INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Université Fédérale Toulouse Midi-Pyrénées

Département de Génie Électrique & Informatique

Rapport du projet Systèmes Informatiques

Du compilateur vers le microprocesseur

Quentin Mouret Richard Nedu

4IC1

26 Mai 2020

Table des matières

Introduction

Le projet a deux objectifs. Le premier est de développer un compilateur pouvant traduire une version simplifiée du langage C en un langage assembleur. Le second est de concevoir et d'implémenter un microprocesseur avec pipe-lines pouvant traiter une partie des instructions assembleurs précédemment obtenues. Pour réaliser le compilateur, nous avons d'abord généré via l'outil LEX l'analyseur lexicographique de notre langage C à partir du lexique de notre programme. Puis, grâce à la syntaxe de notre langage C, nous avons généré via l'outil YACC l'analyseur syntaxique associé. Chaque instruction détectée par l'analyseur syntaxique sera alors traduite en langage assembleur. La partie processeur est constituée de l'implémentation de 4 composants en VHDL sur Xilinx, puis de leur inclusion dans un seul composant qu'est le processeur.

1 Réalisation du compilateur

1.1 Description du langage assembleur

Le langage reconnu par le compilateur est une version simplifiée du langage C. Celui-ci contient la fonction main(), la fonction printf(), les constantes de type entier const et leur noms, les variables de type entier int et leur nom, les opérateurs arithmétiques de base (addition, soustraction, multiplication, division, égalité), les séparateurs (espace, TAB, virgule), le saut de ligne \n ainsi que la ponctuation habituelle du langage C (parenthèses, accolades, points-virgules).
L'ensemble de ces symboles est le lexique de notre langage C. Nous l'avons spécifié à l'aide d'expressions régulières.

1.2 Génération de l'analyseur lexicographique

Les symboles reconnus par contre langage C ont été spécifiés grâce à des expression régulières. Par exemple, le nom d'une constante ou d'une variable doit commencer par une lettre et peut contenir des lettres, des chiffres et le underscore _ . Ainsi, nous avons écrit dans notre programme que le nom d'une constante ou variable est reconnue si l'expression suivante est satisfaite : [a-z][a-z0-9_]*

```
[a-z][a-z0-9_]* { strcpy(yylval.var, yytext);return tVAR;}
```

Figure 1 : Expression régulière pour le nom d'une variable

Une fois l'ensemble des tokens (symboles) décrit, nous l'avons fourni comme entrée du générateur d'analyseur lexical LEX. Pour vérifier que tous les symboles sont bien reconnus, nous avons demandé à LEX d'afficher à l'écran le nom d'un symbole lorsque celui-ci est détecté. Puis nous avons réalisé une série de tests sur différents programmes C. Ci-dessous, se trouve le programme C d'un test que nous avons réalisé :

```
main()
{
    const l;
    int i;
    int j,k;
    int r=0;
    i =3;
    j=4;
    k=8;
    l=9;
    printf (i);
    r=(i+j)*(i+k/j);
    printf ( r );
    if(r==2){
        printf(i);
    } else {
        k = 2;
    }
    if(1){
       i = i+1;
    }
}
```

Figure 2 : Code C utilisé pour les tests

Ainsi que le résultat retourné par l'analyseur lexical :

```
tMAIN tOP tCP tOB tCONST tVAR tSC tINT tVAR tSC tINT tVAR tCOMA tVAR tSC tIN
T tVAR tEQU tNUMBER tSC tVAR tEQU tNUMBER tSC tVAR tEQU tNUMBER tSC tVAR tEQ
U tNUMBER tSC tVAR tEQU tNUMBER tSC tPRINTF tOP tVAR tCP tSC tVAR tEQU tOP t
VAR tADD tVAR tCP tMUL tOP tVAR tADD tVAR tDIV tVAR tCP tSC tPRINTF tOP tVAR
 tCP tSC tIF tOP tVAR tEQU tEQU tNUMBER tCP tOB tPRINTF tOP tVAR tCP tSC tCB
 tELSE tOB tVAR tEQU tNUMBER tSC tCB tIF tOP tNUMBER tCP tOB tVAR tEQU tVAR
tADD tNUMBER tSC tCB tCB
```

Figure 3 : Résultats obtenus avec l'analyse lexical

1.3 Génération de l'analyseur syntaxique

L'étape suivante fut de décrire la syntaxe de notre langage C. Celle-ci est spécifiée sous la forme de règles respectant le format du générateur d'analyseur syntaxique YACC. Pour être le plus clair possible, nous avons adopté l'approche en entonnoir. Ainsi, une expression très générale tel que le corps de la fonction main() est composée de seulement deux expressions : les déclarations et les instructions.

```
BODY : DECLARATIONS INSTRUCTIONS ;
```

Figure 4 : Exemple d'une ligne de l'analyseur syntaxique, reconnaissant le Body d'un programme C

1.4 Traduction en langage assembleur

Notre langage assembleur doit pouvoir reconnaître un ensemble d'opérations telles que l'addition, la multiplication, la soustraction, la division ou l'affectation. Pour y parvenir, il est nécessaire que notre analyseur syntaxique reconnaisse les opérandes des instructions. C'est pourquoi nous avons créé une table stockant les identifiants des différents symboles reconnus par YACC, bien entendu, les identifiants doivent être les mêmes que ceux reconnu lors de l'analyse lexicale. Si un opérande est une variable ou une constante précédemment reconnu par LEX, alors son identifiant sera présent dans la table des symboles. Dans le cas contraire, une variable temporaire sera créée et stockée à la fin du tableau le temps de réaliser l'opération puis supprimée après.

Les opérations de notre langage assembleur ont ensuite été implémentées. Elles prennent en paramètre les symboles précédemment créés. Il s'agit de l'addition, la soustraction, la multiplication, la division, l'affectation, la copie et l'affichage à l'écran. Dès lors que l'une de ces instructions est détectées par notre analyseur syntaxique, une traduction en assembleur est réalisée et affichée à l'écran.

1.5 Prise en compte des instructions conditionnelles if et if-else

La dernière étape de la réalisation du compilateur fut de prendre en comptes les instructions if et if-else. Pour cela, nous avons implémenté les deux sauts JMP (saut inconditionnel) et JMPF (saut si la condition du if n'est pas satisfaite). Pour plus de clarté, une instruction CMP (comparaison) avant l'instruction JMPF fut aussi implémentée. La grande difficulté de ces instructions conditionnelles est de déterminer les lignes où l'on doit sauter. En effet, la traduction en assembleur du corps d'une instruction if ou else est réalisée après l'instruction du saut correspondant. C'est pourquoi, nous laissons les numéros de lignes vierges lors de la traduction en assembleur. Ceux-ci seront remplis une fois que toutes les instructions seront réalisées, il suffira de trouver les lignes où les instructions if et else se terminent. Pour délimiter le début et la fin des instructions if et else, des pseudo-instructions END IF sont créés pendant la traduction en assembleur et seront supprimées une fois les numéros de lignes trouvéEs.

Ci-dessous se trouve le résultat obtenu par le compilateur pour le programme C présenté précédemment :

```
L0 : AFC 2 256
L1 : AFC 5 255
L2 : AFC 3 254
L3 : AFC 1 253
L4 : PRINT 2
L5 : ADD 252 2 5
L6 : DIV 250 3 5
L7 : ADD 249 2 250
L8 : MUL 247 251 248
L9 : AFC 4 247
L10 : PRINT 4
L11 : CMP 4 246
L12 : JMPF L15
L13 : PRINT 2
L14 : JMP L16
L15 : AFC 3 245
L16 : END CMP
L17 : JMPF L20
L18 : ADD 242 2 243
L19 : AFC 2 242
L20 : END CMP
```

Figure 5 : Compilation du programme C

2 Réalisation du processeur

Le système que nous allons implémenter est un microprocesseur à 5 étages de pipe-line. Les 5 étages sont les suivants :

- LI, avec la mémoire d'instructions (*instructs memory IM)*
- DI, avec le banc de registres *(register files RF)*
- EX, avec l'unité arithmétique et logique (*arithmetic-logic unit ALU)*
- MEM, avec la mémoire de données *(data memory DM)*
- RE

Une fois que les différents éléments seront créés et testés, nous les lierons dans un composant processeur, afin de pouvoir réaliser les instructions assembleur affectation (AFC), copie (CPY), opérateurs arithmétiques addition (ADD), soustraction (SUB) et multiplication (MUL), ainsi que deux autres instructions que sont le chargement (LOAD) et la sauvegarder (STORE).

Nous allons à présents présenter les choix d'implémentations pour chaque composant, ainsi qu'une simulation montrant leur bon fonctionnement. Nous n'entrerons pas dans les détails de l'implémentation, mais pointerons les spécificités nous semblant importantes.

2.1 Mémoire d'instruction

La mémoire d'instruction est initialisée à sa création, afin de pouvoir y implémenter les différentes instructions que notre processeur va devoir réaliser.
Nous avons ajouté une instruction "pause", représentée par la ligne :

```
constant pause : std_logic_vector(31 downto 0) := x"00000000";
```

Figure 6 : Signal pause

Cette dernière permet d'attendre un top d'horloge dans qu'aucune nouvelle instruction ne soit prise en compte. C'est utile notamment dans une séquence AFC - CPY, où le CPY veut copier l'information affectée par AFC. Elle doit dans ce cas attendre que le banc de registres soit mis à jour.

Le fonctionnement de ce composant étant similaire à celui de la mémoire de données, nous n'avons pas fait de simulation. Son comportement sera expliqué au travers de la simulation de la mémoire de données.

2.2 Mémoire de données

La spécificité de la mémoire des données est qu'elle est synchronisée sur un front descendant d'horloge, afin de ne pas rajouter un top d'horloge dans le fonctionnement du processeur, et risquer un décalage des informations.

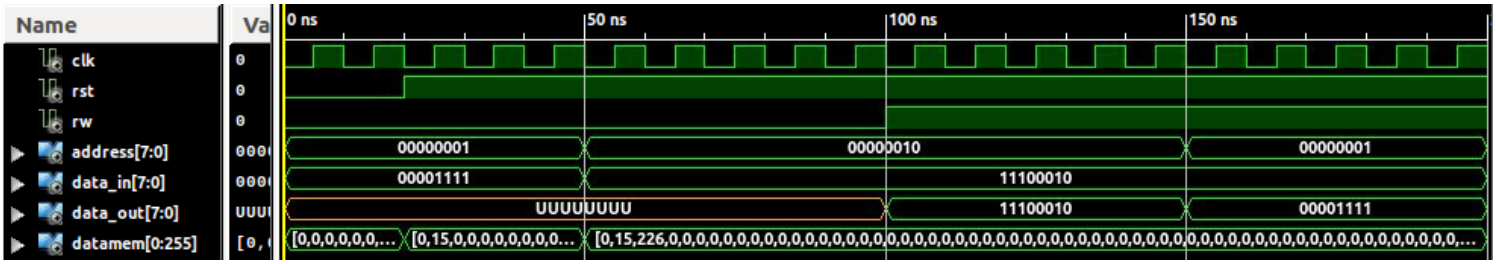

Figure 7 : Simulation de la mémoire de données

Cette simulation est découpée en 4 parties :

- 0-50 ns, après RST qui met toute la mémoire de donnée (*datamem*) à 0, nous écrivons "00001111" à l'adresse "00000001" (1).
- 50-100 ns, nous écrivons "11100010" à l'adresse "00000010" (2).
- 100-150 ns nous lisons que "11100010" à bien été mis à l'adresse (2), sur *data_out.*
- 150-200 ns nous lisons que "00001111" à bien été mis à l'adresse (1) sur *data_out.*

2.3 Banc de registres

Nous avons choisi d'implémenter le banc de registre comme un tableau de 16 std_logic_vector(7 dt 0), les adresses en entrée du composant étant sur 4 bits.

La prochaine simulation fonctionne en 3 étapes :

- 0-50 ns, après le RST (mise à 0 de registers), écriture de la DATA à l'adresse "0001", et lecture en même temps sur Qa. Nous avons fait cela afin d'assurer que si le processeur fait à la fois une requête d'écriture et de lecture au même endroit, il n'y ai pas de conflit. Dans ce cas Qa <= DATA
- 50-100 ns, nous écrivons la DATA à l'adresse "0010"
- 100-150 ns, nous lisons sur Qb la DATA à l'adresse "0010"

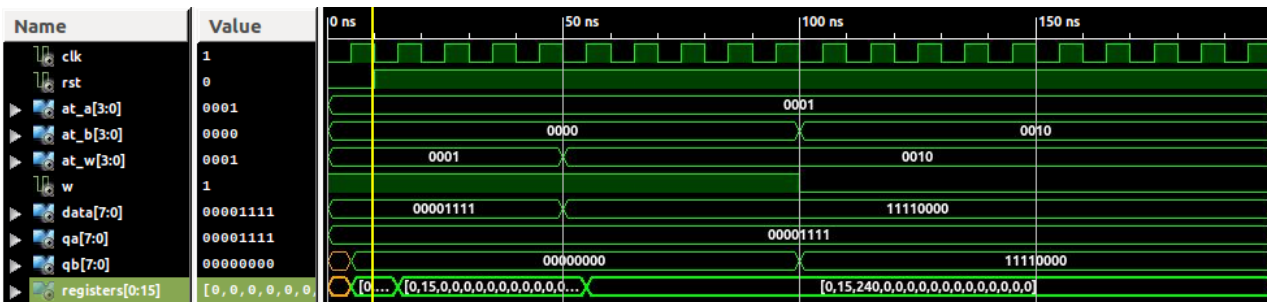

Figure 8 : Simulation du banc de registres

2.4 Unité arithmétique et logique

Nous réalisons des opérations sur 8 bits. Afin de détecter un potentiel overflow/carryflag ou valeur négative, nous passons les valeurs d'entrée sur 16 bits. Nous pouvons ainsi effectuer les opérations et détecter l'une des erreurs décrites, puis ensuite renvoyer en sortie les 8 bits de poids faible.

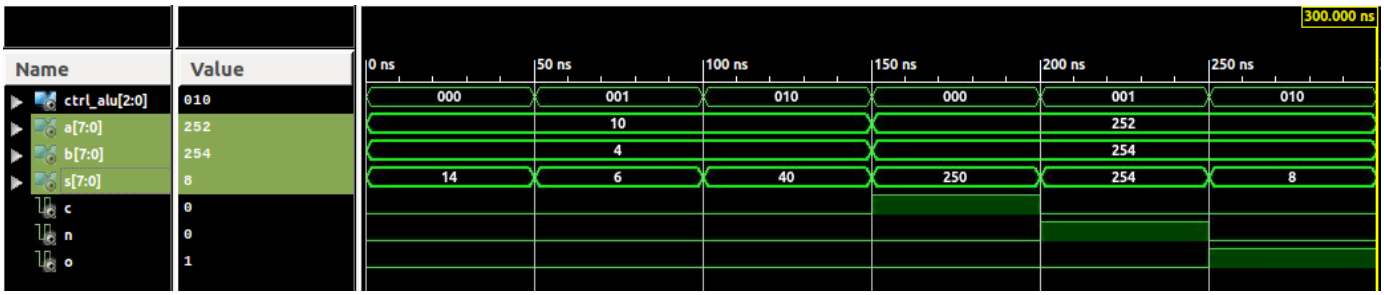

Figure 9 : Simulation de l'ALU

Cette simulation s'organise en deux cycles ADD-SUB-MUL, tout d'abord avec des valeurs de tailles raisonnables. Nous voyons des résultats corrects : 10+4=14 ; 10-4=6 ; 10*4=40.

Ensuite, nous mettons des valeurs déraisonnablement grandes, de façon à ce que les opérations lèvent les flags associés.

2.5 Processeur

Une fois tous les composants créés, nous les avions liés dans un nouveau composant appelé processeur. Afin que ces derniers puissent communiquer, en accord avec le schéma des chemins de données proposé, nous avons créé 5 pipelines comme des tableaux, comme présenté ci-dessous :

```
type pipeline is array(0 to 3) of std_logic_vector(7 downto 0);
signal LIDI, DIEX, EXMEM, MEMRE : pipeline ;
constant A : integer := 0; --registre de destination
constant OP : integer := 1; --code operation
constant B : integer := 2; --registre entree 1
constant C : integer := 3; -- registre entree 2
```

Figure 10 : Déclaration des Pipelines

Nous avons également implémenté un grand nombre de signaux, un pour chaque entrée/sortie de composant. Ces signaux sont mis à jour en permanence en dehors du process() de notre processeur, afin d'assurer une bonne synchronisation.

```
--signaux RegisterFiles
signal sig_RF_atA, sig_RF_atB, sig_RF_atW : std_logic_vector(3 downto 0);
signal sig_RF_W : std_logic;
signal sig_RF_DATA, sig_RF_QA, sig_RF_QB: std_logic_vector(7 downto 0);

--signaux ALU
signal sig_ALU_A, sig_ALU_B, sig_ALU_S : std_logic_vector(7 downto 0);
signal sig_ALU_CTRL : std_logic_vector(2 downto 0);

--signaux DATAMEM
signal sig_DM_AT, sig_DM_IN, sig_DM_OUT : std_logic_vector(7 downto 0);
signal sig_DM_RW : std_logic;

--signaux IM
signal sig_IM_instruction : std_logic_vector(31 downto 0);
signal IP : std_logic_vector(7 downto 0):= "00000000";
```

Figure 11 : Signaux implémentés

La prochaine simulation est l'affichage du banc de registres et de la mémoire de données après les instructions suivantes :

1) AFC R0 3, soit affectation au registre 0 de la valeur 3
2) CPY R1 R0, soit copie de la valeur du registre 0 dans le registre 1
3) ADD R2 R0 R1, soit somme dans le registre 2 de R0 et R1, soit 6
4) MUL R3 R0 R1, soit multiplication de R0 et R2 dans R3, soit 18
5) SUB R4 R3 R2, soit soustraction de R3 par R2 dans R4, soit 12 et dans le même temps STORE 0 R2, soit store à l'adresse 0 dans la mémoire de données la valeur de R2
6) LOAD R6 0, soit load la valeur dans la mémoire de donnée et la mettre dans R6

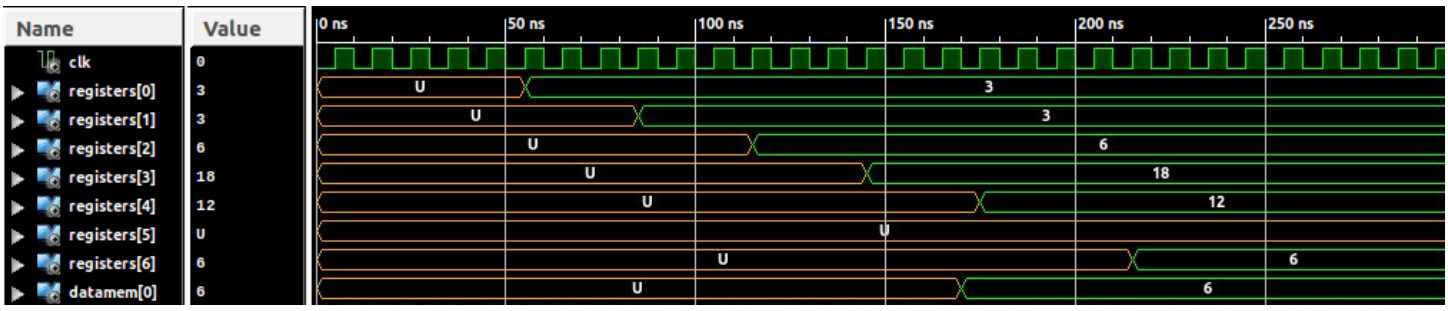

Figure 12 : Simulation finale du processeur

Nous avons volontairement montré seulement les résultats, et non l'évolution de tous les signaux et registres car la simulation aurait été trop dense.

Conclusion

Le projet nous a permis de réaliser un système informatique complet. Un compilateur traduisant une version simplifiée du langage C en un langage assembleur fut développé. Celui-ci prenait initialement en charge les opérations arithmétiques de base, l'affectation, la copie et l'affichage à l'écran. Le respect de l'unicité des tokens dans la table des symboles fut problématique. Nous avons dû veiller à ne pas réintroduire le même token dans la table des symboles lorsque celui-ci est déjà présent, en choisissant quelles instructions donner la priorité pour introduire un token dans la table. Nous avons ensuite amélioré prenant en charge les instructions conditionnelles if et if-else. Pour gérer les instructions de saut, nous déterminons après la traduction assembleur où sauter grâce à des pseudo-instructions délimitant la fin des blocs if et else.

Pour ce qui est de la partie processeur, le système complet a été implémenté. La création des composants a été une chose plutôt simple et rapide, mais la partie processeur a posé beaucoup de problèmes. Nous avons eu de nombreux problèmes de synchronisation, qui n'ont été réparables qu'après de longs et fastidieux moments de recherches. Un code aussi long et complexe se doit d'être écrit de manière rigoureuse. Par ailleurs, nous avons eu un long blocage à cause de l'ALU, qui fonctionnait seule en simulation, mais qui ne fonctionnait plus inclue dans le processeur. Mais après une analyse rigoureuse du code, nous avons corrigé nos erreurs et rendons un processeur fonctionnel.

Nous avons également rencontré un autre problème : le retard. Nous avons pris énormément de retard dans la partie compilateur. Au début des séances de processeur, notre table des symboles n'était absolument pas fonctionnelle. Ainsi nous avons pris la décision de séparer nos travaux, Richard travaillant à rattraper ce retard puis plus tard rejoignant Quentin qui avait travaillé sur la partie processeur afin de ne pas prendre de retard dans cette partie également. C'est sans aucun doute grâce à cette répartition des tâches que nous avons pu finir dans les temps.

Nous n'avons pas implémenté de partie optionnelle, la finalisation du code nous ayant amené au niveau de la deadline.