# 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

### Jeu d'instruction

## Conception d'un compilateur

### Langage accepté

Notre compilateur accepte des programmes écrits dans un langage C réduit. Ce langage supporte :

- les fonctionalités imposées par le cahier des charges (expressions arithmétiques et conditionnelles, constantes, entiers, « if-else », « while »… etc) sauf fonctions et pointeurs
- les opérateurs && et ||
- les boucles « for »
- les variables globales
- les commentaires (// et /* … */)

Il construit et utilise un **arbre syntaxique** pour compiler le programme fourni.

### Différentes étapes de notre processus de compilation

#### 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. Il s'agit ici de créer dans le heap (via de l'allocation dynamique) les noeuds adéquats (instances des structs que nous avons définies) et de les chaîner correctement. Cette structure arborescente présente deux avantages majeurs : elle nous dispensera, par la suite, d'avoir à utiliser une table des labels, et elle permettra d'effectuer plus facilement des optimisations.

#### Représentation intermédiaire

Une fois l'arbre syntaxique produit, nous conservons une représentation arborescente du programme à compiler, mais transformons chaque noeud d'expressions en une séquence d'instructions assembleur. La structure arborescente concerne donc uniquement le « flot » du programme (structures conditionnelles et boucles), et plus les expressions (contrairement à l'AST précédent). 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).

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).

### 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 des registres R0 et R1 risque d'être modifié).

### Notes sur la prise en compte des variables globales :

Un second AST, séparé de celui du programme principal, est généré par l'analyseur syntaxique. A partir de cet AST, on génère une représentation intermédiaire arborescente, et on commence à remplir la table des symboles. L'aspect arborescent de cette représentation intermédiaire est limité aux structures conditionnelles et boucles du programme (if - else, while, for), et en particulier les expressions ne sont plus représentées par des arbres. Par conséquent, *l'arbre correspondant à l'AST de déclaration des variables globales ne comporte qu'une seule branche*. En gardant la même table des symboles et en incrémentant la profondeur, on génère la représentation intermédiaire (arborescente) du programme principal. On chaine ensuite cette représentation intermédiaire sur la fin de celle des variables globales, pour obtenir un seul arbre de représentation intermédiaire. Le traitement est ensuite identique aux processus décrits précédemment.

## Jeu d'instructions

Nos instructions sont codées sur 48 bits. Un tableau présentant notre jeu d'instructions dans sa totalité est disponible en annexe. Le format général de nos instructions est :


**Figure 1 –** Format général de nos instructions

## L'interpréteur

L'implémentation puis l'utilisation d'un interpréteur nous a permis de tester nos programmes compilés. Notre interpréteur est écrit en langage C. La mémoire y est représentée par un tableau, de même que les registres. L'architecture générale est très simple : un pointeur d'instructions désigne l'instruction courante, qui est traitée par un grand bloc « switch », dans lequel les mises à jour correspondant aux instructions sont effectuées. Des informations de débuggage utiles peuvent être affichées.

## Le processeur VHDL

### Fonctionnalités implémentées et présentation rapide

Notre processeur comporte : Une Unité Arithmétique et Logique (ALU) implémentant toutes les opérations arithmétiques (+,*,-) à l'exception de la division ; les instructions logiques (&& et ||) et les comparaisons (>, >=, ==, <, <=) de notre jeu d'instructions. Une banque de registres. Une mémoire d'instructions, un fetch, un pointeur d'instructions et un décodeur. Une unité des gestions des aléas (AMU) . La gestion des sauts (JUMP, JUMPS relatifs et JUMPS relatifs conditionnels). Une RAM.

Les différents composants de notre processeur sont organisés selon un pipeline à 4 étages (cf schéma).

-> Le premier étage charge les instructions, les décode, et lit les données dans la banque de registre de manière asynchrone. En cas d'aléas, l'AMU interrompt le fonctionnement du fetch et entraîne l'envoi d'instructions NOP dans le pipeline. -> Les opérations (arithmétiques, logiques et comparaisons) sont effectuées en asynchrone dans l'ALU situé au second étage. -> Le troisième étage comporte la mémoire. La lecture y est asynchrone tandis que l'écriture y est synchrone. Les buffers séparant les différents étages de notre pipeline fonctionnent bien sûr de façon synchrone. -> Enfin, le dernier étage de notre pipeline abrite le « contrôleur de registres ». C'est ce dernier qui se charge de l'écriture des données

appropriées dans la banque de registres, en fonction de l'instruction en cours. C'est également cette unité qui ordonne au fetch le chargement d'un nouveau PC dans le cas d'un JUMP.

### Schéma de notre processeur

Un schéma représentant l'architecture interne de notre processeur est présenté en page suivante.

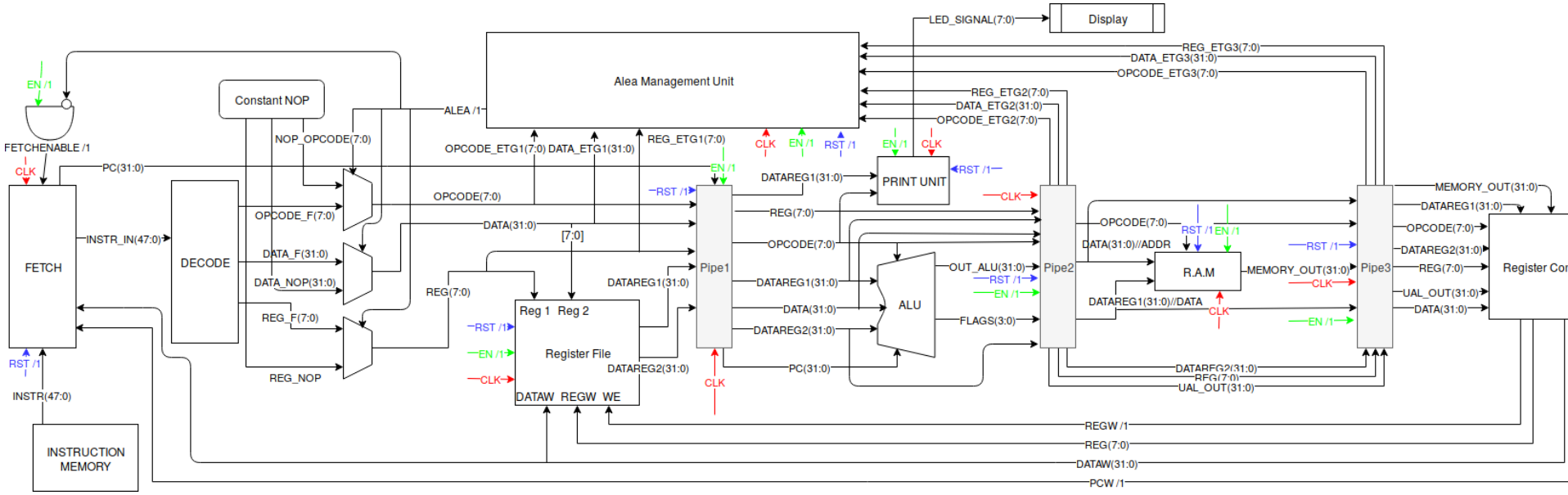


**FIGURE 2 –** Schéma général de notre processeur

# Annexe : Présentation complète de notre jeu d'instructions

## Format

| Champ | OPcode | Opérande1 | Opérande2 |
|---|---|---|---|
| Taille | 8 bits | 8 bits | 32 bits |

## Liste des instructions

| Nom | OPcode |
|---|---|
| MOVE | 0x00 |
| COPY | 0x01 |
| ADD | 0x02 |
| SUB | 0x03 |
| MUL | 0x04 |
| DIV | 0x05 |
| LOAD | 0x06 |
| STORE | 0x07 |
| EQ | 0x08 |
| INF | 0x09 |
| INFEQ | 0x0A |
| SUP | 0x0B |
| SUPEQ | 0x0C |
| PRINT | 0x0D |
| JMP | 0x0E |
| JMPRELADD | 0x0F |
| JMPRELSUB | 0x010 |
| JMPC | 0x11 |
| JMPCRELADD | 0x12 |

| Nom | OPcode |
|---|---|
| JMPCRELSUB | 0x13 |
| NOT | 0x14 |
| AND | 0x15 |
| OR | 0x16 |
| NOP | 0x17 |

### Exemples

| Exemple | OPcode | Opérande1 | Opérande2 |
|---|---|---|---|
| `R0 = R0 + R1` | ADD | R0 | R1 |
| | 0x02 | 0x00 | 0x0000_0001 |
| `R4 = R4 * R12` | MUL | R4 | R12 |
| | 0x04 | 0x04 | 0x0000_000C |
| `R4 = 0x9876_5432` | MOVE | R4 | 0x9876_5432 |
| | 0x00 | 0x04 | 0x9876_5432 |

#### MOVE

#### Assignation de valeur à un registre depuis une donnée

```
MOVE <ra> <val>
```

```
ra <- val
```

#### COPY

```
COPY <ra> <rb>
```

```
ra <- rb
```

**ADD**

```
ADD <ra> <rb>
```

```
ra = ra+b
```

**SUB**

```
SUB <ra> <rb>
```

```
ra= ra - rb
```

**MUL**

```
MUL <ra> <rb>
```

```
ra = ra*rb
```

**DIV**

```
DIV <ra> <rb>
```

```
ra = ra/rb
```

**LOAD**

```
LOAD <ra> <addr>
```

```
ra <- @addr
```

**STORE**

```
STORE <ra> <addr>
```

```
@addr <- ra
```

**EQ**

```
EQ <ra> <rb>
```

```
ra = 1 si ra = rb
```

### INF

```
INF <ra> <rb>
```

```
ra = 1 si ra < rb
```

### INFEQ

```
INFEQ <ra> <rb>
```

```
ra = 1 si ra <= rb
```

### SUP

```
SUP <ra> <rb>
```

```
ra = 1 si ra > rb
```

### SUPEQ

```
SUPEQ <ra> <rb>
```

```
ra = 1 si ra >= rb
```

### PRINT

```
PRINT <ra>
```

```
display the value of <ra> on screen
```

### JMP

```
JMP <unused> <ra>
```

```
Jump to the address of <ra>
```

### JMPRELADD

```
JMPRELADD <unused> <ra>
```

```
Jump to pc+ <ra>
```

**JMPRELSUB**

```
JMPRELSUB <unused> <ra>
```

```
Jump to pc-<ra>
```

**JMPC**

```
JMPC <ra> <rb>
```

```
Jump to <rb> if <ra> is 0
```

**JMPCRELADD**

```
JMPCRELADD <ra> <rb>
```

```
Jump to pc+<rb> if <ra> is 0
```

**JMPCRELSUB**

```
JMPCRELSUB <ra> <rb>
```

```
Jump to pc-<rb> if <ra> is 0
```

**NOT**

```
NOT <ra> <rb>
```

```
<ra> = 1 if <rb> = 0
<ra> = 0 otherwise
```

**AND**

```
AND <ra> <rb>
```

```
<ra> = 1 if (<ra> != 0 and <rb> !=0)
<ra> = 0 otherwise
```

**OR**

```
OR <ra> <rb>
```

```
<ra> = 1 if (<ra> != 0 or <rb> !=0)
<ra> = 0 otherwise
```

**NOP**

```
NOP
```