Conception d'un compilateur et implémentation du jeu d'instruction en VHDL

Julien Ferry, Paul Florence

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

5 Mai 2019

Conception d'un compilateur

Langage accepté

Notre compilateur accepte des programmes écrits dans un langage C réduit. Ce langage supporte : //TODO : MAJ - les fonctionalités imposées par le cahier des charges (constantes, entiers, « if », « while »…etc) sauf fonctions et pointeurs - les expressions conditionnelles (et, ou…) - les boucles « for »

AST

Le programme à compiler, écrit dans notre langage C réduit (décrit précédemment), est tout d'abord transformé en une suite de tokens par un analyseur lexical produit par Lex. Le résultat est ensuite parsé par l'analyseur syntaxique produit par Yacc. Au cours de son analyse (selon la grammaire que nous avons écrite), cet analyseur va effectuer des actions. En utilisant les fonctions appropriées, il va construire l'arbre syntaxique du programme à compiler. Cette structure arborescente présente deux avantages majeurs : elle nous dispensera, par la suite, d'avoir à utiliser une table des symboles, et elle permettra d'effectuer plus facilement des optimisations.

```
if(p < l) {
    p = 5 + (2*l);
}
else {
    p = 2 - (l/2);
}
```

Figure 1 – Extrait d'un programme à compiler

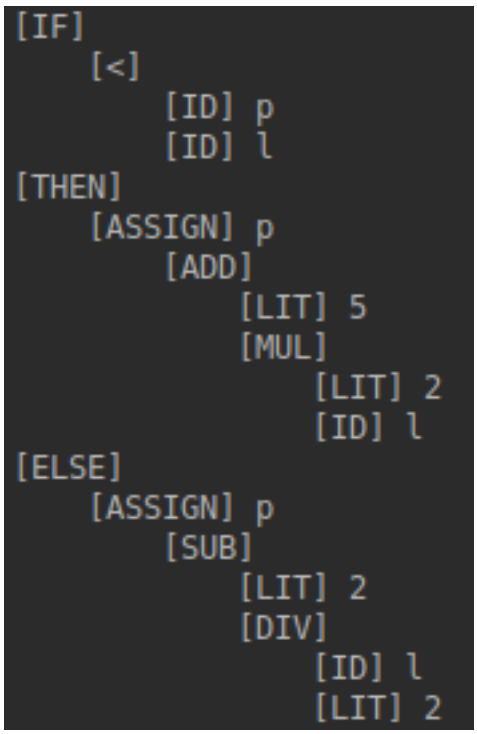

Figure 2 – AST correspondant construit (et affiché dans la console)

Représentation intermédiaire

Une fois l'arbre syntaxique produit, nous conservons une représentation arborescente du programme à compiler, mais transformons chaque noeud en instructions assembleur. Des optimisations peuvent alors être effectuées. Une deuxième passe sur cet « arbre de représentation intermédiaire » permet de « mettre à plat » le code assembleur, c'est à dire de transformer notre arbre d'instructions en une liste chainée d'instructions (le programme assembleur résultant de la compilation).

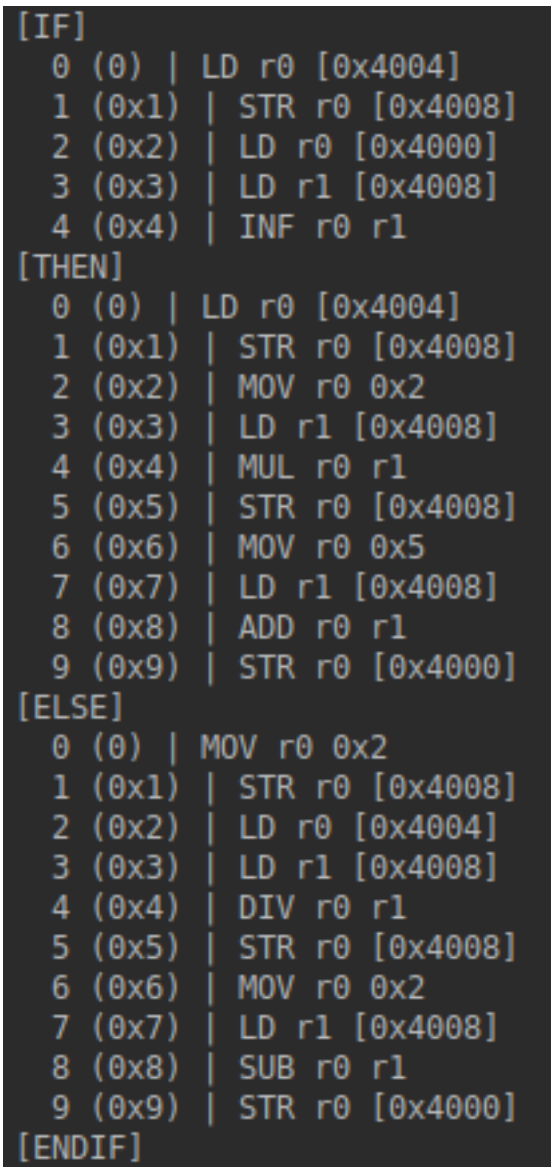

Figure 3 – Arbre de représentation intermédiaire correspondant à l'AST de la figure 2

L'exploitation de cet arbre de représentation intermédiaire ne consiste pas en une simple cancaténation des codes. Il s'agit aussi de générer les instructions de saut et les offset correspondants, pour les boucles et les structures conditionnelles, et d'apporter quelques améliorations au code (entre autres).

```
0 (0) | MOV r0 0
1 (0x1) | STR r0 [0x4000]
2 (0x2) | MOV r0 0x2
3 (0x3) | STR r0 [0x4004]
4 (0x4) | LD r0 [0x4004]
5 (0x5) | STR r0 [0x4008]
6 (0x6) | LD r0 [0x4000]
7 (0x7) | LD r1 [0x4008]
8 (0x8) | INF r0 r1
9 (0x9) | JMPCRELADD r0 @12
10 (0xa) | LD r0 [0x4004]
11 (0xb) | STR r0 [0x4008]
12 (0xc) | MOV r0 0x2
13 (0xd) | LD r1 [0x4008]
14 (0xe) | MUL r0 r1
15 (0xf) | STR r0 [0x4008]
16 (0x10) | MOV r0 0x5
17 (0x11) | LD r1 [0x4008]
18 (0x12) | ADD r0 r1
19 (0x13) | STR r0 [0x4000]
20 (0x14) | JMPRELADD r0 @11
21 (0x15) | MOV r0 0x2
22 (0x16) | STR r0 [0x4008]
23 (0x17) | LD r0 [0x4004]
24 (0x18) | LD r1 [0x4008]
25 (0x19) | DIV r0 r1
26 (0x1a) | STR r0 [0x4008]
27 (0x1b) | MOV r0 0x2
28 (0x1c) | LD r1 [0x4008]
29 (0x1d) | SUB r0 r1
30 (0x1e) | STR r0 [0x4000]
```

Figure 4 – Code assembleur final provenant de la mise à plat de l'IRT de la figure 3

Notes sur la génération de code pour les expressions :

Lors de l'évaluation d'une expression, le cas le plus général peut se résumer à la formule suivante :

Calculer la partie droite, l'empiler ; calculer la partie gauche, l'empiler ; dépiler les deux parties et effectuer l'opération.

C'est ce schéma que nous avons suivi pour notre première implémentation. Cependant, dans des cas simples, cette implémentation s'avère peu performante : elle génère beaucoup de STORE/LOAD consécutifs, et donc des accès mémoire inutiles. Nous avons donc apporté une amélioration qui nous a permis de diminuer la taille du fichier généré pour notre programme test d'environ 30%. Le schéma que nous utilisons est le suivant :

S'il y a une partie gauche : Calculer la partie droite, l'empiler ; calculer la partie gauche, le résultat est contenu dans R0 ; dépiler la partie droite seulement (dans R1) et effectuer l'opération. Sinon : Calculer la partie droite, le résultat est contenu dans R0.

Ainsi, à chaque noeud d'opération, un seul accès mémoire (au maximum) est généré (et il est ici nécessaire car lors de l'évaluation de la partie gauche le contenu du registre R0 risque d'être modifié).

Jeu d'instruction