour : entier (0 = erreur ; 1 = pas d'erreur)

Elle a été copiée sur le cours (elle appelle lire_prochaine_ligne et traiter_ligne en boucle). Elle s'occupe de plus de donner les bonnes valeurs à différentes variables.

- ❖ traiter_ligne :
  - entrée : section (pointeur sur entier)
  - retour : entier (0 = erreur ; 1 = pas d'erreur)

Appelle Analyse_instruction, puis, selon le résultat renvoie une erreur, déclare les sections text ou data, appelle remplir_TS (si globl, extern ou déclaration d'une étiquette), reserver_place (si directive différente de globl, extern, data, text), ou renvoie simplement 1.

- ❖ reserver_place :
  - entrée : rien
  - retour : entier (0 = erreur ; 1 = pas d'erreur)

Agit différemment selon la directive à traiter :

Align : insère des caractères nuls dans le fichier data jusqu'à ce que compteur (adresse relative par rapport au début de la section data) soit un multiple de deux à la puissance de l'argument.

Ascii ou asciiz : vérifie si on a bien une expression entre guillemet et l'écrit dans le fichiers data. Transforme les caractères \n et \t.

Byte, half ou word : écrit les données (si elles sont valides) dans le fichier data sous la forme d'un char pour byte, short int pour half et int pour word.

Space : insère autant d'espaces dans le fichier data que voulu.

- ❖ Remplir_TS :
  - entrée : type (caractère), section (entier) et cur (entier)
  - retour : entier (0 = erreur ; 1 = pas d'erreur)

Les types sont : E = déclaration d'une étiquette, X = directive extern, G = directive globl, I = étiquette vue dans une instruction.
Elle recherche le label dans la TS et modifie ses informations si besoin selon le type :
Si le type est E, elle vérifie qu'il s'agit bien de la première définition et que la variable n'a pas été déclarée en extern. Dans tous les cas, elle force deux bits du champ donnee à 1 (interne et défini) et le bit de section a la valeur de section.
Si le type est X ou G, elle vérifie qu'aucune utilisation préalable n'a été faite (donnee = 0) et remplit le champ donnee.
Si le type est I, elle ne remplit le champ donnee que s'il est nul.

iv) « passe2A »

- ❖ seconde_passe :
  - entrée : rien
  - retour : entier (0 = erreur ; 1 = pas d'erreur)

Elle s'occupe de lire chaque ligne du fichier temp (différemment selon la valeur de l'info principalement). Appelle les fonctions traduire_R, traduire_I ou traduire_J selon le cas (qui ne font que calculer la valeur totale de l'instruction). C'est aussi elle qui remplit l'information de relocation : dans le champ info mais aussi dans l'instruction même dans le cas d'une référence externe. Dans ce dernier cas, elle appelle num pour connaître la place qu'aura le symbole dans la TS sans les internes.

v) « analyseur »

- lire_prochaine_ligne :
  - entrée : rien
  - retour : rien

Elle a été fortement inspirée par le programme de justification de texte rendu antérieurement.
Elle passe les premiers espaces (ou les premières tabulations), puis remplit la tableau ligne jusqu'à un espace (ou une fin de ligne, ou commentaire ou un ":"). Si ce n'est pas un espace, elle retourne directement. Sinon, elle continue en effaçant les eventuels espaces ou tabulations (sauf si un caractère début de chaine a été détecté). Elle retourne en fin de ligne.
Entre chaque étape, si un # a été détecté, la ligne est lue jusqu'au bout mais le tableau n'est alors plus complété.
Il en résulte dans tous les cas : soit une ligne commançant par # (commentaire), soit une étiquette, soit une instruction dans le format "instr arg1,arg2,arg3".

- Analyse_instruction :
  - entrée : rien
  - retour : caractère (F = False, D = Directive, E = Etiquette, R, I, i, p, P ou J)

C'est cette fonction qui écrit dans le fichier temp dans le cas d'une instruction reconnue. Elle traduit aussi les pseudo-mode d'adressage en plusieurs instructions. Enfin (et surtout) c'est elle qui vérifie la syntaxe de la ligne.
Son algorithme est simple :

Si on a une étiquette, elle renvoie immédiatement E.

Si on a une directive, elle en vérifie le nom et si un argument est nécessaire et rend D si ok.

Pour chaque type d'instruction, elle vérifie le nombre d'argument, puis appelle les fonctions label, registre et immediat pour vérifier si ce sont les arguments attendus, enfin écrit dans le répertoire temp l'info (contient le type d'instruction et si les labels sont à prendre en entier, ou juste une partie de leur adresse), un numéro (place de l'instruction dans le tableau concerné), les arguments et enfin un caractère fin de ligne. Dans le cas d'un pseudo mode d'adressage, les instructions sont écrites les unes après les autres.

Enfin, dans le cas d'une pseudo instruction, elle appelle pseudo (comme on n'en traite que peu elles sont traitées au cas par cas et traduites directement).

- registre :
  - entrée : cur (pointeur sur entier) : position sur la ligne
  - retour : entier (numéro du registre ou 32 si inconnu)

Cette fonction, bien que d'apparence longue, ne fait que transformer les noms des registres en numéros et retourner 32 si le registre n'a pas été reconnu ou s'il s'agit du registre 1 dont l'utilisation est interdite. Lorsqu'elle retourne 32, elle positionne cur à son état initial.

b) Editeur de liens

1) Présentation générale et fonctionnement

L'éditeur de liens se compose de 8 modules : global, utils, mains, passe1, passe2, erreurs, erreur1, erreur2. L'exécutable généré se nomme Edl.

On remarquera dans le Makefile, 2 autres exécutables (composés chacun d'un seul module : objet, et ouvrir) qui sont générés lors du Make all : Ecrire et Ouvrir respectivement. Ce sont 2 exécutables qui m'ont permis de tester mon éditeur de liens. Je n'en parlerai pas dans le détail.

i) L'éditeur de Liens

L'éditeur prend sur la ligne de commande les noms de fichiers à lier. Ils doivent être de type « .o » et avoir un nom de 8 lettres au maximum.
Lors d'une édition des liens, les arguments sont traités (options et fichiers objets). L'éditeur de liens enregistre les différents noms de modules (dans une Table des Modules) et commence ensuite la 1ère passe.

La 1ère passe ouvre les différents fichiers objets. Elle vérifie et traite les entêtes et s'occupe de créer une Table des Symboles Globales (TGS) avec la Table des Symboles (TS) de chaque module. Elle résout les adresses des étiquettes (passage de la valeur relative en valeur absolue par rapport à l'adresse d'implantation du module).

Remarque : On suppose que les symboles des TS des fichiers à lier ne doivent comporter que des éléments qui sont définis. (On ne tient compte donc que de la portée globale ou externe).
Le champ état n'est pas tenu en compte(le dernier champ dans l'octet « Olpse »).

A l'issue de cette 1ère passe, la main est retournée à l'éditeur de liens qui appelle ensuite la 2ème passe.
La 2ème passe s'occupe de traiter les différentes sections code et données de chaque module. Elle reloge les adresses dans le cas de la section code. Elle génère alors un fichier temporaire appelé « fcode.temp » pour le code et un fichier « fdata.temp » pour les données.
Dans ces fichiers sont concaténés les différentes section code et données des modules.
La 2ème passe va ensuite généré un fichier avec le nom d'exécutable choisi avec l'entête correspondant.
Elle concatène ce fichier avec les fichiers temporaires code et données.
La main est retournée à l'éditeur de liens qui propose d'effacer les fichiers temporaires si l'utilisateur le veut.

ii) Ecrire

Cet exécutable, composé d'un unique module : objet, m'a permis de générer des fichiers objets au bon format pour tester le bon fonctionnement de l'éditeur de liens.
Il demande d'abord à définir l'entête, ensuite la TS puis la section code et enfin la section données.

C'est un outil assez rudimentaire car il faut tout de même tout calculer les valeurs du Code soit même : champ d'information et valeur de ce qui a été traduit. Et cela sans se tromper dans la saisie : mon exécutable ne permet pas de se tromper. (pas de retour arrière possible).

Pour cela je me suis aidé de lbxspim : en effet, on me donnait alors la valeur de ce qui été traduit et les références relatives aux labels locaux (relogeable code ou données selon le label) ainsi qu'au références externes (je n'avais plus qu'a ajouter alors le numéro d'entrée dans la Table des Symboles que j'avais définie).

Comme l'éditeur de liens permet le choix d'une adresse d'implantation code et données, j'ai choisi les mêmes que lbxspim fournissait pour les tests (je n'avais alors pas à recalculer les adresses relatives).

iii) Ouvrir

Du précédent exécutable, j'ai associé cet allié. Cet exécutable permet de lire un fichier objet : il affiche le contenu en le décomposant.

Il permet de voir la TS, la section code avec les instructions une à une (définition du type et des relocations alors) et la section data (affichage octet par octet).

2) Structures de données, représentation.

i) les structures

Outre les formats de structures nécessaires au fichier objet (déclarés en partie dans les fichiers global.h, passe1.h et passe2.h et dont se sert les exécutables Ecrire et Ouvrir), j'ai déclaré dans le module global mes différentes structures personnelles (encore global.h) et mes variables globales (dans global.c).

Dans un souci d'uniformiser mes données ensembles, j'ai choisi de définir un type de structure assez important (certains pourront dire assez lourd) qui regroupe la Table des Modules et la Table Globale des Symboles : **la Table Globale des Références Externes**.
Ces 2 tables sont elles-mêmes 2 structures que je déclare.

La Table des Modules (Table_Module) :
Elle est composée d'éléments de type Module.
Un Module est défini tel que :

- Son nom (11 lettres max.)
- La taille de sa section code
- La taille de sa section donnée
- L'adresse d'implantation de sa section code
- L'adresse d'implantation de sa section donnée.
- La taille de sa TS
- Une Table de référence.

La Table de référence est de la taille de la TS. C'est un tableau d'entiers. Ces entiers sont le numéro par lequel le symbole $X^{ième}$ de la TS du module se trouve dans la Table Globale des Symboles.
Ex : Table[X] = n.
TGS[n] = > on retrouve le label qui a été défini dans la TS du module.(il faut qu'il soit global à la $2^{ème}$ passe).
Cette table n'est pas déclarée en type car elle n'est utilisé qu'une fois.
En revanche le type Module est bien défini (puisque cette table en est composée).

La Table Globale des Symboles (TGS):
Elle est de type TS. J'ai définit donc le type TS, qui est une table d'éléments de type Symbole dont je rappelle la définition en minimum (voir spécification dans le fichier objet).
Symbole :

- Son nom
- Son champ de Spécification (Olpse)
- Son Adresse

J'ai définit le type Symbole dans global.h tel que utilisé pour le fichier objet.
On peut donc définir :

La Table Globale des Références Externes (Table_Globale_des_Références_Externes) :
Elle est donc composé des 2 précédentes tables.

ii) les variables globales

J'ai défini ensuite en tant que variables globales :

Une Table_Globale_des_Références_Externes : **TGRE**

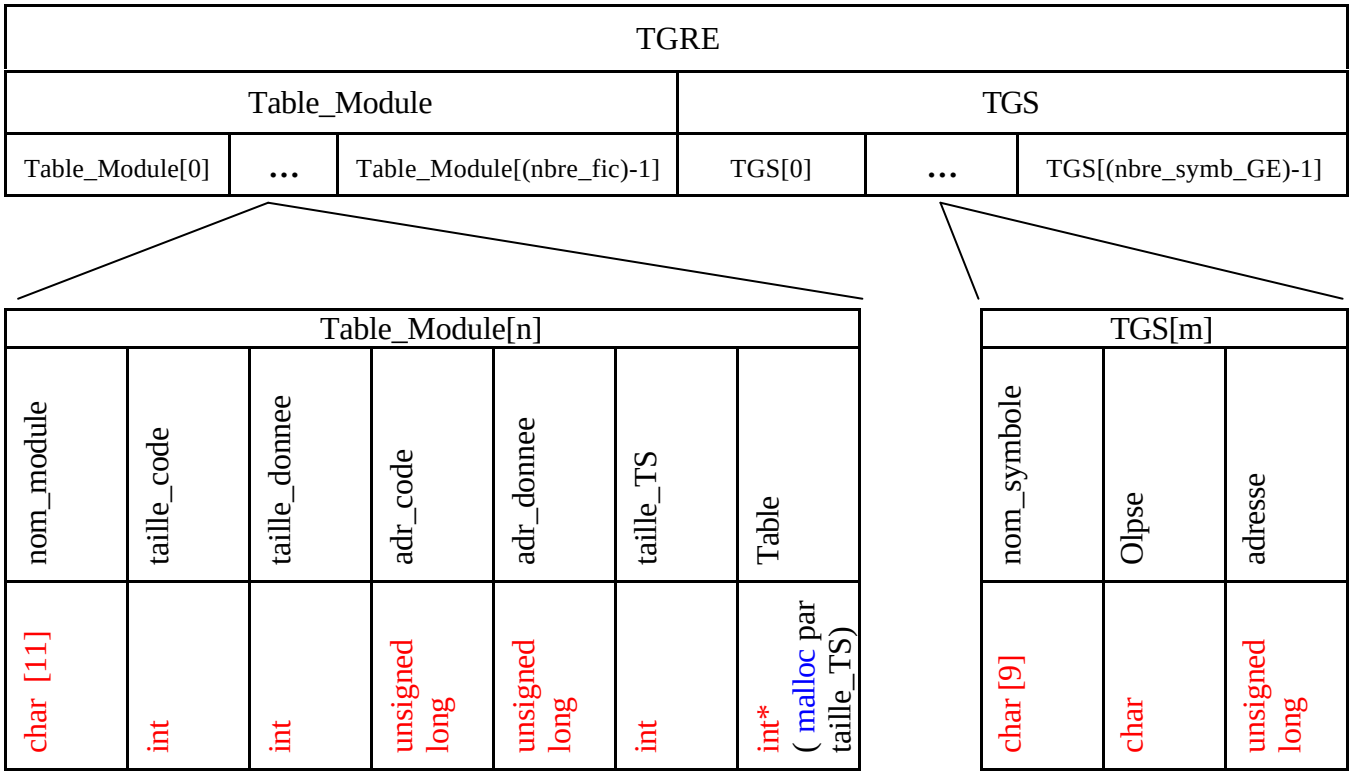
Une chaîne de caractères définissant le point d'entrée : **point_entree**
Une chaîne de caractères définissant le nom de l'exécutable : **nom_exec**

Un entier long non signé définissant l'adresse initiale du code : **adr_init_code**
Un entier long non signé définissant l'adresse initiale des données : **adr_init_data**
Un entier long non signé définissant la taille totale du code : **T_taille_code**
Un entier long non signé définissant la taille totale des données : **T_taille_donnee**

Un entier donnant le nombre de modules traités (taille de la Table des Modules) : **nbre_fic**
Un entier donnant le nombre de symboles Globaux/Externes dans la TGS : **nbre_symb_GE**

Un tableau d'entier long qui sera de la taille du nombre de fichier : **pfic**
Ce tableau est un « repérage » pour les fichiers de l'endroit ou commence la section code (c'est à dire où finit la TS).Il est rempli avec la fonction ftell (qui donne la place du pointeur d'un fichier ouvert) à la 1ère passe, juste après avoir lu la TS du module. Lors de la 2ème passe, on se sert de ce tableau pour placer le pointeur fichier à la fin de la TS du module (où début la section code).
En effet, les fichiers à traiter sont fermés entre les 2 passes.

La représentation de la TGRE serait :



Table_Module est de type Module* (malloc par nbre_fic).

TGS est de type TS (c.a.d. Symbole *). L'allocation se fait par malloc d'un symbole au début puis par realloc à chaque nouveau symbole à entrer (nbre_symb_GE est alors incrémenté)

3) Spécification des modules et algorithmes.

i) en général

On peut donc distinguer les modules suivants :

- main : c'est par là que ça commence ! C'est ici qu'est placée la fonction main.
  C'est là que l'on traite les arguments de la ligne (fichiers à lier et options choisies).
  On appelle la passe1 puis la passe2.
  Le fichier erreurs lui est associé.

- erreurs : ce module ne contient qu'une fonction : erreur qui prend en entrée 2 entiers arg et info.
  On diagnostique ainsi le type d'erreur qui a interrompu le programme. Elle sort de l'exécution.

- global : ce module déclare les structures utiles à tout le programme (notamment le format de la TGRE).Il déclare aussi toute les variables globales.

- utils : ce module contient toutes les fonctions d'affichage de la TGRE et des TS. Elles sont appelées au cours de l'exécution de la 1ère et de la 2ème passe.

- passe1 : ce module réalise la 1ère passe de l'éditeur de liens. Lors du traitement d'un fichier, il lit l'entête, détermine l'adresse d'implantation (code et donnée) et incrémente la taille totale du code et celle de la donnée. Puis il traite les symboles de la TS, les insère ou les mets à jour selon le cas dans la TGS. Il crée alors la table de référencement (c.a.d. Table_Module[n].Table) selon le placement du symbole X^{ième} de la TS, dans la TGS (par ex : Y^{ième} place dans la TGS), on aura :

  Table_Module[n]. Table[X] = Y et TGS[Y] = Symbole X^{ième} que l'on avait dans la TS
  (avec plus ou moins les même spécifications: le champ Olpse peut avoir été mis à jour entre temps avec un autre fichier)

  Il vérifie qu'à la fin la TGS, n'est constituée que de référence globale (et sous entendue définie).
  On demande s'il reste des références externes, si l'utilisateur veut ignorer ou sortir du « linkage ».
  (Notamment pour voir les autres erreurs, car une exécution du programme généré est compromise)

- erreur1 : ce module ne contient aussi qu'une seule fonction tout comme erreurs : une fonction appelée erratum et qui prend en entrée 2 entiers : arg et info. Elle diagnostique les erreurs générées par la passe1. Elle sort de l'exécution.

- passe2 : ce module réalise la 2ème passe de l'éditeur de liens. Il traite les section code et donnée (surtout code vu que la donnée n'est que de la copie).Pour le code, il réalise les différentes réalocations à gérer (réalocation code/donnée, référence externe) sur les différents types d'instructions (notamment type I => sur 16 bits et J=> sur 26 bits). Il écrit les différentes section codes dans le fichier fcode.temp et les sections données fdata.temp. Ensuite l'exécutable est crée avec un entête correspondant. Enfin ce dernier fichier est concaténé avec le fichier fcode.temp puis fdata.temp. On a donc notre exécutable généré dès lors.

- erreur2 : ce module ne contient aussi qu'une seule fonction encore: une fonction appelée errare et qui prend en entrée 2 entiers : arg et info. Elle diagnostique les erreurs générées par la passe2. Elle sort de l'exécution.

ii) Quelques fonctions des modules principaux.

Module main.c est composé outre de la fonction main, des fonction suivantes :

- traiter_point_entrée :
  - entrée : une chaîne de caractères.
  - retour : rien

Cette fonction traite la redéfinition d'un point d'entrée sur la ligne de commande (option –E).

- traiter_adr_init_code :
  - entrée : une chaîne de caractères.
  - retour : rien

Cette fonction traite la redéfinition d'une adresse initiale code sur la ligne de commande(option –C).

- traiter_adr_init_donnee :
  - entrée : une chaîne de caractères.
  - retour : rien

Cette fonction traite la redéfinition d'une adresse initiale donnée sur la ligne de commande(option –D).

- traiter_nom_exec :
  - entrée : une chaîne de caractères.
  - retour : rien

Cette fonction traite la redéfinition d'un nom d'exécutable sur la ligne de commande (option –O).

- test_ligne_arg :
  - entrée : les arguments de la ligne de commande (argc et argv[])
  - retour : entier résultat du test

Cette fonction teste si la ligne d'arguments contient au moins en dernier argument un fichier objet.

- test_fic_objet :
  - entrée : une chaîne de caractères.
  - retour : entier résultat du test

Cette fonction teste si la chaîne de caractères donnée en argument est un fichier objet.

- rech_mod_par_nom :
  - entrée : une chaîne de caractères.
  - retour : entier résultat

Cette fonction recherche un module dans Table_module. S'il n'est pas présent elle retourne (-1) sinon sa place. Elle évite ainsi la double définition de module.

- init :
  - entrée : rien.
  - retour : rien

Cette fonction initialise les variables globales à leur valeur de défaut.

- main :
  - entrée : arguments de la ligne de commande (argc et argv[])
  - retour : entier

C'est la fonction « pilote ». Elle traite les option ensuite la passe1 puis la passe2.

Module passe1.c contient une fonction « principale » appelée par main : : Edl_passe1

❖ Edl_passe1 :
- entrée : rien.
- retour : entier résultat (ça s'est bien passé)

Cette fonction réalise la 1ère passe. Elle gère la Table Globale des Symboles (TGS), les adresses absolues à déterminer, les adresses d'implantation.

❖ recopier_ds_TGS :
- entrée : Symbole s et entier k
- retour : rien

Cette fonction recopie le symbole s à la place k de la TGS.

❖ traiter_entete
- entrée : entier i et pointeur fichier F
- retour : rien

Cette fonction traite l'entête du fichier traité et inscrit les renseignements récoltés (taille_code, taille_TS,…etc.) à la places i dans Table_module.

❖ traiter_TS
- entrée : entier i et pointeur fichier F.
- retour : rien

Cette fonction traite la TS du fichier traité et crée la table de référencement (Table_module[i].Table). Elle traite chaque symbole de la TS du fichier par rapport à la TGS.

Module passe2.c contient une fonction « principale » appelée par main : : Edl_passe2

❖ Edl_passe2 :
- entrée : rien.
- retour : entier résultat (ça s'est bien passé)

Cette fonction réalise la 2ème passe.Elle ouvre les fichiers temporaires fcode.temp et fdata.temp. Elle traite chaque fichier : appel du traitement section code et section data.
Elle fait rechercher le point d'entrée et fait construire l'entête.
Elle appelle à la concaténation des fichiers.
A l'issue, le fichier exécutable est générée.

❖ traiter_section_code :
- entrée : entier i, pointeur fichier F et pointeur fichier Fcode
- retour : rien

Cette fonction traite la section code du fichier F (réalocation) par rapport aux renseignements fournis à la place i dans Table_module. Elle écrit le résultat dans le fichier temporaire pointé par Fcode.

❖ traiter_section_data :
- entrée : entier i, pointeur fichier F et pointeur fichier Fdata
- retour : rien

Cette fonction traite la section donnée du fichier. Elle écrit le résultat dans le fichier temporaire pointé par Fdata.

❖ rechercher_pt_entree :
- entrée : rien
- retour : entier long non signé

Cette fonction recherche dans la TGS, la valeur de l'adresse du point d'entrée défini dans point_entree. Elle retourne sa valeur.

❖ ecrire_entete_exe :
- entrée : entier long non signé Pt_entree
- retour : rien

Cette fonction crée un fichier (le futur fichier exécutable) avec le nom pris dans nom_exec.
Elle crée l'entête du fichier exécutable.

❖ concatenation :
- entrée : rien
- retour : rien

Cette fonction réalise la concaténation dans le fichier créé par ecrire_entete_exe, des fichiers fcode.temp puis fdata.temp. A la sortie, on a donc le fichier exécutable du nom choisi (ou celui par défaut : mon_exec) où tous les modules ont été liés

c) Programme de tri sur liste doublement chaînée

1) Programme en C

Jusqu'à ce jour, notre plus grande préoccupation a été d'harmoniser notre travail. C'est pourquoi nous avons réglé en détail la forme du fichier objet. Nous avons aussi beaucoup réfléchi (sans pour autant faire l'algorithme définitif) à la librairie d'instruction.

Nous avons aussi entièrement implémenté le programme de tri en C sur les listes chaînées. Il se compose de quatre modules (comme imposé par l'énoncé) plus un module global dans lequel sont définies les variables globales et structures utilisées.

Les structures utilisées sont les suivantes :

ELMT : pointeur sur une structure composée d'un champ info (entier non signé) et d'un champ sp (entier non signé). C'est un élément de la liste

Liste : pointeur sur une structure composée d'une tête (ELMT) et d'une queue (ELMT). Elle désigne la liste.

Module 1 : mod1.c : Affichage et saisie

saisie : donnée : rien
résultat : Entier : la valeur saisie par l'utilisateur

Elle s'occupe de demander à l'utilisateur d'entrer un nombre positif ou –1 (cette valeur correspond à la sortie des boucles appelant saisie)

affichage : donnée : Entier non signé x
résultat : rien

Elle affiche x à l'écran.

menu : donnée : rien
résultat : Entier

Permet à l'utilisateur de choisir entre Ajouter des éléments à la liste (1), en retirer (2), les afficher dans l'ordre croissant (3) ou décroissant (4), et enfin de sortir du programme (5). Elle retourne le choix de l'utilisateur.

Module 2 : mod2.c : allocation et libération de la mémoire

creer_elt : donnée : Entier non signé x
résultat : ELMT

Alloue de la place pour un élément dont l'info sera x et sp sera nul (car calculé ailleurs).

liberer_elt : donnée : ELMT elt
résultat : rien

Libère la place allouée pour elt.

Module 3 : mod3.c : opération sur les listes

creer_liste_vide :donnée : rien
résultat : rien

Créé une liste vide (queue = tête = 0).

inserer : donnée : ELMT elt, Liste lt
résultat : liste lt

Insère l'élément elt dans la Liste lt en tête et renvoie lt.

rechercher : donnée : Entier non signé a
résultat : ELMT

Cherche l'entier a dans les champs info de tous les éléments de la liste. Si elle trouve un élément ayant a en info, elle le renvoie. Renvoie un pointeur sur NULL sinon.

supprimer : donnée : Liste lt
résultat : Liste lt

Supprime la tête de la liste et la renvoie. Affiche un message d'erreur si lt est vide.

supprimer_un_elt : donnée : ELMT elt
résultat : rien

Recherche l'élément et le supprime. S'il est en queue, on inverse la liste , on appelle supprimer et on inverse à nouveau la liste. S'il est à une position quelconque (autre que tete ou queue), on créé une liste temporaire de tête l'élément recherché et on en supprime la tête.

inserer_un_elt : donnée : ELMT elt
résultat : rien

Recherche la place de l'élément à insérer (recherche_prec et recherche_suiv), créé une liste temporaire de tête l'élément qui sera juste après et insère l'élément en tête (inserer).

recherche_suiv : donnée : ELMT elt
résultat : ELMT

Recherche le premier élément dont l'info est strictement supérieure à celle de ELMT et renvoie cet élément. Cet élément existe nécessairement car sinon cette fonction n'est pas appelée.

recherche_prec : donnée : ELMT elt
résultat : ELMT

Recherche le dernier élément dont l'info est strictement inférieure à celle de ELMT et renvoie cet élément. Cet élément existe nécessairement car sinon cette fonction n'est pas appelée.

imprimer_liste : donnée : rien
résultat : rien

Affiche tous les éléments de la liste.

inverser : donnée : rien
résultat : rien

Met la tête en queue et la queue en tête. La liste est ainsi inversée.

liberer_liste : donnée : rien
Résultat : rien

Libère tous les éléments (liberer_elt) de la liste.

Module 4 : mod4.c : main

remplir : donnée : rien
résultat : rien

Tant que l'utilisateur rentre un nombre positif (saisie), un élément est créé (créer_elt) puis insérer dans la liste (inserer_un_elt).

suppr : donnée : rien
résultat : rien

Tant que l'utilisateur entre un nombre positif (saisie), il est cherché (rechercher) et supprimer si trouvé (supprimer_un_elt).

main : donnée : rien
résultat : Entier

Créé une liste vide (créer_liste_vide), puis, en boucle, appelle menu puis remplir, suppr ou imprimer_liste selon le cas. Pour imprimer dans l'ordre décroissant, elle inverse d'abord la liste (inverser), puis l'imprime et l'inverse à nouveau. Pour quitter, elle appelle d'abord liberer_liste et renvoie 1 (succès).

2) Programme en MIPS

Nous avons un peu simplifier le programme pour le traduire en MIPS : la liste l étant très souvent en paramètre, nous l'avons déclarée en globale. Le programme résultant est un peu moins souple (puisque une et une seule liste appelée l est gérée alors par les modules) mais plus simple à traduire (les registres $a0 et $v0 sont moins souvent utilisés).

Comme nous faisions le plus souvent des passages de valeurs par adresses (et non par valeur : ce qui aurait conduit à la recopie des valeurs de la structure liste dans des registres, beaucoup trop lourd…Ce qui nous a fait aussi penser à faire passer notre liste en globale), nous avons modifié les modules de C que nous vous avions précédemment fourni pour les adapter à peu près à ce que le MIPS réalise. En vérité, on remarquera que le C et le MIPS ne correspondent pas exactement (notamment le passage par adresse).

De plus, le module 3 étant beaucoup plus long que les autres, nous avons réparti le travail en le considérant aussi grand que la réunion des trois autres.

d) Problèmes et tests

L'assembleur a mis quelques temps a pouvoir produire un fichier objet.
On a pu tester avec les exécutables fournis par l'éditeur de liens avec Ecrire et Ouvrir.

Les modules avec lesquels on a fait les tests ont été fact.s et principal.s . On ne pouvait que choisir des modules pas trop longs car le calcul et la saisie pour créer un fichier objet (par Ecrire) est longue et fastidieuse. De plus, il contenait assez importante gamme d'instructions et quelques commentaires, ce qui a permis de tester l'assembleur.

Les problèmes de l'assembleur ont été variés : l'utilisation prévue initialement de certaines fonctions n'a pas été faite, ce qui a considérablement augmenté le nombre de lignes de codes. Par exemple, une instruction fprintf aurait du remplacer les fwrite en cascade. Son utilisation (comme celle de la fonction pow) a donné des résultats quelques fois étranges. Elle a été remplacée par des décalages. L'information enreg n'a pas été prise en compte dans l'algorithme de départ. Son utilisation aurait permis de réduire le nombre de lignes (notamment avec des switch) et de variables (comme forme dans Analyse_instruction). Quelques erreurs se sont aussi glissées, principalement des oublis lors de la traduction de l'algorithme en programme C. Erreurs qui se sont répétées à divers endroits du programme via les copier-coller. La plupart de ces erreurs ont été rectifiées avec l'utilisation de fact.s et du programme Ouvrir créé par l'éditeur de liens. Par exemple, pour la vérification des registres, le caractère # terminal a été initialement oublié. Ou encore, pour les instructions de type I, la non utilisation d'un masque provoquait un dépassement sur 15ème bit.

Les problèmes de l'éditeur de liens ont été assez minimes : erreurs de recopie (inversion lors de la relocation l'adresse initiale de code alors qu'il fallait l'adresse initiale des données).
Des problèmes résolus grâce aux affichages (les fonctions dans utils ont été effectuées pour pouvoir voir comment le fichier objet était traité. Par ailleurs la fonction traiter_section_code fait afficher toutes les instructions et montre comment elle les traite).

A ce jour, les modules sur les listes doublement chaînées n'ont pu générer un exécutable correct.
(Lorsque l'on charge par lbxspim, il y erreur…Nous ne savons pas de qui proviens exactement l'erreur car nous n'avons pas eu le temps d'y réfléchir - suite au mail nous conseillant d'arrêter de debugger nos programmes.).
De plus, au moins un gros problème subsiste quant à l'assembleur puisque le fait de changer les #include de place génère une erreur sur l'exécutable final.

CONCLUSION

Bien que ce projet ne soit pas entièrement terminé, il nous aura appris un certain nombre de choses. Notamment, il nous est apparu que nous avions relativement mal compris le cours théorique ; nous avons donc remplit certaines lacunes et rectifié nos croyances.

Il nous a fallu commencer le projet en travaillant ensemble (format du fichier objet), puis séparément (assembleur et éditeur de liens) et enfin revenir ensemble. Ceci nous a permis de voir les problèmes dû à la répartition du travail. De plus, n'ayant pas terminé le projet dans les temps, nous avons connu les inconvénients des « retouches de dernière minute ».

D'une manière générale, nous gardons un souvenir positif quant au projet et nous pensons qu'il n'aura pas été fait en vain. Nous espérons le finir avant l'année prochaine (pendant les vacances de Pâques?) pour pouvoir le compléter avec le projet de compilation et ainsi être sûrs d'avoir bien compris toutes les étapes menant à la compilation d'un programme.

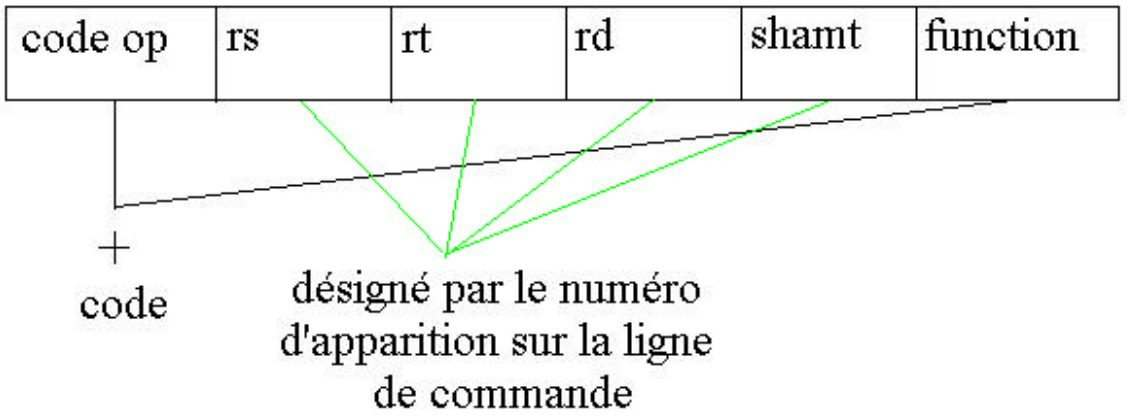
ANNEXES

LISTE DES INSTRUCTIONS GEREES

ADD	Addition (with overflow)
ADDI	Addition Immediate (with overflow)
ADDU	Addition (without overflow)
ADDIU	Addition Immediate (without overflow)
AND	And
ANDI	And Immediate
BEQ	Branch Instruction
BEQZ	Branch on Equal Zero
BGEZ	Branch on Equal Than Equal Zero
BGEZAL	Branch Equal Than Equal Zero and link
BGTZ	Branch on Greater Than Zero
BLEZ	Branch on Less or Equal Zero
BLTZ	Branch on Less Than Zero
BLTZAL	Branch on Less Than Zero And Link
BNE	Branch on Not Equal
DIV	Divide (signed)
DIVU	Divide (unsigned)
J	Jump
JAL	Jump And Link
JALR	Jump And Link Register
JR	Jump Register
LA	Load adress
LB	Load Byte
LBU	Load Byte Unsigned
LH	Load Halfword
LHU	Load Halword Unsigned
LI	Load Immediate
LUI	Load Upper Immediate
LW	Load Word
LWL	Load Word Left
LWR	Load Word Right
MFHI	Move From High
MFLO	Move From Low
MTHI	Move To High
MTLO	Move To Low
MOVE	Move
MULT	Multiply
MULTU	Unsigned Multiply
NOP	No Operation
NOR	Nor
OR	Or
ORI	Or Immediate
RFE	Return From Exception
SB	Store Byte
SH	Store Halfword
SLL	Shift Left Logical
SLLV	Shift Left Logical Variable
SLT	Set Less Than
SLTI	Set Less Than Immediate
SLTIU	Set Less Than Immediate Unsigned
SLTU	Set Less Than Unsigned
SRA	Shift Right Arithmetic
SRAV	Shift Right Arithmetic Variable
SRL	Shift Right Logical
SRLV	Shift Right Logical Variable
SUB	Substract (with overflow)
SUBU	Substract (without overflow)
SW	Store Word
SWL	Store Word Left
SWR	Store Word Right
SYSCALL	Syscall
XOR	Xor
XORI	Xor Immediate

TYPE D'INSTRUCTIONS

Type R : il s'agit de celui vu en cours. Il est défini ici par le type R1 : un nom (nom de l'instruction), un code (il s'agit du code op auquel on a ajouté tout ce qui était constant : voir schéma), trois registres rs, rt et rd, et enfin un champ shamt.



Un Exemple concret :
Add rd,rs,rt. Code op : 0. Champ function : 32.
Cette instruction est désigné par :

code = 0 + 32
rs = 2 (deuxieme argument) ; rt = 3 (troisieme argument) ; rd = 1 (premier argument)
shamt = 0 (n'apparait pas sur la ligne de commande).

Type I (ou I1): Il s'agit d'une partie des instructions de type I vues dans le cours : seulement celles qui acceptent comme dernier argument seulement un immédiat (ex : addi). Elles sont définies par le type I1 : un nom, un code et deux registres rs et rt (il est sous entendu que l'instruction doit se terminer par un immédiat) qui suivent les même conventions que pour le type R.

Type i (ou I2): Il s'agit des instructions de type I du cours qui n'acceptent qu'un label pour dernier argument. En pratique il s'agit de toutes les instructions de branchement conditionnel. Elles sont définies de la même manière que les précédentes instructions (I1).

Type p (ou I3): Les instructions de type I qui acceptent pour dernier argument une adresse. Il peut s'agir d'un pseudo-mode d'adressage (d'où le nom p). Elles désignées par le type p1 : le nom et le code. Il est sous-entendu qu'un registre est nécessaire suivi d'une adresse (qui peut être : un immédiat, un label, un immédiat suivi d'un déplacement ou un label suivi d'un déplacement).

Type J : Il s'agit du type J vu en cours. Il est lui aussi désigné seulement par son nom et son code (p1). Le label final est sous-entendu.

Type P : Ce sont les pseudo-instructions. Elles sont traitées au cas par cas par l'analyseur.

FORMAT DU FICHIER OBJET

<table>
<tr><td rowspan="4">Structure de l’entête</td><td colspan="3">Nom (11 caractères max.)</td></tr>
<tr><td colspan="3">Taille de la TS (éléments)</td></tr>
<tr><td colspan="3">Taille du code (en octets, et sans compter le champ d’info)</td></tr>
<tr><td colspan="3">Taille des données (octets)</td></tr>
<tr><td rowspan="6">Structure de la définition des Symboles (TS) :

Soit : TS[taille TS]</td><td colspan="3">Nom du symbole (9 caractères max.)</td></tr>
<tr><td rowspan="5">1 octet « Olpse » / « donnee » (Rem : les locaux sont éliminés du fichier objet : l’éditeur de liens ne teste que le « 2ème bit de ce champ »)</td><td colspan="2">4 bits à 0</td></tr>
<tr><td rowspan="3">2 bits de portée</td><td>Local = 01</td></tr>
<tr><td>Global =11</td></tr>
<tr><td>Externe =10</td></tr>
<tr><td colspan="2">1 bit de section (1=text ; 0=data)</td></tr>
<tr><td colspan="2">1 bit état (1)défini et (0)non.</td></tr>
<tr><td colspan="3">Adresse relative (entier long non signé)</td></tr>
<tr><td rowspan="11">Structure de la définition des instructions (Code) :

Soit : Code[taille Code/4].</td><td rowspan="2">1 octet : « enreg »

(on remarquera que cet octet est écrit avec la place d’un entier)</td><td>1 bit à 0</td><td></td></tr>
<tr><td rowspan="5">3 bits de type</td><td>R=000</td></tr>
<tr><td>I1=001</td></tr>
<tr><td>I2=010</td></tr>
<tr><td>I3=011</td></tr>
<tr><td>J=100</td></tr>
<tr><td rowspan="4">2 bits d’info de relocation</td><td>Définie=00</td></tr>
<tr><td>Relog/code=01</td></tr>
<tr><td>Relog/data=10</td></tr>
<tr><td>Ref.externe=11</td></tr>
<tr><td rowspan="2">2 bits d’info supplément.</td><td>Bas=01</td></tr>
<tr><td>Haut=10</td></tr>
<tr><td colspan="3">Valeur de ce que l’on a pu traduire (+ n° ref.ext.) (4 octets) : « val »</td></tr>
<tr><td>Définition de la section Data.

Soit :Data[taille Données]</td><td colspan="3">Caractère.</td></tr>
</table>