

Rapport Projet Système Informatique

Rostom Baccar

June 2022

Contents

1	Partie Logicielle	3
1.1	Démarche conception	3
1.2	Structure du programme et choix d'implémentation	3
1.3	Variables	3
1.3.1	Tableau de symboles	3
1.3.2	Déclaration	3
1.3.3	Affectation	4
1.3.4	Variables locales et profondeur	4
1.4	Instructions Assembleur et tableau d'instructions	4
1.5	Fonctionnalités implémentées	4
1.5.1	If / Else	4
1.5.2	While	4
1.5.3	Fonctions	5
1.6	Interpréteur	5
1.7	Limitations et problèmes rencontrés	5
1.7.1	Blocs imbriqués	5
1.7.2	Appel de fonction	6
1.7.3	Instruction Print	6
1.7.4	Gestion des variables temporaires	6
1.7.5	Fin du programme	6
2	Partie Matérielle	7
2.1	Démarche conception	7
2.2	Microprocesseur	7

2.2.1	Mapping et Débuggage	7
2.2.2	Test des instructions	7
2.2.3	Résultats obtenus	8

1 Partie Logicielle

1.1 Démarche conception

Le premier fichier ajouté est le fichier **Lex.l**. Une fois les tokens définis, le deuxième fichier ajouté est le **Yacc.y** donc la structure sera expliquée dans le paragraphe qui suit. La conception du fichier **Yacc.y** a naturellement fait appel à la création de deux autres fichiers **sTable.c** et **iTable.c** pour la table des symboles et la table d'instructions respectivement. Un autre fichier **pTable.c** a été ajouté plus tard afin de faciliter le passage des paramètres pour les fonctions. Un fichier texte **ASM** est produit à la suite de l'exécution de la commande **make**. Ce fichier contient l'ensemble des instructions générées du code testé. Le fichier est ensuite supprimé en faisant un **make clean**. Enfin, pour automatiser la phase de test, chaque fichier de test a sa propre commande dans le **makefile**. En plus de cela, chaque test a son propre script qui consiste à exécuter les commandes **make**, **clean**, **make clear** et **make -nom-test**.

1.2 Structure du programme et choix d'implémentation

Le programme consiste en un main suivi de fonctions. Le main est obligatoire mais les fonctions autres que le main ne le sont pas. Le main doit aussi obligatoirement se trouver tout au début du programme. Les fonctions déclarées devront donc se retrouver juste après le main. La raison pour une telle organisation sera expliquée dans la partie Fonctions. Le main, ainsi que les fonctions, contiennent un body. Celui-ci peut contenir des instructions. Voici la liste des instructions possibles:

- Instructions de Print
- Déclaration de(s) variable(s) ou de constante(s)
- Affectation d'une variable ou d'une constante
- Déclaration et affectation d'une variable ou d'une constante
- Bloc If / Else
- Bloc While
- Opérations
- Appel de fonction

1.3 Variables

1.3.1 Tableau de symboles

Les variables sont gérées par l'intermédiaire d'un tableau de symboles dynamique. Celui-ci a comme seul but la gestion des cases mémoires pour que les instructions assembleur générées soient cohérentes en évitant notamment d'écraser des valeurs dont on aurait besoin. Il s'agit donc d'un tableau qui gère les variables locales avec un paramètre de profondeur afin de s'en débarrasser à la fin du bloc. Les variables temporaires sont aussi bien gérées par ce tableau puisqu'elles sont dépilées juste après que les instructions assembleur les concernant aient bien été générées. La taille maximale du tableau est fixée à 1000 symboles et la gestion est dynamique car la taille du tableau varie selon le dépilement ou non de certains symboles. De nouveaux symboles peuvent donc prendre la place d'anciens symboles dont nous n'avons plus besoin. Cela nous permet donc d'optimiser l'utilisation des cases mémoire.

1.3.2 Déclaration

Les variables ont un seul type qui est le type entier. La déclaration d'une ou plusieurs variables est possible. La déclaration d'une ou plusieurs constantes est également possible. La différence entre les constantes et les variables est que, une fois affectée, une constante ne peut plus être affectée à nouveau.

1.3.3 Affectation

L'affectation de plusieurs variables ou plusieurs constantes en même temps n'est pas possible. En revanche, pour une seule variable ou constante, une déclaration suivie immédiatement d'une affectation est possible.

1.3.4 Variables locales et profondeur

Les variables locales sont créées lorsqu'on entre dans un bloc if ou un bloc while ou lors de la définition d'une fonction. En effet, une fonction peut très bien utiliser ses propres variables locales. Dans le cas où la même variable existe plusieurs fois dans le tableau de symboles mais avec des profondeurs différentes, la priorité revient à la variable donc la profondeur est la plus importante.

```
void main(){
    int a = 10;
    int b = 20;
    int c = 30;
    a = fun(c);
}

int fun(int a){
    a = a + b;
    return a;
}
```

Dans cet exemple, nous pouvons constater que, à l'intérieur de la fonction **fun**, la variable **a** utilisée est une variable locale à cette fonction tandis que la variable **b** utilisée est la variable globale déclarée et affectée dans le main.

1.4 Instructions Assembleur et tableau d'instructions

Les instructions assembleur sont générées au fur et à mesure de l'analyse du code de test. Elle sont stockée dans une table d'instructions. Elles ont des champs **.function** et **.ret** dont l'utilité sera expliquée dans la partie Fonction. Inspiré par le tableau des symboles, le tableau d'instructions est un tableau qui stocke les instructions assembleur au fur et à mesure de leur génération. Il ne s'agit pas d'un tableau dynamique puisqu'aucune instruction n'est supprimée une fois qu'elle est ajoutée au tableau. Le tableau est limité à 1000 cases et sert à afficher en une fois l'ensemble des instructions générées à la fin de l'analyse du programme de test.

1.5 Fonctionnalités implémentées

1.5.1 If / Else

Les blocs if et else génèrent des instructions de type Jump. Le bloc if génère une instruction **JMF** qui est patchée à la fin du bloc. Le bloc else n'est pas obligatoire. Mais si celui-ci est renseigné, une instruction **JMP** est générée en plus à la fin du bloc if afin de sauter les instructions du else au cas où la condition du if est vraie. La condition qui peut être renseignée entre parenthèses peut être une expression booléenne, une variable ou simplement un **true** ou un **false**. Toutes les variables déclarées dans le body des blocs if et else sont considérées comme des variables locales.

1.5.2 While

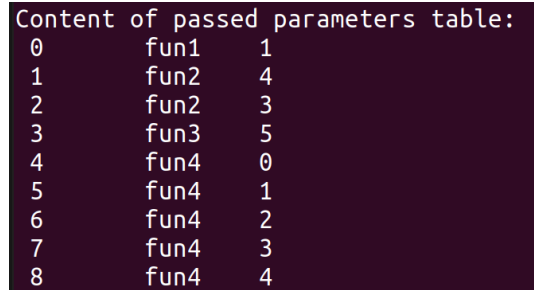
Le bloc while est très similaire à celui du bloc if avec un **JMF** au début et un **JMP** à la fin du bloc à une différence près: dans le but de réévaluer la condition du while, le **JMP** de la fin prend en compte le nombre d'instructions générées entre parenthèses pour être correctement patché. La condition qui peut être renseignée entre parenthèses est identique à celle du bloc if.

1.5.3 Fonctions

Deux types de fonctions ont été implémentées: les fonctions qui ne renvoient rien et dont le type est void, et les fonctions qui renvoient un entier et donc de type entier. Lors d'un appel de fonction depuis de main, une instruction **JMP** est générée pour sauter à la ligne où la fonction est appelée. Cette instruction est donc patchée une fois la définition de la fonction est faite. Ce qui veut dire qu'une fonction définie avant le main devra patcher une instruction non encore générée par le main. C'est pour cela que toutes les définitions de fonctions doivent se trouver après le main.

- Passage de paramètres

Le passage des paramètres pour les fonctions se fait à travers un tableau de paramètres.



```
Content of passed parameters table:
0      fun1      1
1      fun2      4
2      fun2      3
3      fun3      5
4      fun4      0
5      fun4      1
6      fun4      2
7      fun4      3
8      fun4      4
```

La figure ci-dessus nous montre que la fonction **fun1** a passé 3 variables en paramètres, ces variables devant récupérer les valeurs contenues respectivement aux adresses 1, 4 et 3. La correspondance entre les variables passées en paramètres et les variables locales de la fonction est faite à travers un champ **.fonction** qui spécifie le nom de la fonction.

- Valeur de retour

Une fonction qui renvoie un entier a l'obligation d'avoir un retour de variable. L'appel de ce type de fonction se fait dans le main à travers une affectation de variable. Dans ce cas, le main génère une instruction **COP** pour affecter la bonne valeur à la variable, instruction patchée à la fin de la définition de la fonction appelée avec la bonne adresse de retour de la fonction. Afin de patcher la bonne instruction **COP** du main, un champ **.ret** de l'instruction est utilisé pour renseigner le nom de la fonction.

- Instructions **JMP** patchées

A la fin de la définition d'une fonction, celle-ci génère une instruction **JMP** afin de revenir au programme du main. Seulement pour savoir quelle instruction **JMP** du main il faut patcher, un champ **.fonction** de l'instruction est utilisé pour renseigner le nom de la fonction.

1.6 Interpréteur

Un interpréteur a été implémentée afin d'afficher le contenu de chaque registre à la fin de l'exécution du programme de test. Faute de temps, celui-ci interprète seulement les instructions **AFC**, **COP**, **ADD**, **SUB**, **MUL**, **DIV**. L'interpréteur est limité à 16 registres différents.

1.7 Limitations et problèmes rencontrés

1.7.1 Blocs imbriqués

Il n'est pas possible d'imbriquer les blocs de part la nature du patch des différents sauts de ligne. En effet, une correction est possible à ce souci en rajoutant notamment des champs spécifiques ou même une profondeur aux instructions, mais faute de temps cette implémentation n'a pas pu être réalisée. Les blocs concernées sont les If / Else, les While et les fonctions.

1.7.2 Appel de fonction

L'instruction **JMP** générée à la fin de la définition d'une fonction donnée pour revenir au programme du main est une instruction statique, comme toutes les autres instructions. On ne peut donc avoir qu'un seul argument. Ce qui fait que plusieurs appels de la même fonction n'est pas possible. En effet, l'appel plusieurs fois de la même fonction obligerait la fonction à générer un **JMP** dont l'argument changerait à chaque fois.

1.7.3 Instruction Print

L'instruction **Print** ne peut prendre en paramètre qu'une variable ou un nombre. L'impression directe du résultat d'une opération par exemple n'est pas possible. Il faudra stocker le résultat de cette opération dans une variable qu'on imprimerait à la suite. En effet, faute de temps, cette implémentation n'a pas pu être complète.

1.7.4 Gestion des variables temporaires

Les variables temporaires sont très utiles pour stocker des valeurs temporairement dans le tableau de symboles avant de les dépiler juste après la génération des instructions assembleur. Elles sont très utiles notamment pour les opérations puisqu'elles servent de variables auxiliaires. Mais elles ne sont pas utiles pour les affectations simples. En l'état actuel du programme, les variables temporaires sont utilisées pour les affectations simples ce qui n'est pas très optimal en terme de mémoire. Cependant, il est possible que cela soit nécessaire pour le processeur.

1.7.5 Fin du programme

Du fait que les instructions du main sont générées en premier puis les instructions des fonctions en second, le code assembleur final présentera des boucles infinies si on l'interprète comme il est. En effet, puisque toutes les fonctions se terminent par une instruction **JMP**, nous n'allons jamais finir d'analyser le programme. Ce problème a été corrigé en ajoutant à la fin du main une instruction **NOP** qui dirait au parser d'arrêter d'analyser le programme.

2 Partie Matérielle

2.1 Démarche conception

La conception de la partie matérielle a été faite en suivant les consignes du sujet: les entités ont été implémentées une par une et testées une par une (sauf pour la mémoire d'instructions). S'en suit la création de l'entité finale qui est le microprocesseur englobant toutes les précédentes entités. Une fois le chemin de données implémenté, une phase de test du microprocesseur a été entamée.

2.2 Microprocesseur

2.2.1 Mapping et Débuggage

Cette phase est celle qui a pris le plus de temps et qui a permis de corriger un certain nombres d'erreurs dans le code d'origine. En effet, mis à part les erreurs de synthèse qui peuvent être corrigées simplement, la simulation ne montre pas exactement où l'erreur s'est produite. Il faut donc vérifier les entrées et les sorties de chaque pipeline et de chaque entité pour identifier le problème et le corriger au niveau du programme.

2.2.2 Test des instructions

La phase de test a consisté à écrire en dur dans la mémoire d'instructions différentes instructions. Ces différentes instructions modifient le contenu du banc de registre ou bien la mémoire de données (selon l'instruction utilisée). Le test a donc consisté à tester chaque type d'instructions et vérifier que le registre correspondant ou la case mémoire correspondante a été modifiée correctement. Pour une lecture plus aisée, nous demandons au microprocesseur de puiser dans la mémoire d'instructions et de lire une instruction toutes les 100ns.

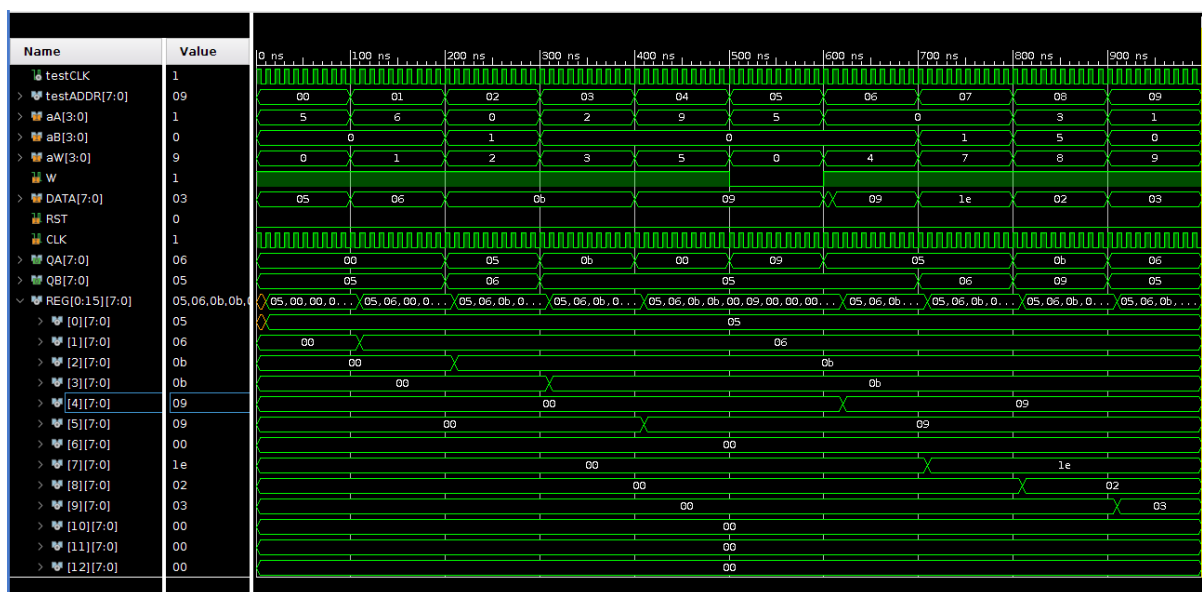
2.2.3 Résultats obtenus

```
--THE INSTRUCTIONS HAVE BEEN INITIALIZED IN THE INSTRUCTION BANK FILE
--WE LOADED THE 8 FIRST SLOTS
--WE WILL NOW SEE IF THE COMPONENTS WORK PROPERLY BY ANALYZING THEIR FIELD CONTENTS

testADDR <= "00000000", --AFC 0 5, REG(0) WILL CONTAIN 5
"00000001" after 100ns, --AFC 1 6, REG(1) WILL CONTAIN 6
"00000010" after 200ns, --ADD 2 0 1, REG(2) WILL CONTAIN 11
"00000011" after 300ns, --COP 3 2, REG(3) WILL CONTAIN 11
"00000100" after 400ns, --AFC 5 9, REG(5) WILL CONTAIN 9
"00000101" after 500ns, --STORE 0 5, MEM(0) WILL CONTAIN 9 !NOTE! WE STORE IN THE MEMORY BANK AND NOT THE REGISTER FILE
"00000110" after 600ns, --LOAD 4 0, REG(4) WILL CONTAIN 9
"00000111" after 700ns, --MUL 7 4 1, REG(7) WILL CONTAIN 9*6=54 AKA 0X36
"00001000" after 800ns, --SOU 8 3 5, REG(8) WILL CONTAIN 11-9=2
"00001001" after 900ns, --DIV 9 1 0, REG(9) WILL CONTAIN 6/2=3 !NOTE! THE LAST ARG IS USELESS, WE JUST DEVIDE BY 2 (LEFT SHIFT ONCE)

end Behavioral;

--TESTS FOR FINAL PART
MEM(0) <= x"06000500", --AFC 0 5, REG(0) WILL CONTAIN 5
MEM(1) <= x"06010600", --AFC 1 6, REG(1) WILL CONTAIN 6
MEM(2) <= x"01020001", --ADD 2 0 1, REG(2) WILL CONTAIN 11
MEM(3) <= x"05030200", --COP 3 2, REG(3) WILL CONTAIN 11
MEM(4) <= x"06050900", --AFC 5 9, REG(5) WILL CONTAIN 9
MEM(5) <= x"08000500", --STORE 0 5, MEM(0) WILL CONTAIN 9 !NOTE! WE STORE IN THE MEMORY BANK AND NOT THE REGISTER FILE
MEM(6) <= x"07040000", --LOAD 4 0, REG(4) WILL CONTAIN 9
MEM(7) <= x"02070001", --MUL 7 0 1, REG(7) WILL CONTAIN 5*6=30 AKA 0X1E
MEM(8) <= x"03080305", --SOU 8 3 5, REG(8) WILL CONTAIN 11-9=2
MEM(9) <= x"04090100", --DIV 9 1 0, REG(9) WILL CONTAIN 6/2=3 !NOTE! THE LAST ARG IS USELESS, WE JUST DEVIDE BY 2 (LEFT SHIFT ONCE)
```



Les figures ci-dessus montrent les différentes instructions utilisées pour la phase le test ainsi que les différents registres et leur contenu en fonction du temps.