

## Rapport projet système informatique

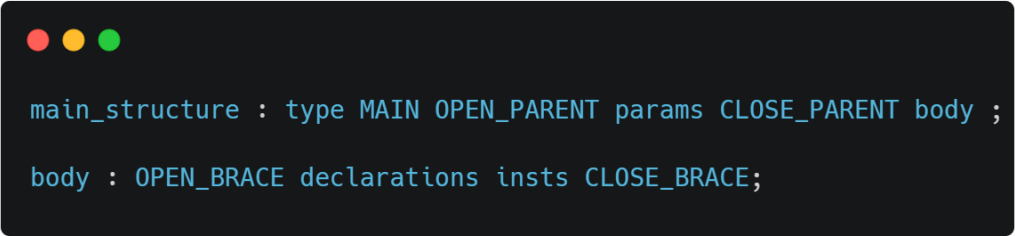
### Démarche de conception

La démarche de conception s'est déroulée en deux parties séparées, la première concernant le compilateur et l'interpréteur et la deuxième concernant le processeur. Pour la partie du compilateur et de l'interpréteur, nous avons suivi les étapes essentielles sans penser aux structures de données pouvant être utiles plus tard car nous ne n'avions pas d'idée sur comment faire le compilateur. Une fois la grammaire finie et le fichier lex modifié plusieurs fois pour s'adapter aux exigences de la grammaire nous avons réfléchi sur comment créer un compilateur qui soit capable de générer un code assembleur en prenant compte de différents registres et de la mémoire pour l'utilisation de variables. Pour les registres nous nous sommes basés sur quelques connaissances sur différentes architecture pour faire un choix sur la manière d'utiliser les registres. Nous avons fait le choix d'écrire la grammaire en premier puis en parallèle implémenter les différentes structures de données nécessaires au fonctionnement du compilateur au fur et à mesure du besoin pour pouvoir s'adapter plus facilement. Au niveau du processeur, nous avons suivi le plan détaillé de la conception de chaque composante interne du processeur et fais des tests progressifs pour assurer un fonctionnement de chaque composante. De cette manière nous pouvions avancer pas à pas sans se soucier du travail fait précédemment.

### Choix d'implémentation

Concernant notre compilateur, nous avons implémenté la reconnaissance de la fonction main, des structures if et while. Mais également la reconnaissance des entiers et des constantes d'entiers, des opérations entre ces derniers et des opérations booléennes.

Pour cela, nous avons défini des règles de grammaire définissant la structure main, en effet elle est composée d'un type renvoyé, du token main, de parenthèses dans lesquelles se trouvent les arguments puis d'un body.



```
main_structure : type MAIN OPEN_PARENT params CLOSE_PARENT body ;  
body : OPEN_BRACE declarations insts CLOSE_BRACE;
```

Par rapport au body, les déclarations de variables se trouvent avant les instructions comme dans un programme C. Le body commence au niveau d'une accolade ouvrante et finit lorsqu'une accolade fermante est trouvée.

Une instruction peut être une affectation d'une valeur à une variable, un print, un bloc si, un bloc tant que ou un retour de résultat. Les instructions sont une série de ces dernières.

```
insts : inst insts | ;
inst : affectation
    | print
    | ifBlock
    | whileBlock
    | RETURN value SEMICOLON {affectation(0,$2, linenumber); linenumber++;}
    | RETURN name SEMICOLON;
```

Les déclarations de variables se font grâce à des ids qui sont une série de noms de variables. Une déclaration doit définir le type de variable. Ce peut être une déclaration simple c'est-à-dire sans affecter de valeur à cette variable soit une déclaration puis une affectation de valeur. Les ids sont uniquement utilisées pour les déclarations. Plus tard pour accéder à ces variables nous utiliserons les règles associées à name.

```
declaration : type ids SEMICOLON | type id EQUAL value SEMICOLON ;
declarations : declaration declarations | declaration;

id : ALPHA ;
ids : id COMMA ids | id ;
```

Ensuite, un bloc if peut être un if seul, soit un if puis des else. Le if est défini par un token if puis la condition entre parenthèses puis un body comme défini précédemment. Puis, le bloc if peut être un if seul, soit un bloc if suivi du token else et d'un autre if donc une structure else if soit d'un else seul.

```

if : IF OPEN_PARENT condition CLOSE_PARENT body;

ifBlock : if
        | if ELSE if
        | if ELSE body;

```

Quant à la structure while, elle est commencée lors d'un token while, suivi d'une condition entre parenthèses puis d'un body. Comme ci-dessous :

```

whileBlock : WHILE OPEN_PARENT condition CLOSE_PARENT body;

```

Les conditions permettent de comparer des valeurs qui sont des valeurs directement et des names qui sont des noms de variables entre eux. Nous avons considéré qu'une condition peut se faire directement avec des valeurs et des variables et vérifier si cette variable est nulle ou non. Puis nous avons les opérateurs classiques comme la négation, les opérateurs comparatifs et l'égalité.

Enfin, les opérations arithmétiques ont une structure qui permet de considérer les priorités des opérations, d'abord les opérations entre parenthèses, puis les multiplication et division, enfin les soustractions et additions. Comme précédemment les opérations arithmétiques peuvent se faire soit avec des valeurs soit avec des noms de variables indistinctement.

Ainsi, notre grammaire permet d'identifier les structures nécessaires dans une fonction main d'un programme en C.

Pour pouvoir transformer l'identification du code vers un code assembleur lisible et un machine, il a fallu d'abord gérer la déclaration des variables faites au cours du code. Pour cela nous avons utilisé une liste contenant le nom de la variable, son adresse, son type et sa profondeur. À cette liste était associée différentes fonctions permettant de la manipuler de manière plus simple et propre dans le fichier yacc. La fonction exists nous a permis de pouvoir traiter les erreurs d'utilisation de variables non déclarées.

```

// Inserts node
void insertFirst(char newName[MAX_LENGTH_NAME], int newAddress, enum t_type newType, int newDepth);
// returns -1 if not found, 1 if the variable is found
int exists(char toGet[MAX_LENGTH_NAME]);
// Deletes and returns a node, returns NULL if it does not exist
Node *delete (char toGet[MAX_LENGTH_NAME]);
// Deletes variables from a given scope
// Argument : Scope to delete
void deleteScope(int scopeToDelete);
// Returns the address of a variable, returns -1 if it does not exist

```

Pour générer ensuite un code assembleur, il a fallu créer des fonctions permettant d'écrire dans un fichier séparé de code assembleur traduit. Cependant avant d'écrire directement dans un fichier il a fallu utiliser une liste pour stocker les instructions lignes par lignes pour modifier les jumps utilisés lors de while ou de if-else. Ensuite une fois tout le fichier parcouru, une fonction écrivant ligne par ligne le tableau dans un fichier txt est appelée pour avoir le résultat désiré. Nous avons pris la décision de donner des rôles spécifiques à des registres pour par exemple accueillir le résultat d'opérations arithmétiques ou encore booléennes.

Ensuite au niveau de l'interpréteur, il a fallu simuler le fonctionnement d'un processeur en implémentant un tableau de 16 integers représentant les registres ainsi qu'une mémoire de taille fixe. De cette manière en parcourant le fichier assembleur généré, il est possible de simuler les mouvements de valeurs dans les registres et la mémoire.

Concernant la partie du processeur, nous avons décidé de ne pas utiliser la sortie QX du banc de registre ainsi que les carries de l'UAL. L'implémentation des différentes parties du processeur ont été faites en accord avec les explications données dans le sujet.

## **Difficultés rencontrées et solutions explorées**

Différents problèmes ont été rencontrés, dans la partie du processeur, un des problèmes principaux et toujours persistant a été la présence de undefined values. Après avoir modifié le fonctionnement interne des buffers du pipeline en utilisant des opérations synchrones pour la réception de valeur et de l'asynchrone pour leur envoi, nous avons pu réduire le nombre de undefined value mais il en reste encore dont nous n'arrivons pas à identifier la source.

Pour ce qui est du compilateur, nous avons des difficultés avec les priorités sur les opérations booléennes ainsi que sur les opérations avec des nombres, nous avons rajouté des règles pour préciser la grammaire ce qui a fortement allongé la longueur du code source. Une des principales difficultés rencontrées a été la correction des jumps pour les boucles whiles et les if-else. En effet, pour que cela marche nous avons dû penser à une solution qui permettrait de gérer des boucles whiles imbriqués ainsi que des sauts conditionnels. La principale solution trouvée était d'utiliser un tableau qui stockait chaque ligne d'assembleur pour ensuite pouvoir y accéder et ensuite la modifier. Nous avons dû cependant changer notre manière de procéder car jusqu'à présent nous écrivions directement dans un fichier txt. Pour gérer l'imbrication de saut, nous avons décidé d'utiliser une pile contenant les différentes lignes pour les différents types de sauts: une pour les sauts de conditions et l'autre pour les sauts sans condition.

```

typedef struct NodeStack
{
    int lineNumber;
    struct NodeStack *next;
} NodeStack;

void add_while(int lineNumber);
void add_JMF(int lineNumber);

int pop_while();
int pop_JMF();

```

Nous avons ajouté les instructions store et load dans la partie du compilateur pour utiliser la mémoire. L'instruction store permettait de faire une sauvegarde de la valeur d'un registre dans une adresse de la mémoire. L'instruction load permettait de charger la valeur contenue à une adresse dans un registre. Ces deux instructions assembleurs ont été ajoutées pour pouvoir utiliser les variables déclarées et conserver leur valeur dans la mémoire dans l'optique de rapprocher le code assembleur de la partie processeur du projet.

Finalement, au niveau des résultats, nous avons pu construire un processeur fonctionnel cependant sans gestion d'aléas. Du côté du compilateur, nous avons obtenu un compilateur capable de générer un code assembleur en utilisant des registres et de la mémoire en partant d'un code proche du langage c.

Voici un exemple du code assembleur généré

AFC 11 5

AFC 12 2

MUL 13 11 12

AFC 11 2

ADD 13 11 13

STR 0 13

LOA 14 0

STR 1 14

LOA 15 0

PRI 15

LOA 15 1

PRI 15

AFC 0 0

Voici le code “c” correspondant:

```
int main() {  
    int i;  
    int j;  
    i = 2 + 5 * 2;  
    j = i;  
    printf(i);  
    printf(j);  
    return 0;  
}
```