INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Université de Toulouse

Projet Système informatique

Compilateur/Processeur

SELLAMI Kais - MARTIN Cédric

Institut National des Sciences Appliquées de Toulouse

18 mai 2025

Table des matières

Chapitre 1

Introduction

Dans le cadre de ce projet, nous avons conçu un compilateur simplifié du langage C en utilisant les outils `YACC` et `LEX`. Ce compilateur est capable de reconnaître et traiter les concepts fondamentaux de la programmation tels que les opérations arithmétiques, les structures conditionnelles, les fonctions et les boucles.

Dans un second temps, nous avons développé un processeur basé sur l'architecture `RISC-V`, capable d'exécuter les instructions générées par notre compilateur. Ce processeur a été implémenté en `VHDL` et déployé sur une carte `FPGA`, illustrant ainsi l'intégration complète entre un compilateur logiciel et une architecture matérielle.

Chapitre 2

Compilateur C Simplifié

2.1 Démarche de conception

Le projet a débuté par la définition des fonctionnalités attendues :
— Déclarations de variables (entiers uniquement) ;
— Affectations simples et opérations arithmétiques ;
— Structures conditionnelles `if`/`else` ;
— Boucles `while` et `for` ;
— Fonctions avec retour de valeur.

2.2 Choix d'implémentation

Nous avons utilisé `Flex` et `Bison`. Le compilateur génère du code assembleur compatible avec notre processeur RISC-V.

2.2.1 Analyse lexicale

Extrait du fichier `Analyseur_Lex.l` :

```
D [0-9]
C [A-Za-z]
%%
printf              return tPRINTF;
main                return tMAIN;
while               return tWHILE;
if                  return tIF;
else                return tELSE;
{D}+                {yylval.nb = atoi(yytext); return tNB ;}
{C}({C}|{D}|"_")*   {strcpy(yylval.id, yytext); return tID;}
```

2.2.2 Analyse syntaxique

Extrait Bison pour les instructions conditionnelles :

```
If: tIF tOP Operation tCP Body OptElse ;
OptElse :
           tELSE Body
         | %prec tWoElse;
```


2.3 Problèmes rencontrés et solutions

Structures conditionnelles

Problème : le saut inutile en l'absence de `else`. Solution : insérer un `NOP` pour conserver la cohérence du code.

Boucles `for`

Problème : l'ordre des instructions diffère entre le C et l'assembleur. Solution : restructuration en utilisant des sauts conditionnels. Exemple :

```c
int main () {
    int a  = 7;
    int i;
    for ( i=0; i < 10; i = i+1){
        a = a + 1;
    }
}
```

```
0 : 9 1 0 0  // jump 1 => jump to main
1 : 6 1 7 0  // affect 7 to @1
2 : 5 0 1 0  // copy @1 to @0
3 : 6 2 0 0  // affect 0 to @ 2
4 : 5 1 2 0  // copy @2 to @1
5 : 5 2 1 0  // copy @1 to @2 => temporary variable
6 : 6 3 10 0 // affect 10 to @3
7 : 11 2 3 0 // 2 inferior to 3
8 : 10 2 20 0// jump to 20 if 2 is false
9 : 9 15 0 0 // jump to 15
10 : 5 3 1 0 // copy @1 to @3 (i)
11 : 6 4 1 0 // affect 1 to @4
12 : 1 3 4 3 // add @3 to @4
13 : 5 1 3 0 // copy @3 to 1 (i=i+1)
14 : 9 5 0 0 // jump to 5
15 : 5 3 0 0 // copy @0 to @3 (i)
16 : 6 4 1 0 // affect 1 to @4
17 : 1 3 4 3 // sum @3 and @4
18 : 5 0 3 0 // copy @3 to @0 (a=a+1)
19 : 9 12 0 0// jump back to 12
```

FIGURE 2.1 – Code binaire généré pour une boucle for

```c
/*
     1/AFF
     2/Op (ex: i<5)<--
 ----3/JMF           |
 | -------4/jmp      |
 | |  -->7/op(ex ++) |
 | |  | 8/jmp -------|
 | |--|-> 5/Body
 |    |------6/jmp
 |
 |---> fin du for

*/
```

Appels de fonctions

Problème : gestion de la pile d'appel. Solution : conventions strictes (sauvegarde, restauration, empilement).

Afin de gérer ces problèmes nous avons suivis les instructions des traces de compilation fournis en annexes dans moodle.

2.4 Résultats obtenus

Programmes correctement gérés :
- Calculs arithmétiques ;
- Conditions imbriquées ;
- Boucles `while` et `for` (partiellement) ;
- Appels de fonctions ;

Voici des exemples de code C simple qu'on a compilé et la compilations donnée. Il est à noter que nous avons rajouter les commentaires à la main afin de mieux expliquer les différentes partie.

```c
int somme(int a , int b){
    return a+b;
}

int differnece(int a , int b){
    return a-b;
}

int main () {
    int a  = 7;
    if (a == 0){
        int b = a+1;
        if (b == a ){
            printf(somme(a,b));
        }else{
            printf(b);
        }
    }
}
```

```
0 : 9 1 0 0  // jump 1 => jump to main
1 : 6 1 7 0  // affect 7 to @1
2 : 5 0 1 0  // copy @1 to @0
3 : 6 2 0 0  // affect 0 to @ 2
4 : 5 1 2 0  // copy @2 to @1
5 : 5 2 1 0  // copy @1 to @2 => temporary variable
6 : 6 3 10 0 // affect 10 to @3
7 : 11 2 3 0 // 2 inferior to 3
8 : 10 2 20 0// jump to 20 if 2 is false
9 : 9 15 0 0 // jump to 15
10 : 5 3 1 0 // copy @1 to @3 (i)
11 : 6 4 1 0 // affect 1 to @4
12 : 1 3 4 3 // add @3 to @4
13 : 5 1 3 0 // copy @3 to 1 (i=i+1)
14 : 9 5 0 0 // jump to 5
15 : 5 3 0 0 // copy @0 to @3 (i)
16 : 6 4 1 0 // affect 1 to @4
17 : 1 3 4 3 // sum @3 and @4
18 : 5 0 3 0 // copy @3 to @0 (a=a+1)
19 : 9 12 0 0// jump back to 12
```

FIGURE 2.2 – Code binaire généré pour une fonction

```c
int main () {
    int a = 0;
    while (a<20)
```


```
    {
        a = a + 1;
        a = a * 2;
    }
    printf(a);
}
```

```
0 : 9 1 0 0     //jump to main
1 : 6 1 0 0     //affect 0 to @1
2 : 5 0 1 0     //copy @1 to @0
3 : 5 1 0 0     //copy @0 to @1
4 : 6 2 20 0    //affect 20 to @2
5 : 11 1 2 0    // 2 < 1
6 : 10 1 16 0   //jump to 16 if @1 is false
7 : 5 1 0 0     //copy @0 to @1
8 : 6 2 1 0     //affect 1 to @2
9 : 1 1 2 1     //sum @1 and @2 into @1
10 : 5 0 1 0    //copy @1 to @0
11 : 5 1 0 0    //copy @0 to @1
12 : 6 2 2 0    //affect 2 to @2
13 : 2 1 2 1    //multiply @1 by @2 into @1
14 : 5 0 1 0    //copy @1 to @0
15 : 9 3 0 0    //jump line 3
16 : 5 1 0 0    //copy @0 to @1
17 : 19 1 0 0   //print @1
```

FIGURE 2.3 – Code binaire généré pour une fonction

2.5 Instructions assembleur ajoutées

- `NOP` : opération vide (no operation) ;
- `ADD` : addition (`+`) ;
- `MUL` : multiplication (`*`) ;
- `SOU` : soustraction (`-`) ;
- `DIV` : division (`/`) ;
- `COP` : copie de registre ;
- `AFC` : affectation d'une valeur immédiate ;
- `STR` : stockage en mémoire (store) ;
- `LDR` : chargement depuis la mémoire (load register) ;
- `JMP` : saut inconditionnel ;
- `JMF` : saut si faux (jump if false) ;
- `INF` : inférieur à (`<`) ;
- `INFE` : inférieur ou égal à (`<=`) ;
- `SUP` : supérieur à (`>`) ;
- `SUPE` : supérieur ou égal à (`>=`) ;
- `EQU` : égal à (`==`) ;
- `NEQU` : différent de (`!=`) ;
- `OR` : ou logique ;
- `AND` : et logique ;
- `XOR` : ou exclusif ;
- `NOT` : négation logique ;
- `PRI` : affichage (print) ;
- `CALL` : appel de fonction ;
- `RET` : retour de fonction ;
- `PUSH` : augmenter le début de la pile d'une valeur ;
- `POP` : restaurer la pile d'une certaine valeur ;

Chapitre 3

Processeur architecture RISC-V

3.1 Démarche de conception

La partie matérielle a commencé par la définition des différentes parties :
— Unité Arithmétique et Logique ;
— Banc de Registres ;
— Mémoire des données ;
— Mémoire des instructions ;
— Etage ;

3.2 Choix d'implémentation

Nous avons procédé à l'implémentation de l'architecture suivante en premier lieu :

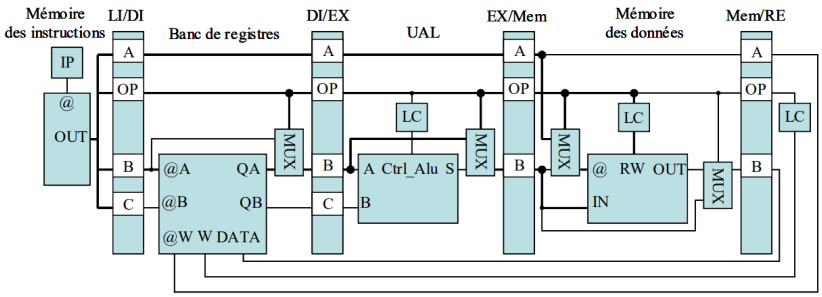

FIGURE 3.1 – Architecture suggéré pour un processeur

Nous avons implémenté les differents multiplexeurs, les controleurs logiques ainsi que le chemin de données.

3.3 Problèmes rencontrés et solutions

3.3.1 Synchronisation

Selon les spécifications du cahier des charges, la mémoire de données devait être synchronisée à l'horloge, aussi bien en lecture qu'en écriture. Afin de maintenir une cohérence globale, nous avons également synchronisé tous les étages du processeur. Toutefois, cela a entraîné un problème de synchronisation : lors d'une lecture mémoire, les signaux de destination et d'opération arrivaient une période d'horloge avant la sortie effective des données de la mémoire. Cette désynchronisation, due au retard induit par la synchronisation sur le dernier étage, a nécessité des ajustements pour garantir un fonctionnement cohérent du processeur.
Dans un premier temps, nous avons ajouté un étage intermédiaire permettant de retenir temporairement

les signaux de destination, de source 1 et d'opération. Ces signaux sont ainsi réémis au front montant de l'horloge suivant, afin de les aligner temporellement avec la sortie mémoire. Ensuite, nous avons intégré un multiplexeur à l'entrée source 1 du dernier étage, permettant de sélectionner dynamiquement entre la sortie de la mémoire et la valeur source 1 initiale, selon le type d'instruction exécutée.
Nous avons ensuite constaté que cette solution introduisait un nouvel étage dans le pipeline, allongeant ainsi le chemin critique. Chaque instruction se voyait alors retardée d'un cycle supplémentaire, ce qui augmentait le temps d'exécution global. Afin de minimiser ce temps et de préserver les performances du processeur, nous avons cherché à optimiser notre approche.

Nous avons finalement opté pour une architecture dans laquelle le dernier étage relaie directement la valeur d'entrée de la source 1 (B), celle-ci étant déjà synchronisée avec l'horloge via la mémoire. Seuls les autres signaux (l'opération, la destinationet la source 1 qui correspond à la sortie de l'étage d'avant) sont alors synchronisés à ce niveau, ce qui permet de limiter la latence tout en conservant la cohérence nécessaire à l'exécution des instructions.

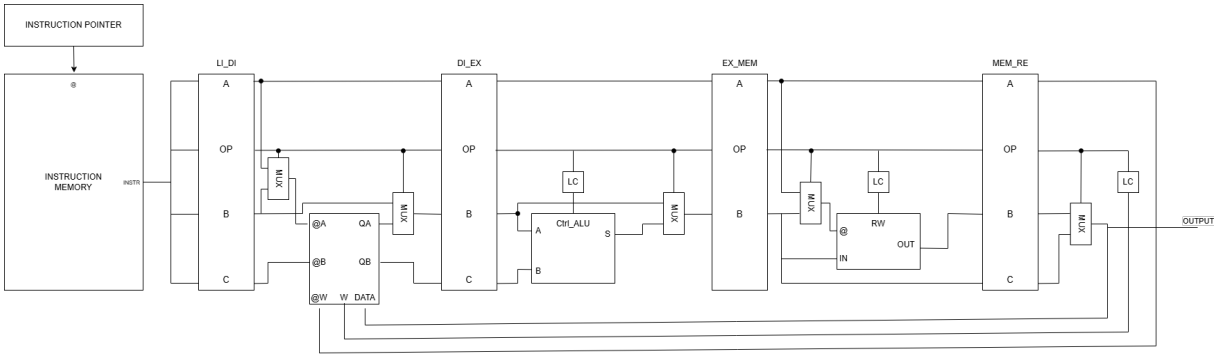

FIGURE 3.2 – Architecture adaptée

3.3.2 Impression sortie

L'instruction `PRINT` spécifie uniquement une destination, sans être directement liée à la mémoire des registres. Pour pouvoir accéder à la valeur correspondante, nous avons donc dû ajouter un multiplexeur entre la destination et la source 1 à l'entrée de la mémoire des registres. Ce multiplexeur permet de charger le contenu de la destination, de le relayer, puis de l'acheminer jusqu'au dernier étage pour être affiché. Ce mécanisme est illustré dans la figure présentée ci-dessus.

3.3.3 Gestion des aléas de données

Comme chaque instruction nécessite plusieurs cycles pour s'exécuter (par exemple, cinq cycles pour une écriture dans un registre), nous avons rapidement compris qu'une opération ne pouvait être exécutée que si ses opérandes (sources) ne correspondaient pas aux destinations des instructions en cours dans les différents étages du pipeline, en particulier celles qui réalisent une opération d'écriture. Cette contrainte est essentielle pour éviter les conflits de données et garantir l'intégrité des résultats produits par le processeur. Nous avons donc géré ces conditions de dépendances dans un processus situé au niveau de la mémoire de données. Lorsqu'un aléa est détecté, nous injectons des instructions `NOP` afin de bloquer l'exécution, tout en maintenant le pointeur d'instructions inchangé. Cela permet de préserver la cohérence des données tout en évitant l'exécution prématurée d'instructions dépendantes.

3.3.4 Gestion des sauts absolus

Pour permettre l'exécution des opérations de saut absolu, nous avons intégré au pipeline une vérification spécifique de l'opération en cours. Lorsqu'un `JMP` est détecté, la valeur du pointeur d'instruction (`IP`) est modifiée en conséquence afin de refléter la nouvelle adresse cible. Cependant, cette approche a rapidement révélé un problème de synchronisation : le changement de l'`IP` n'était pas correctement aligné avec l'injection de la nouvelle instruction, l'instruction d'après était systématiquement executée.

Pour remédier à cela, nous avons décidé de synchroniser la mise à jour de l'`IP` sur le front montant de l'horloge. Ce choix garantit que le changement d'adresse ne prend effet qu'au moment exact où une nouvelle instruction doit être chargée, assurant ainsi la cohérence du flot de contrôle.

3.3.5 Gestion des aléas de branchement

Pour permettre l'exécution de sauts conditionnels, nous devions d'abord récupérer la valeur d'un registre source (`src1`), la comparer à zéro au sein de l'unité arithmétique et logique (UAL), puis exploiter le drapeau `Z` (zéro) résultant de cette comparaison. Dans le cas d'une instruction `JMF` (Jump if False), si le drapeau indique que la condition est remplie, nous devons alors modifier la valeur du pointeur d'instruction (`IP`) pour effectuer le saut.
Cependant, ce saut conditionnel soulève un enjeu crucial : les instructions déjà injectées dans les étages suivants du pipeline doivent être annulées pour éviter l'exécution d'opérations invalides. Pour cela, nous avons introduit un signal de contrôle nommé $\texttt{JMP}_F ALSE_F LAG, quis'activelorsquelaconditiondesautestvrifie.$

$\texttt{À la montée de ce signal, la mémoire d'instruction met à jour l'IP avec la nouvelle adresse, et nous réinitialisons l'étage LI}_D I(chargementinstruction/dcodeurinstruction)l'aidedusignaldereset, afindep$

3.4 Exécution

Dans un premier temps nous allons présenter ce code:

```
ROM(0) <= AFC & x"01" & x"09" & x"00";   -- 9 à R1
ROM(1) <= AFC & x"02" & x"07" & x"00";   -- 7 a R2
ROM(2) <= AFC & x"03" & x"06" & x"00";   -- 6 a R3
ROM(3) <= ADD & x"04" & x"02" & x"03";   -- R4 = R2 + R3 = 13 = 0x0d
ROM(4) <= ADD & x"01" & x"01" & x"01";   -- R1 = R1 + R1 = 9 + 9 = 18 = 0x12
ROM(5) <= COP & x"00" & x"01" & x"00";   -- R0 = R1 = 0x12
ROM(6) <= COP & x"01" & x"04" & x"00";   -- R1 = R4 = 0x0d
ROM(7) <= STR & x"05" & x"03" & x"00";   -- @5 = R3 = 0x06
ROM(8) <= STR & x"01" & x"01" & x"00";   -- @1 = R1 = 0x0d
ROM(9) <= LDR & x"07" & x"05" & x"00";   -- R7 = @5 = 0x06
ROM(10) <= JMF & x"10" & x"00" & x"00";  -- JMP 16 => si false on ne fait rien sinon on fait les affectations et on imprime
ROM(11) <= AFC & x"0b" & x"0b" & x"00";
ROM(12) <= AFC & x"0c" & x"0c" & x"00";
ROM(13) <= AFC & x"0d" & x"0d" & x"00";
ROM(14) <= AFC & x"0e" & x"0e" & x"00";
ROM(15) <= PRINT & x"0e" & x"00" & x"00";   -- R7 = @5 = 0x06
```

FIGURE 3.3 – Code exemple

Il est à noter que vu que la valeur du registre 0 n'est pas 0, le saut conditionnel ne sera pas effectué. Ci-dessous les résultats:

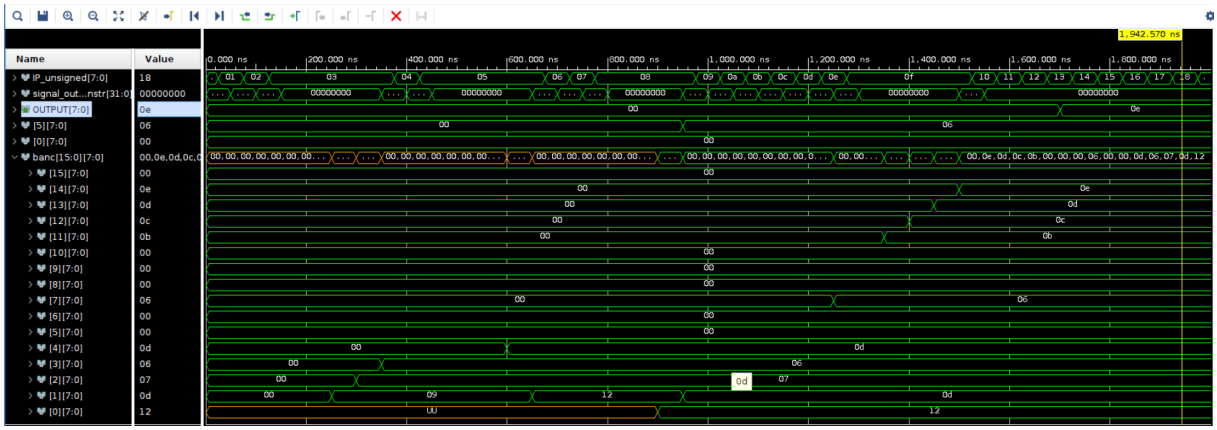

FIGURE 3.4 – Résultats simulation sans saut

En changeant le registre du jmf et en mettant le registre 15 qui est censé contenir 0, on s'attend à ce qu'on effectue le saut et donc ne pas executer les affectations et le print :

```
ROM(0) <= AFC & x"01" & x"09" & x"00";   -- 9 à R1
ROM(1) <= AFC & x"02" & x"07" & x"00";   -- 7 a R2
ROM(2) <= AFC & x"03" & x"06" & x"00";   -- 6 a R3
ROM(3) <= ADD & x"04" & x"02" & x"03";   -- R4 = R2 + R3 = 13 = 0x0d
ROM(4) <= ADD & x"01" & x"01" & x"01";   -- R1 = R1 + R1 = 9 + 9 = 18 = 0x12
ROM(5) <= COP & x"00" & x"01" & x"00";   -- R0 = R1 = 0x12
ROM(6) <= COP & x"01" & x"04" & x"00";   -- R1 = R4 = 0x0d
ROM(7) <= STR & x"05" & x"03" & x"00";   -- @5 = R3 = 0x06
ROM(8) <= STR & x"01" & x"01" & x"00";   -- @1 = R1 = 0x0d
ROM(9) <= LDR & x"07" & x"05" & x"00";   -- R7 = @5 = 0x06
ROM(10) <= JMF & x"10" & x"0f" & x"00";   -- JMP 16 => si false on ne fait rien sinon on fait les affectations et on imprime
ROM(11) <= AFC & x"0b" & x"0b" & x"00";
ROM(12) <= AFC & x"0c" & x"0c" & x"00";
ROM(13) <= AFC & x"0d" & x"0d" & x"00";
ROM(14) <= AFC & x"0e" & x"0e" & x"00";
ROM(15) <= PRINT & x"0e" & x"00" & x"00";   -- R7 = @5 = 0x06
```

FIGURE 3.5 – Code exemple avec saut conditionnel

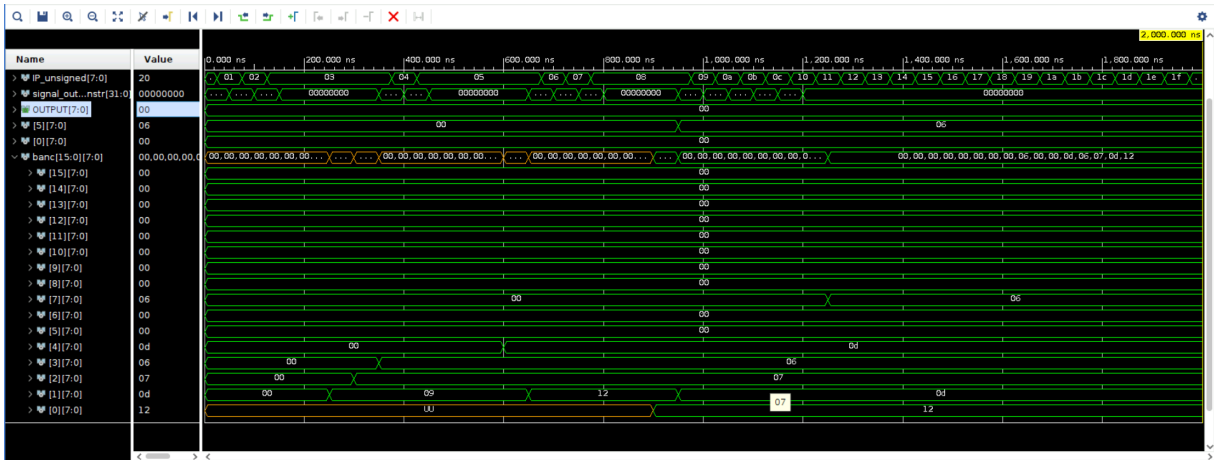

FIGURE 3.6 – Resultats simulation avec saut