Rapport Travaux Pratiques Métaheuristique

Gehin Clement

26 Mai

1 Introduction

Dans le cadre de l'UF "Architecture logicielle et matérielle des systèmes informatiques" nous avons développé aux cours des scéances de travaux pratiques un compilateur pour le langage C ainsi que un microprocesseur pouvant exécuter le code assembleur généré par le compilateur. Ce rapport fait état de nos réalisations des deux parties du projet. Vous pouvez trouver les différents code sources avec dans le même dossier que ce rapport.

2 Compilateur

2.1 Analyseur lexical

Dans un premier temps nous avons réalisé notre analyseur lexical. Le principe est de définir les différents token de notre langage. Lorsque l'analyseur détectera un token lorsqu'il analysera un fichier il nous retournera une valeur de token que l'on pourra utiliser dans notre compilateur.

Il existe différent de type de token. Nous avons les différents mots clés de notre langage. Ces mots clés peuvent être des noms de fonction particuliers (main, printf) des mots logiques ou de déclaration (int, const, if, while) ou encore des opérandes (+,-,*,/). L'autre type de mot clés sont des expressions régulières qui permettent de représenter des noms de variable ou fonction, les nombres ou même les espaces. Si la chaîne de caractère analysée ne correspond à aucun token une erreur sera levée. Nous somme maintenant en mesure de parser notre fichier et de vérifier si une faute de vocabulaire est présente dans ce dernier. Nous pouvons maintenant passer à la définition de la grammaire.

2.2 Analyseur Syntaxique

Le but de l'analyseur syntaxique est de vérifier la grammaire de notre fichier. Pour cela nous devons établir des règles sur l'ordre d'apparition

des tokens. En plus de cela, l'analyseur lexical va nous permettre de réaliser des actions lorsque des règles vont être rencontrées.

Nous avons d'abord définit les différents tokens qui peuvent être retournés par l'analyseur lexical. Nous avons définit des priorité sur certain d'entre eux afin d'éviter les conflits et les ambiguïtés. Nous avons définit une priorité droite pour "=", "¿" et "¡". Cela signifie que lorsqu'une expression contiendra l'un de ces tokens la partie droite sera traité en première. Par exemple, dans "a = b = 2", "b = 2" est d'abord traité puis nous réalisons " a = b". Nous avons mis une priorité à gauche pour "+", "-", "*" ,et "/" ainsi si nous avons " 2 + 2 -+1", nous traiterons " 2+ 2" puis " 4+1". Finalement nous avons également mis des priorités entre tous ces tokens, "*" et "/" sont plus prioritaires que "+" et "-" qui sont plus prioritaires que "=", "¿" et "¡" afin de respecter les conventions de calcul.

Les règles sont définies à partir d'une succession de token ou d'autre règle dans un ordre précis. Ainsi nous avons une première règle générale (File) puis nous descendons au fur et à mesure dans les autres règles. La première difficulté rencontré a été le faite que l'on peut déclarer plusieur variable sur une même ligne. Pour cela nous avons définit la règle suivante : Definition : Prefix DefVar DefVarN tSEMICOLON; . La règle DeVarN est définie comme étant elle aussi une DefVar précédé ou alors vide. Ainsi nous pouvons déclarer plusieur variable en imposant d'en avoir une au minimum.

La deuxième difficultées concernant l'écriture des règles a été lors de l'écriture du if. En effet, un if peut être suivi d'un else ou non. A l'instar de la déclaration nous avons créé une règle Alternative qui peut être vide ou contenir la syntaxe du else. Ainsi nous rendons le else facultatif. L'écriture des règles n'a pas été le point le plus problématique, les difficultés se sont retrouvées dans l'écriture de la table des symboles et des instructions.

2.3 Table des symboles

La table des symboles est un tableau qui contient les différentes variables rencontrées dans le programme. Le tableau est composé ici de 100 lignes. Cette valeur est arbitraire mais elle fixe le nombre de variable maximum que l'on peut sauvegarder donc rencontrer dans notre programme

sur un même niveau de profondeur. Nos fichier test ne contiennent jamais plus de 100 variables, cependant cette valeur peut être facilement ajustée si besoin.

Notre table contient 6 colonnes : nom, adresse, type, constante, initialisé et profondeur. Les adresses ne correspondent à la ligne car nous avons décidé de gérer l'affectation des adresses. Pour cela nous connaissons la dernière adresse ainsi que le type de la variable stockée à cette adresse, nous pouvons donc en déduire l'adresse pour notre nouvelle variable. La profondeur est indispensable car en c nous pouvons avoir deux variables avec le même noms à des profondeurs différentes. Vous pouvez voir sur la Figure 1 un exemple d'une table remplie ainsi que le fichier qui nous a permis de la remplir.

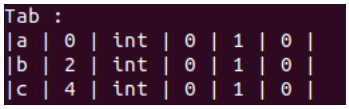

Figure 1: Table des symboles. Colonnes : nom, adresse, type, initialisé, profondeur

La table est initialisée au début de notre fichier dans la règle File. Les lignes sont ajoutés à la table lorsque l'on rencontre une nouvelle variable grâce à la règle DefVar. Il est possible de déclarer une variable sans l'initialiser, une variable est alors toujours ajoutée comme non initialisée. L'initialisation est gérée dans la règle Allocation, où l'on passe la colonne initialisé de la variable correspondante à 1.

Enfin la profondeur actuelle est stockée dans une variable globale. Ainsi on peut connaître à tout moment la profondeur. La profondeur est mise à jour à chaque fois que l'on rencontre des accolades, une accolade ouvrante l'augmente quand une fermante la diminue. La profondeur est alors constamment à jours. Lorsque l'on repasse à une profondeur supérieur nous supprimons toute les lignes qui correspondent à la profondeur inférieur. En effet, les variables étant locale à leur profondeur lorsque l'on remonte vers la profondeur supérieur nous ne devons pas pouvoir accéder au variable de la profondeur inférieur. Nous ne les utiliserons plus, les supprimées permet donc de libérer de l'espace.

2.4 Instructions

Pour la gestion de la traduction en langage assembleur nous avons utilisé deux structures, une table des instructions et une pile pour les données temporaires. La table des instructions est composée de 300 lignes ce qui correspond au nombre maximum d'opération réalisable. Encore une fois ce nombre est arbitraire et peut être facilement changé. La table des instructions à 4 colonnes : code, résultat, valeur 1 et valeur 2. Le code est un entier qui nous permet d'identifier l'instruction. La pile des données temporaires a été fixée à une taille de 100. Il y a qutre grand type d'instruction que nous allons détailler, les instructions d'affectations, les instructions d'opérations, les instructions de comparaison et les instructions de saut.

Pour une affectation nous avons deux cas, soit nous affectons une valeur directement soit nous affectons la valeur d'une autre variable. Dans les deux cas nous allons stocker cette valeur au sommet de la pile, si c'est une simple valeur nous la stockons directement, si c'est une variable nous réalisons une recherche de la valeur par le biais de la table des symboles. Au final la valeur est au sommet de la pile, nous pouvons donc la récupérer dans la pile est l'écrire à l'adresse qui est indiqué dans la table des symboles. Par exemple, si nous avons dans la table des symboles "a" à l'adresse 500 et nous voulons affecter sa valeur à "b" à l'adresse 502. Pour cela, on récupère la valeur de l'adresse 500, nous la plaçons dans la pile, puis nous la récupérons pour la récupérer à l'adresse 502.

Pour les opérations et les comparaisons, nous récupérons la valeur à la gauche de l'opérande et à la droite par le même procédé que pour l'allocation. Nous passons ces deux valeurs au sommet de la pile. Nous réalisons l'opération en récupérant les deux valeurs du dessus de la pile et stockons le résultat au sommet (après avoir supprimé les deux valeurs utilisées pour l'opération). Nous avons donc une valeur au sommet de la pile, il ne reste plus qu'à l'affecter comme décrit précédemment.

Finalement, lorsqu'un if va être rencontré nous allons ajouter l'instruction JumpIF en récupérant la valeur du test au sommet de la pile. nous allons passer tout le Body du if et ajouter les instructions relative à ce dernier. Lorsque nous sortons de son Body nous allons retrouver l'instruction correspondant au if dans notre table d'instruction pour y ajouter l'information

concernant l'adresse de l'instruction courante afin de savoir ou réaliser le saut si le test est faux. Par exemple, imaginons que nous écrivons notre JumpIF à l'adresse 3. Nous parcourons le Body qui lui contient deux instructions. A la fin du body nous somme donc à l'adresse 6, nous allons donc retrouver notre instruction à l'adresse 3 pour lui donner l'information de se rendre directement à l'adresse 6 si le test est faux. Nous utilisons le même principe pour le else mais en utilisant l'instruction Jump.

Le while utilise plus ou moins le même procédé, nous récupérons l'adresse de l'instruction mais avant le test (le test doit être réalisé à chaque itération), nous parcourons le body est ajoutons une instruction de saut avec l'adresse que l'on a stocké. Si nous nous arrêtions là la boucle se réaliserait à l'infini, lorsque nous atteindrons l'instruction de Jump nous retournerons au niveau de l'instruction du test. Pour éviter ce problème nous avons ajouté le même mécanisme que pour le if/else. Nous ajoutons tout d'abord une instruction JumpIF, puis une instruction Jump. L'instruction Jump est écrite pour nous faire sauter au-delà de l'instruction de retour du while coupant ainsi la boucle. l'instruction JumpIF est écrite pour nous faire sauter au-delà de l'instruction qui nous permet d'écrire le Jump nous permettant ainsi d'exécuter le body est l'instruction de retour du while.

Une fois que toute les instructions ont été ajouté nous les traduisons en langage assembleur grâce à la fonction translate(). La Figure 2 et 3 montre un exemple de fichier avec sa traduction assembleur (la mémoire commence à 500).

```
1  int main () {
2      int a = 2;
3      int b;
4      int c = 0;
5
6      if (a = 2){
7          b = 4;
8      }
9      else {
10         b = 8;
11     }
12     while (b > 2) {
13         b = b - 1;
14         c = c + b;
15     }
16 }
```

Figure 2: Fichier C traduit Figure 3

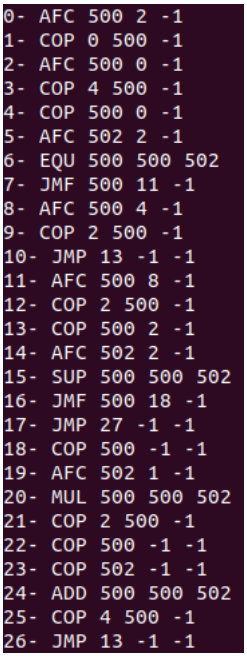

Figure 3: Traduction du fichier C Figure 2

3 Microprocesseur avec pipeline

3.1 Architecture et langage

Dans la deuxième partie de ce projet, nous devions concevoir un microprocesseur de type RISC avec pipe-line. Celui-ci a été implémenté en VHDL et permet d'utiliser le langage assembleur orienté registre suivant :

Par manque de temps sur la seconde partie, nous n'avons pas pu implémenter les opérations de Division (0x04) et de Sauvegarde (0x08). Les autres opérations ont été implémentées et testées.

Notre microprocesseur possède une architecture avec un pipeline à 5 étages. Chaque étage de ce microprocesseur contient différents composants, chacun implémenté et testé indépendamment avant l'ajout au pipeline.

Opération	Code	Format d'instruction				Description
		OP	A	B	C	
Addition	0x01	ADD	Ri	Rj	Rk	[Ri] ← [Rj] + [Rk]
Multiplication	0x02	MUL	Ri	Rj	Rk	[Ri] ← [Rj] * [Rk]
Soustraction	0x03	SOU	Ri	Rj	Rk	[Ri] ← [Rj] - [Rk]
Copie	0x05	COP	Ri	Rj	_	[Ri] ← [Rj]
Affectation	0x06	AFC	Ri	j	_	[Ri] ← j
Chargement	0x07	LOAD	Ri	@j	_	[Ri] ← [@j]

Figure 4: Description des opérations assembleur

3.2 Unité arithmétique et logique

Ce composant permet de réaliser les opérations d'Addition, de Multiplication et de Soustraction. Il reçoit en entrée les valeurs de A, B et OP de l'instruction à réaliser. Le résultat de l'opération correspond à la sortie de l'UAL avec différents flags. Bien que ces flags soient gérés au sein de l'UAL, ils ne seront pas utilisés dans la suite du projet.

Au niveau de l'implémentation, l'UAL est un composant combinatoire, ou asynchrone, donc le processus réalisant l'opération ne prend pas en compte l'horloge. Et l'opération à réaliser est déterminée grâce à un bloc case sur le code d'opération. Les signaux d'entrée de l'UAL sont sur 8 bits et la sortie doit elle est aussi être sur 8 bit. Cependant les opérations réalisées peuvent donner des résultats plus grands. L'addition et la soustraction peuvent générer un nombre sur 9 bit et la multiplication un nombre sur 16 bit. Ainsi, nous avons créé un signal auxiliaire S_16 correspondant au résultat de l'opération sur 16 bits. Nous avons ensuite en extraire les 8 bits qui nous intéressent. Voici comme exemple, l'implémentation de l'addition :

Si le code d'opération en entrée de l'UAL ne correspond pas à une des trois opérations, le composant renvoie par défaut des signaux à zéro.

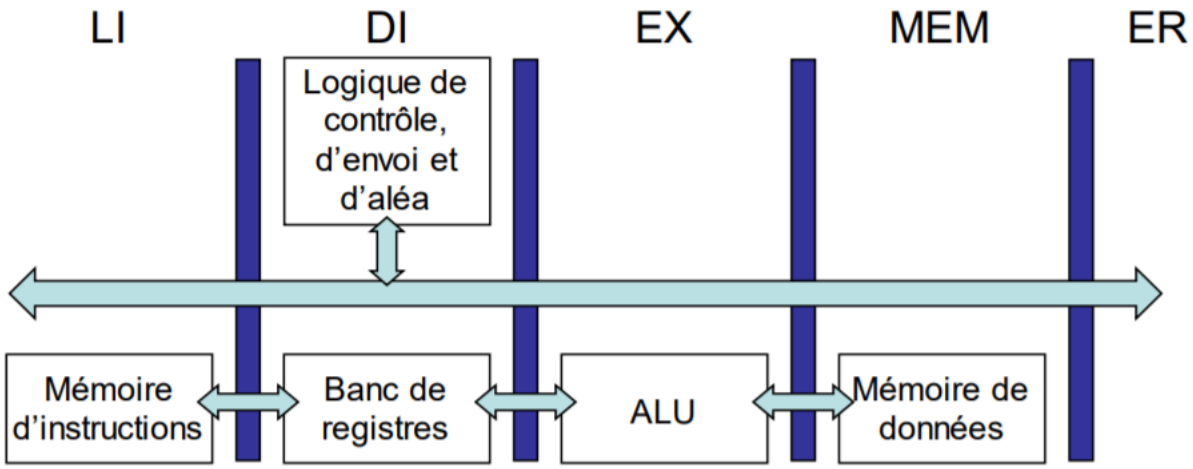

Figure 5: Etages du pipeline

3.3 Banc de registre

Le langage assembleur que nous devions utiliser est un langage orienté registre. Pour cela, nous avons créé 16 registres 8 bits pouvant être utilisé en lecture comme en écriture. Notre banc de registre est capable de sortir le contenu de 2 registres simultanément en mode lecture. Les adresses de ces registres sont donnés en entrée du composant. En écriture, nous écrivons le contenu du signal d'entrée DATA dans un registre.

Le reset et l'écriture dans un registre sont réalisés de façon synchrone avec l'horloge.

Pour avoir les 16 registres nécessaires, nous avons créé un tableau de STD_LOGIC_VECTOR de taille 16. Nous pouvons accéder à ce tableau en utilisant un signal interne. Lorsque l'on souhaite accéder en lecture ou écriture à un registre, nous devons convertir nos valeurs d'entrées en integer grâce à la librairie $IEEE.NUMERIC\_STD$.

```
--addition
when "001" =>
    S_16 <= ("00000000" & A) + ("00000000" & B);
    S <= S_16 (7 downto 0);
    C <= S_16(8);
    N <= S_16(7);
    if A = "00000000" AND B = "00000000" then
        Z <= '1';
    else
        z <= '0';
    end if;
    if A(7) = B(7) AND B(7) /= S_16(7) then
        O <= '1';
    else
        O <= '0';
    end if;
```

Figure 6: Code VHDL pour l'addition dans l'ALU

3.4 Banc de mémoire

Pour notre microprocesseur, nous avons implémenté deux bancs supplémentaires utilisés pour la mémoire. L'un d'entre eux correspondra à une mémoire ROM dans laquelle nous écrirons la liste des instructions que notre microprocesseur devra réaliser. Le second servira de mémoire RAM. Nous pourrons utiliser les opérations LOAD et STORE (non implémentée pour le moment) afin d'accéder en lecture ou en écriture à cette mémoire. Elle contiendra donc les variables temporaires utiles pour le programme traduit en assembleur.

Comme pour le banc de registres, les banc de mémoires sont implémentés en utilisant des tableaux de STD_LOGIC_VECTOR mais avec une taille de 256 (les adresses seront en 8 bits). La principale différence entre les mémoires RAM et ROM est la prise en compte du signal de reset. Celui ci est ignoré par la mémoire ROM pour ne pas perdre son contenu.

Pour initialiser la mémoire ROM, nous utilisons un fichier init.exe contenant sur chaque ligne une instruction écrite en hexadécimal.

3.5 Chemin de données

Une fois tous les composants implémentés et testés indépendamment, il faut maintenant les relier entre eux. Pour cela, nous avons 5 étages de pipeline. Entre 2 pipeline, nous propageons 3 à 4 signaux chacun sur 8 bits. Le premier signal A correspond pour chacun des opérations à une adresse. L'opération STORE n'étant pas implémentée, ce signal contient donc toujours l'adresse d'un registre. A chaque étage du pipeline, ce signal est propagée jusqu'au banc de registre.

Les autres signaux OP, B et C vont eux être utilisés par les différents composants positionnés dans les étages du pipeline. Pour pouvoir transmettre les différents signaux entre les étages du pipeline, nous avons créé 4 signaux auxiliaires pour chacun d'entre eux comme le montre la Figure 7 . Ces signaux sont utilisés ensuite lors de l'attribution des ports de chaque composant. La figure 8 montre un exemple pour l'UAL.

```
signal A_LI_DI : STD_LOGIC_VECTOR (7 downto 0);
signal OP_LI_DI : STD_LOGIC_VECTOR (7 downto 0);
signal B_LI_DI : STD_LOGIC_VECTOR (7 downto 0);
signal C_LI_DI : STD_LOGIC_VECTOR (7 downto 0);
```

Figure 7: Déclaration de 4 signaux auxiliaires

```
alu_instance : entity WORK.alu
port map (
    A => B_DI_EX,
    B => C_DI_EX,
    Ctrl_Alu => LC_ALU,
    S => S_OUT);
```

Figure 8: Instanciation de l'ALU

Pour chaque étage, nous avons créé des processus synchrone permettant la propagation des signaux inchangés s'il n'y a pas de reset. Cela concerne donc principalement les signaux A et OP

Les signaux B et C en revanche sont utilisés pour les composants et

```
EX_MEM : process (CLK, OP_DI_EX, A_DI_EX, B_DI_EX,RST)
begin
    if RST = '0' then
        if rising_edge(CLK) then
            OP_EX_MEM <= OP_DI_EX;
            A_EX_MEM <= A_DI_EX;
        end if;
    else
        OP_EX_MEM <= "00000000";
        A_EX_MEM <= "00000000";
    end if;
end process;
```

Figure 9: Processus d'un étage du pipeline

la/les sorties de ceux-ci seront utilisés pour l'étage suivant. Cela est possible grâce à une bonne attribution des ports de chaque composant.

Afin de lire les instructions une par une, nous avons ajouté un process IP qui incrémente un signal auxiliaire de 1 à chaque front montant d'horloge. Ce signal envoyé au banc de mémoire ROM est l'adresse de l'instruction dans le tableau créé précédemment.

Enfin, comme les opérations ont des nombres et types d'arguments différents, nous avons ajouté des composants synchrones dans notre chemin de données. Les process MUX en fin d'étage permettent de propager suivant le code d'opération la sortie du composant de l'étage ou le signal propagé par l'étage précédent. Nous avons aussi ajouté des process LC asynchrones qui traduisent le code d'opération en entrée en un signal utilisé par le composant de l'étage. Par exemple, l'un d'entre eux permet de déterminer le bit de lecture/écriture.

4 Conclusion

A la fin de ce projet, nous avons implémenté un compilateur pour un langage C simplifié ainsi qu'un processeur exécutant un code assembleur

orienté registre. Ce projet nous a permis de renforcer nos connaissances en architecture matérielle. Cependant, il reste encore quelques améliorations à faire : implémentation des instructions DIV et STORE, test du processeur sur la carte FPGA, gestion des aléas...