Présentation emul-mips

Thibaut Barnel, Pierre Coimbra

Décembre 2020

1 Projet CS351 : emul-mips

Nous allons présenter ici le fonctionnement de notre code. Bien que les commentaires dans le code permettent de comprendre le fonctionnement de base du code il est recommandé de lire ce document pour en comprendre les détails.

Nous allons aborder les points suivant :

- Le stockage des informations associé aux opérations
- La vérification de la structure des instructions
- Le fonctionnement de la table des symboles pour la gestion des labels
- Notre implémentation des registres
- Notre implémentation de la mémoire

2 Les fonctionnalités clé de notre code et les modules de base

2.1 Vérification de la structure des instructions

Cette fonctionnalité permet de s'assurer que les arguments passé dans l'instruction sont du bon type.

Par exemple l'instruction `ADD $1,$2,3` n'est pas correcte car le troisième argument doit être de type registre.

Pour cela nous avons mis en place un système de somme de contrôle elle devra :

- Assurer l'unicité des combinaisons, une valeur ne pourra correspondre qu'a une combinaisons
  - Par exemple 1,2,3 n'est pas unique car 3 peut être 1+2 ou juste 3
- Permettre d'identifier quel.s argument.s sont de mauvais type mais aussi de quel type ils doivent être
- Être indépendant de la valeurs des arguments
- Être le plus simple possible

Pour cela nous allons mettre en place une somme de contrôle pour les arguments de type registre et une pour les arguments de type immédiat.

Nous partons du principe qu'il ne pourra pas y avoir plus de 3 type registres et 3 type immédiat car nous sommes dans une architecture de processeur (3,0). A noter qu'un des arguments peut être un adressage indirect par registre avec déplacement, c'est à dire de la forme suivante : `imm($reg)`

Voici comment nous allons procéder pour le calcul de cette checksum :

Prenons l'exemple des arguments suivant `ARG1 ARG2 ARG3`

On parcourt les arguments un par un, en fonction du type (registre ou immédiat) on pondère la somme de contrôle correspondante par la puissance de 2 du numéro de l'argument.

Par exemple pour `ADD $10,$11,$12` on aura :

```
checksumReg=1*2^1+1*2^2+1*2^3=14
checksumImm=0*2^1+0*2^2+0*2^3=0
```

et pour `SW $10, 10,($2)` on aura :

```
checksumReg=1*2^1+0*2^2+1*2^3=10
checksumImm=0*2^1+1*2^2+0*2^3=4
```

Ainsi on pourra vérifier que les arguments soient du bon type en comparant les checksums théorique stockée dans la structure de l'opération avec les checksums calculées. Si les deux ne correspondent pas on est capable de dire d'où vient le.s erreur.s.

2.2 Stockage des informations associé aux opérations

Cette partie est une des clés de voûte de notre programme. Elle nous permet d'associer une opération à 3 critères qui nous permettent de regrouper les 28 opérations en 16 groupes. Cette fonctionnalités nous serra utile aussi bien lors de la traduction des instructions en hexadécimal que lors du décodage des codes hexadécimal pour l'exécution et nous permettra aussi de factoriser des fonctionnalités dans le code.

Nous avons trié les instructions selon trois critères :

- Le type d'instruction (R,I ou J)
- L'ordre des bits dans la valeur binaire de l'instruction c'est à dire le positionnement des champs de bits dans l'instruction
- Le style de remplissage c'est à dire ce que l'on doit mettre dans les différents champs de bits de l'instruction

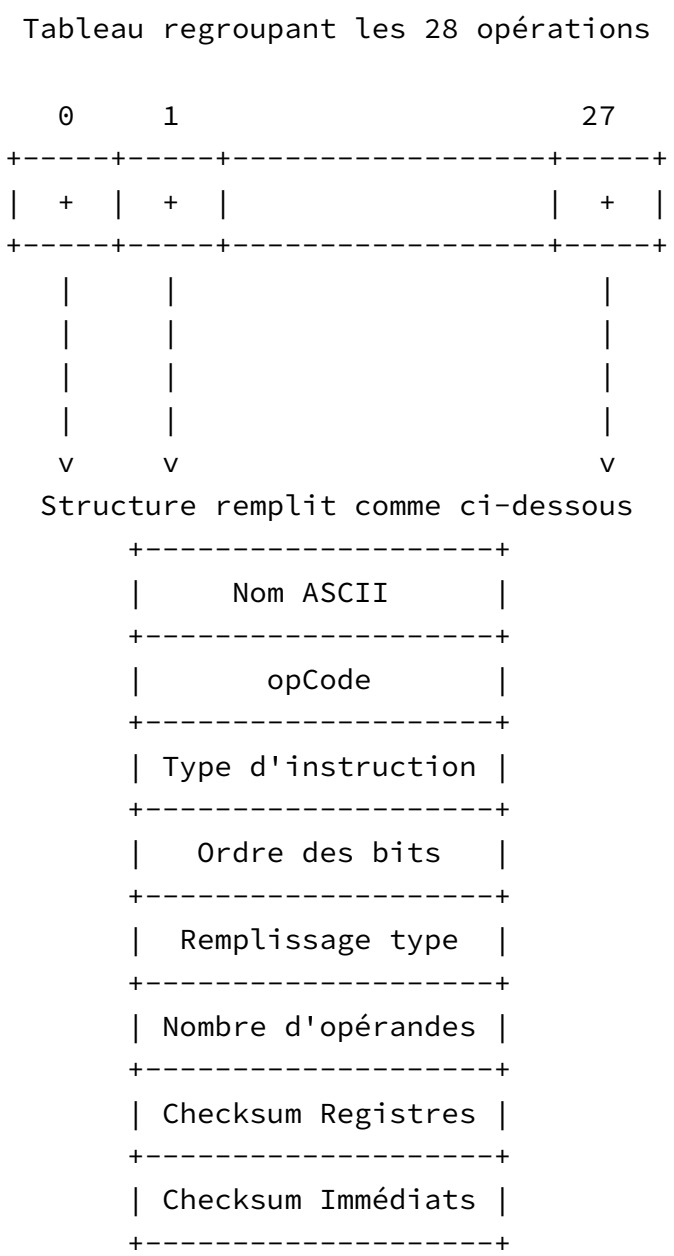
En plus de ces critères nous stockons l'opcode et le nom ASCII de l'opération afin de pouvoir encoder ou décoder les instructions.

Nous avons ajouter trois paramètres qui nous permettent de vérifier que l'instruction est valide :

- Le nombre théorique d'opérande que l'instruction requiers que nous comparerons au nombre d'instructions trouvé dans l'instruction lors du parsage.
- La checksum des arguments de type registre
- La checksum des arguments de type immédiat

Nous allons détailler ces deux derniers points dans la partie suivante

Voici comment notre structure est représenté en mémoire :



Ainsi que la déclaration de notre structure :

```
typedef struct instruction {
  char nom[TAILLE_MAX_OPERATEUR]; /* en ascii */
  unsigned int opcode; /* entier de l'instruction */
  char typeInstruction; /* 'R' || 'I' || 'J' */
  int ordreBits; /* Ordre des bits */
  int styleRemplissage; /* Style de remplissage des champs */
  int nbOperande; /* Nombre d'opérandes requis pour fonctionner */
  int checksumReg; /* Checksum théorique des registres */
  int checksumImm; /* Checksum théorique des immédiats */
} instruction;
```

Pour plus de détail l'annexe 1 décrit comment nous avons séparées les instructions en 3 familles.

Résumons ce qu'il se passera quand on recevra une ligne de code assembleur :

- On met au propre l'instruction, chaque partie est séparée de la précédente par un espace on aura quelque chose de la forme `OPE ARG1 ARG2 ARG3`
- A partir de cette instruction traitée on génère un segment assembleur affichable de la forme `OPE ARG1,ARG2,ARG3` dans le même temps,
  - On remplit un tableau contenant les valeurs entière des arguments de l'instruction.
  - On calcule les sommes de contrôle pour les registres et les arguments.
  - Si un argument est un label on recherche la correspondance du label dans la table des symboles et on place la correspondance du label dans le tableau des opérandes.
- En cas de non correspondance de l'opération, du nombre d'opérandes ou des sommes de contrôle on ne prend pas en compte l'instruction et on passe à la suivante. Dans tout les autres cas on passe au calcul de la valeur hexadécimale de l'opération.

2.3 Gestion des labels

La gestion des labels et nécessaire pour les instructions de type J, ici nous n'implémenterons que les instructions J et JAL. Il faudra associé un label à la valeur du programCounter de l'instruction suivante (même si il n'y en a pas). Ces correspondances seront stocké dans la table des symboles que nous avons implémenté dans le module table.

Par exemple avec le programme suivant :

```
00056812 ADDI $20,$0,8
label:
00056816 NOP
00056820 ADDI $18,$18,1
00056824 ADDI $19,$19,4
00056828 BEQ $19,$20,4
00056832 JAL label
00056836 NOP
00056840 JAL exit
00056844 NOP
exit:
```

On aura la table des symboles suivante :

```
label  56816
exit   56848
```

Il important de remarquer qu'un label peut être déclaré après son appel (c'est le cas du label exit). Il faudra donc remplir la table des symboles avant d'exécuter le programme.

Pour les labels nous avons donc une fonction qui analysera tout le code et qui remplira la table des symboles avec l'exécution du programme. Il faudra bien vérifier que l'instruction est valide pour ne pas incrémenter le programCounter inutilement afin d'avoir la bonne correspondance label/PC dans la table des symboles.

Cette fonction ferra un peu doublons mais elle est nécessaire au bon fonctionnement des labels.

A noter que dans l'affichage on continuera d'afficher le label et non la valeur du programCounter correspondante.

Pour l'implémentation de la table des symboles nous allons faire une liste chaînée contenant un string pour stocker le symbole, un entier pour stocker la valeur et l'adresse de l'élément suivant.

Nous implémenterons les fonctionnalités suivantes :

- Insertion en tête
- Insertion en queue
- Suppression en tête
- Affichage de la table des symboles
- Libération de la table des symboles
- Remplissage de la table des symboles

Pour ce dernier points nous allons faire comme décrit plus haut, c'est à dire analyser tout le code pour déterminer pour associer chaque label au program counter de l'instruction qui suit le label.

Voilà comment fonctionne notre système de vérification des instructions

2.4 Module registres

L'objectif de ce module est de fournir une implémentation des registres d'un processeur MIPS en C. Ce module permettra de lire la valeur d'un registre, de la changer et de traduire un registre mnémonique.

Pour la représentation en mémoire de ces registres, nous avons procédé de la manière suivante, une structure représentant un registre organisé de la manière suivante :

```
+----------------------+
|  Numéro du registre  |
+----------------------+
|       Nom ASCII      |
+----------------------+
|  Valeur du registre  |
+----------------------+
```

Le numéro du registre, le nom mnémonique et la valeur du registre seront initialisés par une fonction à l'aide d'un fichier. Par défaut, la valeur du registre sera initialisée à une valeur arbitraire pour éviter des valeurs aléatoires.

Afin de pouvoir utiliser la même structure pour tout les registres, pour tout numéro inférieur ou égal à 31 nous avons affaire à un registre classique par contre si le numéro du registre est supérieur à 31 on à affaire à un registre spécialisés.

Cela permet de simplifier la recherche de ces registres.

- Le numéro du registre sera un nombre entier entre 0 et 34, on le stockera sous la forme d'un int.
  - Les registres spéciaux PC,HI,LO auront comme numéro respectif 32,33,34
- Le nom mnémonique sera un tableau de 5 char car le plus long est zero.
- La valeur du registre sera un unsigned long int (non signé grâce au complément à deux).

Il y a donc 35 registres pour notre implémentation, nous allons donc utiliser un tableau de 35 cases ou chaque case pointera vers une structure décrite ci-dessus qui sera associé au registre correspondant.

Les 35 registres seront donc stockés dans un tableau ou chaque case contiendra la structure associée au registre portant le numéro de la case.

Ainsi pour trouver la structure associé à un registre, il nous suffira d'aller voir dans la case du tableau associé au registre, on pourra à partir de là consulter ou modifier la valeur du registre. Nous n'avons donc pas besoin d'une fonction pour

2.5 Module mémoire

Une case mémoire est composée de 32 bits

La mémoire est stockée entre les adresses 0x0000 et 0xFFFF.

Les instructions pouvant accéder à la mémoire dans notre implémentation sont LW et SW. Elles permettent de faire le lien entre un registre et la mémoire en transférant un mot de 32 bits soit 4 octets.

Ces 4 octets devront être stockés à la suite dans la mémoire en respectant la norme Big Endian (octet de poids fort à l'adresse la plus basse). De plus, l'adresse du premier octet d'un mot devra être divisible par 4.

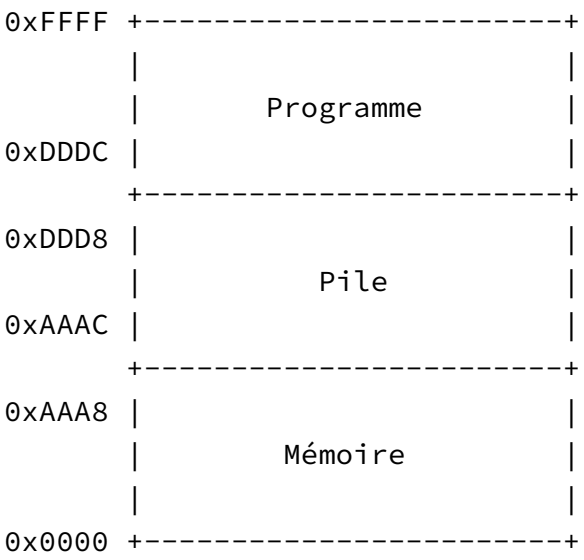
En résumé la quantité de mémoire utilisée par un programme n'est pas prédictible.

La première question soulevée est la suivante : Faut-il avoir une mémoire gérée par mots (bloc de 4 octets) ou une mémoire gérée par octet ?

Vus que pour notre implémentation, nous n'utilisons pas les directives et que les deux seules opérations accédant à la mémoire sont LW et SW.

Nous avons fait le choix d'une implémentation modulaire sous forme de liste chaînée supportant à la fois l'adressage à l'octet et l'adressage au mot. Dans notre cas, nous stockerons que des mots, l'adresse dans la mémoire à laquelle ce mot sera accessible devra donc être l'adresse du premier octet de ce mot (et donc un multiple de 4).

Nous allons utiliser la map mémoire suivante.



La mémoire est accessible avec les instructions *LW* et *SW* se trouvera donc entre les adresses 0x0000 et 0xAAA8 et sera remplit des adresses bases aux adresses hautes.

Pour la pile, elle se remplit par le bas donc elle se trouvera entre 0xDDD8 et 0xAAAC. Cela nous laisse de la place si on veut par la suite ajouter des parties à notre map mémoire.

Le programme sera lui stocké entre les adresses 0xDDDC et 0xFFFF. Ce qui est suffisant pour stocker notre programme dans notre cas.

Vu que la mémoire ne sera pas trop utilisée nous allons utiliser une liste chaînée dynamiquement alloué pour utiliser le moins de mémoire possible sur l'hôte.

On aura donc la map mémoire qui contiendra :

- Une partie pour la mémoire
- Une partie pour le programme
- Une partie pour la pile descendante

L'utilisation de la pile est gérée par le registre *sp* qui pointe vers l'adresse du haut de la pile. On peut donc utiliser la même implémentation pour la pile et pour la mémoire seule l'utilisation varie.

De même pour le programme, l'instruction en cours sera pointée par le registre *PC*. À noté qu'avant d'incrémenter le PC il faudra vérifier que l'on est bien une case après.

Pour l'implémentation, nous avons retenu celle d'une unique liste chaînée.

Chaque maillon de la liste sera composé comme décrit ci-dessous :

- L'adresse de la case mémoire du premier octet du mot (un multiple de 4)
- La valeur du mot sous forme de tableau

- Un pointeur vers l'adresse suivante, si ce pointeur vaut NULL, c'est la fin de la mémoire

Pour des questions de simplicité, on veillera à ce que les adresses reste dans l'ordre. De plus, si on insère une valeur dans une case mémoire déjà existante, on remplacera la valeur.

En résumé, un maillon contiendra un mot et sera désigné par l'adresse de la case mémoire du premier octet du mot.

Si l'on souhaite un jour stocker par octet il suffira de rajouter des fonctions d'insertion pour un octet, l'implémentation ne changera pas.

Maintenant que nous avons vu les partie du code permettant de faire fonctionner notre simulateur nous allons voir comment est traitée une instruction

Pour rappel voici la liste de nos modules

- Module hex -> Permet de parser et de traduire une instruction en hexadécimal
- Module table -> Fourni une implémentation de la table des symboles, nécessaire pour les labels
- Module registre -> Fourni une implémentation statique des registres (un tableau de 35 cases) et permet de traduire les mnémoniques
- Module mémoire -> Fourni une implémentation dynamique de la mémoire (une liste chainée)
- Module outils -> Fourni des fonctions classique nécessaire dans plusieurs modules (ex. valeurDecimale, complément à deux…)
- Module calcul -> Permet exécuter une instruction en mettant à jour la mémoire et les registres à partir du code hexadécimal de notre fonction
- Module console -> Utilise les modules précédents pour fournir un affichage à l'utilisateur dans plusieurs modes de fonctionnement (automatique, pas à pas et intéractif)

Nous n'avons pas encore abordé les 3 derniers modules, pour le module outils sont fonctionnement est clair nous n'en parlerons donc pas, nous allons aborder le reste dans la suite.

2.6 Gestion de l'affichage et modes de fonctionnement

2.6.1 Mode automatique

Commande : `emul-mips in.txt out.txt`

Dans ce mode le programme prend en entrée deux fichiers, un contenant les segments assembleur à simuler et un de sortie dans lequel on écrira les valeurs hexadécimale des segments.

L'exécution se passe de la manière suivante :

- Première lecture du fichier pour remplir la table des symboles
- Seconde lecture du fichier pour traduire les segments en hexadécimal. Dans cette partie on met les valeurs hexadécimal dans la mémoire, dans le fichier de sortie et dans la structure d'affichage
- On affiche tout les segments et leurs valeur hexadécimale respective et on attend une pression sur entrer pour exécuter le programme
- Pour l'exécution on lit les instructions dans la mémoire à l'adresse du program counter, on exécute les instructions puis on incrémente le program counter pour passer à la suivante.
- A la fin de l'exécution on affiche l'état des registres et de la mémoire avant de tout libérer pour éviter les fuites mémoire

2.6.2 Mode pas à pas

Commande : `emul-mips -pas in.txt out.txt`

Le fonctionnement est proche de celui du mode automatique à la différence des points suivant :

- Lors de la seconde lecture on fait une pause entre chaque instruction et l'utilisateur peut choisir entre les options suivantes
  - Passer l'instruction (ne pas utiliser si il y a des labels)
  - Aller à l'instruction suivante
  - Sauter le lecture pour passer à l'exécution
- Lors de l'exécution il y a une pause entre chaque instruction et l'utilisateur peut choisir entre les options suivantes :
  - Afficher l'état actuel des registres
  - Afficher l'état actuel de la mémoire
  - Afficher le programme avec une flèche indiquant notre avancement dans le programme
  - Passer à l'instruction suivante

2.6.3 Mode interactif

Commande : `emul-mips -pas out.txt`

Dans ce mode le programme prend en entrée un fichier de sortie dans lequel on écrira les valeurs hexadécimale des segments.

L'exécution se passe de la manière suivante :

- On attend la saisie d'une instruction par l'utilisateur, à ce moment :
  - Si l'instruction est invalide on retourne à l'étape d'avant
  - Sinon on passe à l'étape suivante
- Une fois une instruction correcte saisie on propose à l'utilisateur de choisir entre :
  - Afficher l'état actuel des registres
  - Afficher l'état actuel de la mémoire
  - Afficher le programme actuel avec une flèche indiquant notre avancement dans le programme
  - Exécuter l'instruction
- Une fois l'instruction exécuter l'utilisateur peut choisir entre :
  - Afficher l'état actuel des registres
  - Afficher l'état actuel de la mémoire
  - Afficher le programme actuel avec une flèche indiquant notre avancement dans le programme
  - Passer à la saisie d'une autre instruction

L'instruction EXIT permet d'interrompre la saisie et affiche l'état final des registres et de la mémoire

2.6.4 Fonctionnement de l'affichage des segments en cours d'exécution

Nos différents modes on besoin de pouvoir afficher l'intégralité des segments assembleur à tout moment de l'exécution or ce n'est pas possible dans l'état actuel car nous stockons uniquement la valeur hexadécimale de l'instruction dans la mémoire mais pas le string associé à l'opération.

Pour cela nous allons devoir ajouter une mémoire que nous utiliserons seulement pour l'affichage et qui contiendra :

- La valeur du program counter associé à l'instruction
- La valeur hexadécimale de l'instruction
- La chaine parsé du segment assembleur

Cela prendra la forme d'une liste chainée car nous ne pouvons pas prévoir la longueur du programme à l'avance.

Voici la définition de la structure associée

```
typedef struct segment {
  int pc;
  unsigned long int hex;
  char *asem;
  struct segment* suivant;
} segment;

typedef segment* prog;
```

On a implémenté des fonctions permettant :

- D'insérer un segment
- D'afficher les segments mis en forme
- De libérer tout les segments pour éviter les fuites mémoire

C'est tout pour la gestion de la console utilisateur

2.7 Cycle de traitement d'une instruction

Le traitement de l'instruction ce fait via les modules hex et calcul.

2.7.1 Module hex

Nous avons déjà vu la partie parsage et vérification de la forme de l'instruction, nous allons voir ici la traduction en hexadécimal.

La partie parsage nous fourni un pointeur vers une structure contenant toutes les informations sur l'opération de l'instruction.

A partir de ces informations nous allons associer une valeur à chaque partie de notre instruction en fonction : du type d'instruction, de l'ordre des bits et du style de remplissage.

Une fois que les différents champs de notre opérations sont remplit, on détermine la valeur hexadécimale grâce à une succession de décalage et de OU binaire. Cela revient à mettre côte à côte toutes les champs pour obtenir le code de l'instruction sur 32 bits.

L'avantage est que nous n'avons ni à utiliser des tableaux ni à convertir du binaire en hexadécimal ou autre.

2.7.2 Module calcul

Ce module permet d'exécuter une instruction à partir de son code hexadécimal.

- On commence par identifier le type d'instruction à l'aide d'un masque sur les 6 derniers bits pour savoir si on à une instruction de type R ou de type I/J
- On identifie l'opcode à l'aide d'un autre masque puis on recherche l'instruction associé à cet opcode dans notre structure principale
- Une fois notre opération trouvée dans la structure on extrait de l'instruction la valeur de chaque opérande en utilisant des masques et des décalage binaire. On identifie les champs à extraire grâce à l'ordre des bits (fournis par la structure principale)
- Une fois que c'est fait on exécute l'instruction et on met à jour les registres et la mémoire. On identifie l'action à effectuer à l'aide du style de remplissage et de l'opcode.

Cette fonction s'occupe de mettre à jour le program counter afin que la fonction appelante (module console) puisse récupérer l'instruction suivante dans la mémoire avant de l'exécuter.

3 Répartition des tâches

Nous avons élaboré ensemble la structure principale qui nous permet de stocker toutes les informations des structures. Nous avons commencé par séparer toutes les instructions en plusieurs groupes comme décrit dans l'annexe 1 puis nous en avons fait l'implémentation en C à l'aide d'une structure et d'un fichier de stockage pour remplir la structure.

3.1 Tâches effectuées par Pierre Coimbra

Voici un résumé de mes contributions dans les différents modules de notre code.

3.1.1 Module Hex

Pour ce module, je me suis chargé de concevoir le parseur que nous avons détaillé plus haut. Il y a eu une première version très simple du parseur qui ne vérifiez ni que les paramètres passé soient du bon type ni que l'on passé le bon nombre de paramètres et qui ne supportez pas les labels. Cette version, bien que fonctionnelle, ne me satisfaisait pas, j'ai donc profité de l'ajout du support des labels pour totalement refaire le parseur. Même si il est plus complexe que le premier il s'occupe lui-même d'extraire les informations de l'instruction (opération et opérandes) ce qui n'était pas le cas avant.

J'ai aussi implémenté la fonction hexLigne qui permet de générer le code hexadécimal d'un segment assembleur. C'est la fonction principale du module hex (qui correspond à la v1). Cette fonction s'appuie sur le parseur et sur la structure principale de contrôle.

3.1.2 Module Console

J'ai mis en place l'interface utilisateur qui se présente sous la forme d'une console et qui utilise les autres fonctions pour faire fonctionner notre simulateur. Une partie de l'affichage dépend sur la structure de stockage des segments implémenté par Thibaut.

3.1.3 Module mémoire

Je me suis occupé de mettre en place le module mémoire. C'est-à-dire la map mémoire et l'implémentation qu'on en a fait sous forme de liste chaînée gardée triée par adresse. Nous allons utiliser ce module pour tous les accès en mémoire que ça soit en lecture ou en écriture.

3.1.4 Module tools

Au niveau du module tools j'ai implémenté la fonction hexToDec qui nous est utile pour le support des valeurs hexadécimales dans les instructions.

3.1.5 Module calcul

J'ai implémenté l'intégralité du module calcul qui permet d'exécuter un segment assembleur à partir de son code hexadécimal. Il supporte toutes les instructions présentes dans l'annexe 2.

3.1.6 Labels

Pour finir je me suis occupé du support des labels et donc de la création du module table qui fournit une implémentation de la table des symboles sous forme de liste chaînée et une fonction permettant de remplir la table des symboles avant l'exécution des segments assembleurs. Ainsi que de la modification du module calcul, pour qu'il fonctionne avec des instructions de type J.

3.1.7 Conclusion personnelle

Ce projet m'a beaucoup apporté au niveau programmation en C, j'avais déjà un niveau correct étant donné que je viens du premier cycle, mais c'est la première fois que j'étais sur un projet contenant plusieurs modules et de nombreuses fonctions. Ce projet était donc mon premier "gros" projet de développement en binôme et j'ai beaucoup apprécié. En plus de ça cela m'a permis d'apprendre de nouvelles choses en C comme par exemple les décalages binaire et les manipulations binaire en général. J'ai aussi appris à faire attention à la "qualité" de notre code ce qui n'était pas forcément le cas lors d'un TP par manque de temps. On a fait attention aux fuites mémoire ainsi qu'aux réactions de notre code à des situations non prévu (segfault ou buffer overflow suite à une entrée inattendu par exemple). Ce qui m'a fait découvrir des outils comme Address Sanitizer et Valgrind pour vérifier l'absence de fuites mémoire ou de problème dans les malloc.

3.2 Tâches effectuées par Thibaut Barnel

4 Annexe 1

Les instructions sont séparées en 3 familles que nous avons recoupé en groupes et sous-groupes comme décrit ci-dessous.

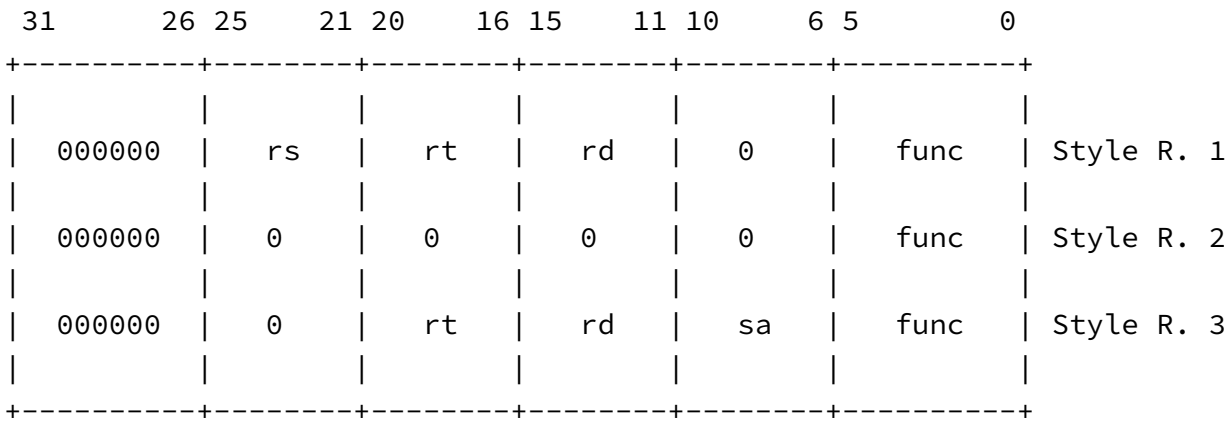
4.0.1 Instruction de type R

Les bits de 31 à 26 contiennent des 0.

Différents cas en fonction du type de type R

4.0.1.1 Ordre des bits 1

Ordre des bits 1 - Type R

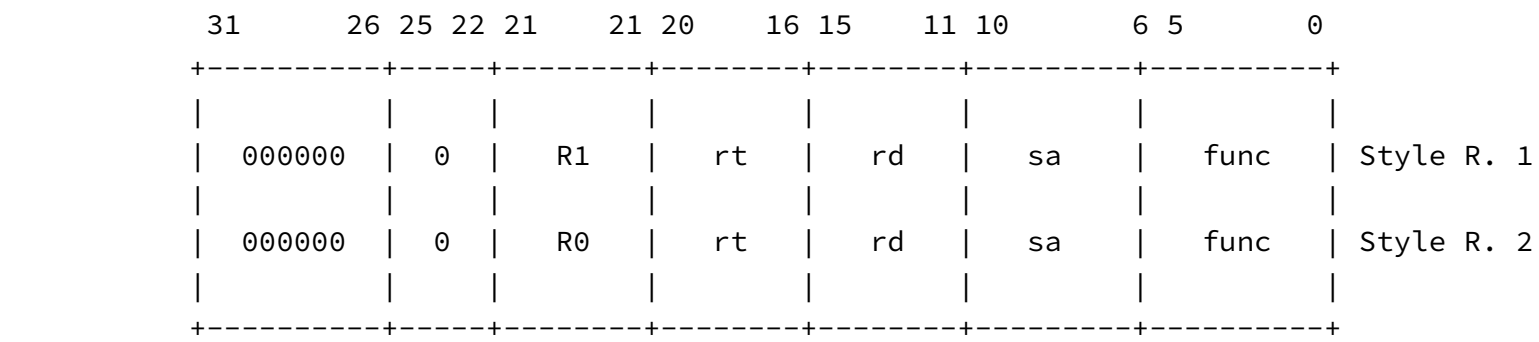


4.0.1.1.1 Style de remplissage 1 : (0/rs/rt/rd/0/func)

- ADD : 100000 (25/21 20/16 15/11 10/6 5/0)
- AND : 100100 (25/21 20/16 15/11 10/6 5/0)
- XOR : 100110 (25/21 20/16 15/11 10/6 5/0)
- OR : 100101 (25/21 20/16 15/11 10/6 5/0)
- SLT : 101010 (25/21 20/16 15/11 10/6 5/0)
- SUB : 100010 (25/21 20/16 15/11 10/6 5/0) ##### Style de remplissage 2 : (0/0/0/0/0/func)
- NOP : 000000 (25/21 20/16 15/11 10/6 5/0) ##### Style de remplissage 3 : (0/0/rt/rd/sa/func)
- SLL : 000000 (25/21 20/16 15/11 10/6 5/0)

4.0.1.2 Ordre des bits 2

Ordre des bits 2 - Type R

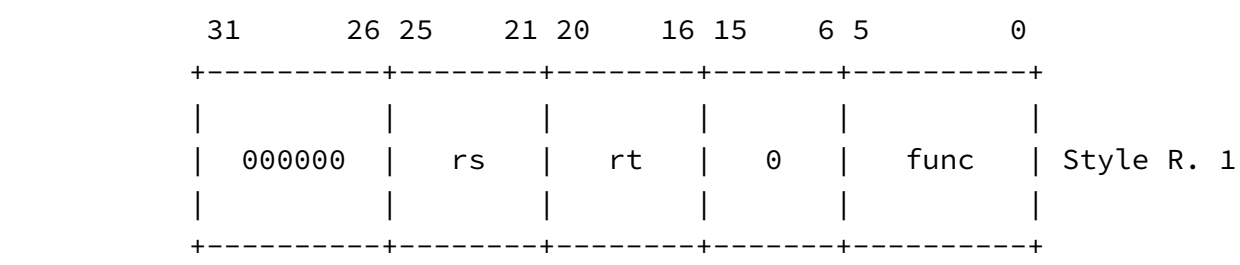


4.0.1.2.1 Style de remplissage 1 : (0/0/R1/rt/rd/sa/func)

- ROTR : 000010 (25/22 21 20/16 15/11 10/6 5/0) ##### Style de remplissage 2 : (0/0/R0/rt/rd/sa/func)
- SRL : 000010 (25/22 21 20/16 15/11 10/6 5/0)

4.0.1.3 Ordre des bits 3

Ordre des bits 3 - Type R

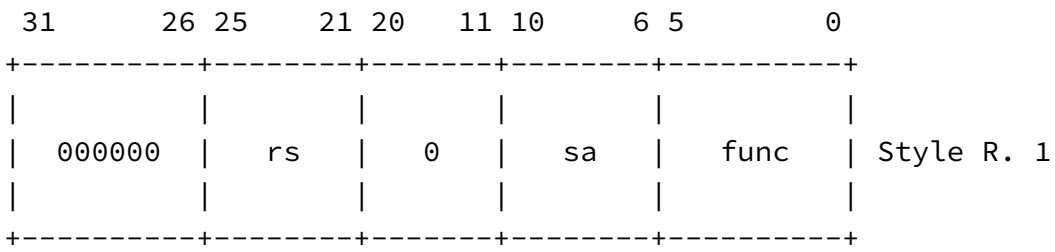


4.0.1.3.1 Style de remplissage 1 : (0/rs/rt/0/func)

- MULT : 011000 (25/21 20/16 15/6 5/0)
- DIV : 011010 (25/21 20/16 15/6 5/0)

4.0.1.4 Ordre des bits 4

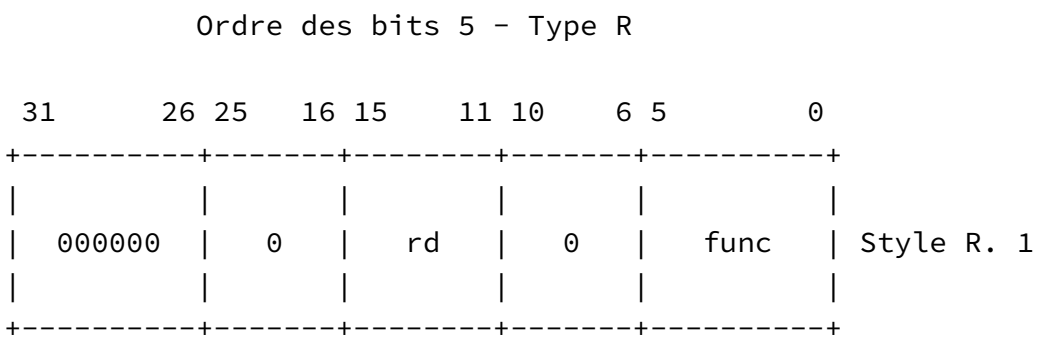
Ordre des bits 4 - Type R



4.0.1.4.1 Style de remplissage 1 : (0/rs/0/hint/func)

- JR : 001000 (25/21 20/11 10/6 5/0)

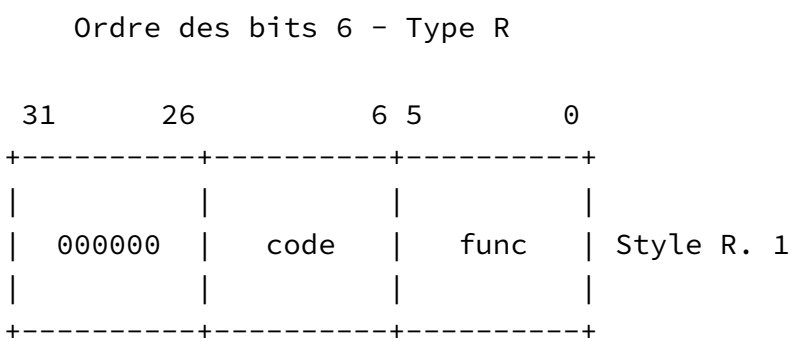
4.0.1.5 Ordre des bits 5



4.0.1.5.1 Style de remplissage 1 : (0/0/rd/0/func)

- MFHI : 010000 (25/16 15/11 10/6 5/0)
- MFLO : 010010 (25/16 15/11 10/6 5/0)

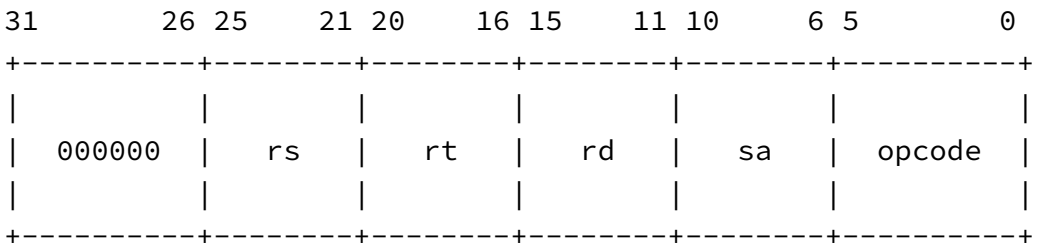
4.0.1.6 Ordre des bits 6



4.0.1.6.1 Style de remplissage 1 : (0/code/func)

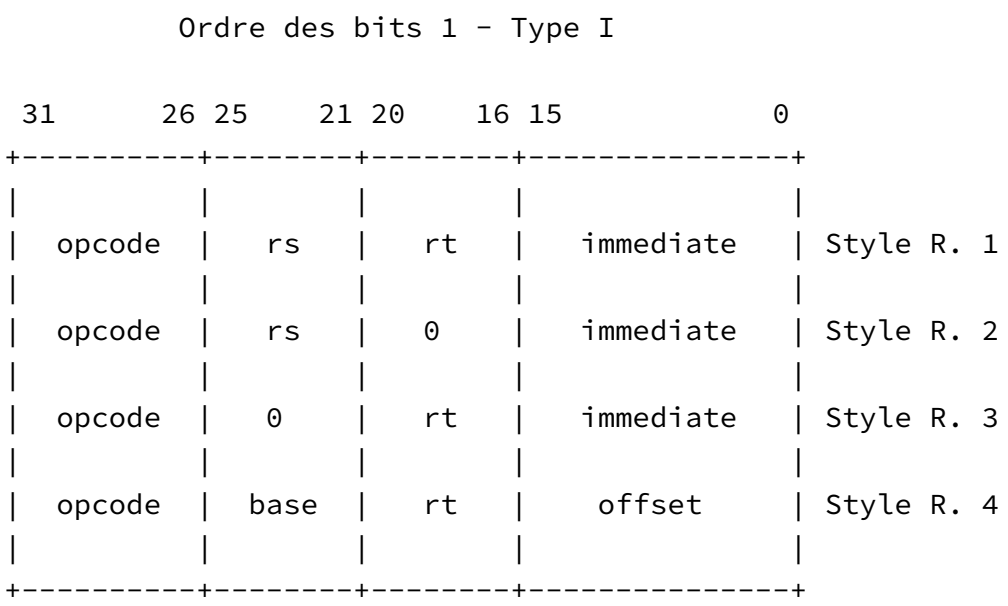
- SYSCALL : 001100 (25/6 5/0)

En résumé on peut dire que toutes ces opérations on la structure suivante en mémoire :



4.0.2 Instruction de type I

4.0.2.1 Ordre des bits 1



4.0.2.1.1 Style de remplissage 1 : (opcode/rs/rt/immediate)

- ADDI : 001000 (25/21 20/16 15/0)
- BEQ : 000100 (25/21 20/16 15/0)
- BNE : 000101 (25/21 20/16 15/0)

4.0.2.1.2 Style de remplissage 2 : (opcode/rs/0/immediate)

- BGTZ : 000111 (25/21 20/16 15/0)
- BLEZ : 000110 (25/21 20/16 15/0)

4.0.2.1.3 Style de remplissage 3 : (opcode/0/rt/immediate)

- LUI : 001111 (25/21 20/16 15/0)

4.0.2.1.4 Style de remplissage 4 : (opcode/base/rt/offset)

- LW : 100011 (25/21 20/16 15/0)
- SW : 101011 (25/21 20/16 15/0)

En résumé on peut dire que toutes ces opérations on la structure suivante en mémoire :

```
 31      26 25    21 20    16 15            0
+----------+--------+--------+---------------+
|          |        |        |               |
|  opcode  |   rs   |   rt   |   immediate   |
|          |        |        |               |
+----------+--------+--------+---------------+
```

4.0.3 Instruction de type J

4.0.3.1 Ordre des bits 1

```
 Ordre des bits 1 - Type J

 31      26 25          0
+----------+-------------+
|          |             |
|  opcode  | instr_index | Style R. 1
|          |             |
+----------+-------------+
```

4.0.3.1.1 Style de remplissage 1 : (opcode/instr_index)

- J : 000010 (25/0)
- JAL : 000011 (25/0)

Il y a donc 16 possibilités en tout.