INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Projet Systèmes Informatiques

Du compilateur vers le microprocesseur

Rédigé par : LACROIX Raphaël - LEJEUNE Aurélia

- Version du 31 mai 2023 -

Première partie

Document

Table of Contents

Lien du Repo Git

1 Introduction

Dans le cadre de notre 4^e année à l'INSA de Toulouse, nous avons réalisé un compilateur C ainsi qu'un CPU porté sur FPGA conçus pour fonctionner ensemble. L'intégralité de notre code se trouve sur notre Git à l'adresse suivante :

https://git.etud.insa-toulouse.fr/rlacroix/Projet-Systemes-Informatiques.git

Nous avons implémenté toutes les parties obligatoires du projet et nous avons ajouté à cela la gestion des aléas et sauts au niveau du microprocesseur, un cross-assembleur permettant de traduire le code assembleur en code objet compréhensible par le FPGA et une visualisation de l'interpréteur.

2 Démarche conception abordée

Afin de mener à bien le projet, nous avons dans un premier temps simplifié le langage C. Notre compilateur C supporte les opérations arithmétiques (addition, soustraction, multiplication et division entière) avec leur priorité, les déclarations de variables ainsi que les déclarations multivariables, les boucles while et les conditions complexes. Pour le moment, le compilateur ne fonctionne qu'avec des entiers, mais nous avons construit notre code de manière à pouvoir facilement rajouter la possibilité d'utiliser des floats ou d'autres types pour la suite. Nous avons essayé d'être le plus proche possible du vrai langage C sur les opérations que nous supportons, nous avons effectué divers tests de compilation pour voir ceux qui étaient autorisés ou non par ce langage afin de le reproduire au mieux sur notre langage.

Ce projet a demandé un travail assez important. Afin de s'organiser au mieux pour être efficace, nous avons mis en place une méthode de travail. À chaque début d'étapes, nous avons discuté entre nous pour choisir les spécifications puis nous nous sommes répartis les tâches. Enfin, nous avons mis en commun le travail réalisé puis nous avons recommencé une nouvelle discussion. Pour le VHDL, comme il s'agissait d'un nouveau langage, nous avons décidé de travailler ensemble afin de mieux l'appréhender puis une fois que nous étions à l'aise, nous avons séparé le travail.

3 Choix d'implémentation

3.1 Compilateur

3.1.1 Table des symboles

L'objectif du compilateur est de traduire le langage C en langage assembleur. Pour cela, nous avons eu besoin de créer une table de symboles permettant de stocker les différentes informations relatives aux fonctions et à leurs variables internes (contenue dans table.c). Nous avons fait le choix d'implémenter un tableau à la place d'une file. Le tableau a l'avantage d'avoir une complexité en $O(1)$) lorsqu'on a besoin d'accéder à une variable de la table des symboles (pour modifier son état ou pour lire son contenu). Le tableau peut cependant poser problème, car on ne sait pas, à l'avance, quelle sera la taille de la table de symboles. C'est pour cette raison que nous avons implémenté un tableau ré-alouable. Lorsque le tableau atteint sa longueur maximale, nous réallouons un tableau ayant pour taille le double. De même manière, lorsque le tableau diminue jusqu'au quart de sa taille, nous diminuons la taille allouée de moitié.

```
1  /*Checks for the length of the array and reallocates if necessary*/
2  void checkArraySanity(){
3      if (currentIndex == maxIndex){
4          reallocateArray(maxIndex * 2);
5      } else {
6          if (currentIndex < maxIndex / 2 && maxIndex / 2 > START_TABLE_SIZE){
7              reallocateArray(maxIndex / 2);
```

```
        }
    }
}

/*reallocates the array with the specified size*/
void reallocateArray(int size){
    Symbol *temp = (Symbol *)realloc(symbolTable, (sizeof(Symbol) * size));
    if (temp != NULL){
        symbolTable = temp;
    }
    else {
        error("Cannot allocate more memory.\n");
    }
}
```

3.1.2 Proximité avec le language C d'origine

Comme spécifié plus haut, pour essayer de s'approcher le plus possible du C "officiel" nous nous sommes basés sur ce que *gcc* accepte et n'accepte pas dans ses analyses lexicales et grammaticales. Cela nous a permis d'avoir un compilateur plus proche du "vrai" C, et au passage d'en apprendre plus sur le fonctionnement de *gcc* et ses particularités.

3.1.3 Gestion des sauts dans le compilateur

Dans un premier temps, nous avons décidé de gérer les sauts des if et des boucles while avec des labels, avant de remplacer ces labels par le numéro de la ligne en question. Cela nous semblait plus proche de l'assembleur vu dans les cours d'assembleur de 3^e année et nous laissait la marge de manœuvre nécessaire pour potentiellement étoffer le jeu d'instruction (ce qui changerait les offsets de jump). Nous avons par la suite choisi de calculer les lignes des jumps directement depuis le yacc comme demandé dans le polycopié du projet.

Cette approche est bien plus optimisée en termes de parcours du fichier (une seule traversée lors de la compilation) mais présente une rigidité qui complique le processus de cross-compilation que nous avons entrepris par la suite.

3.1.4 Gestions des variables et des registres

La gestion des additions passe par l'utilisation de variables temporaires. En effet, quand on effectue une addition telle que $a = b * c + 3$, il faut dans un premier temps stocker les variables b et c puis mettre le résultat de cette multiplication dans une variable, additionner le tout avec 3 et de nouveau stocker cette information. Nous avons ainsi constaté que pour toutes les opérations arithmétiques, quelles qu'elles soient, nous avons besoin de seulement deux variables temporaires. Pour cette raison, nous avons développé un code permettant d'alterner l'utilisation entre deux variables et de vérifier à chaque fois si la variable existe déjà, pour la créer le cas échéant. (Cela permet de supprimer les variables à chaque fois qu'elles ne sont plus utilisées afin que la mémoire soit plus optimisée).

Nous avons fait la même chose pour le code assembleur lorsque nous avons utilisé des registres.

```
/* Adds an element and returns the offset of it */
int addElementAndGetAddress(char* name, enumVarType type){
    addElement(name,type);
    return getOffset(name);
}

```

```
/* Adds a temporary Int element and returns the offset of it */
int addTempINTAndGetAddress(){
    char name[NAME_MAX_LENGTH];
    if (tempCounter == 0){
        // we create the first temporary variable and use it
        addElement("0_TEMP_INT",INT);
        strcpy(name, "0_TEMP_INT");
    } else if (tempCounter == 1) {
        // we create the second temporary variable and use it
        addElement("1_TEMP_INT",INT);
        strcpy(name, "1_TEMP_INT");
    } else {
        // we use the right temporary variable
        sprintf(name, "%d_TEMP_INT", tempCounter % 2);
    }
    tempCounter++;
    return getOffset(name);
}

```

3.2 Interpréteur et cross-assembleur

3.2.1 Interpréteur

Un interpréteur simple affichant en parallèle les instructions et les valeurs des registres au moment de l'exécution de l'instruction en question *(package : rich, v13.3.5)*.

AFC 5 0	5 :0
COP 0 5	5 :0, 0 :0
AFC 5 20	5 :20, 0 :0
COP 1 5	5 :20, 0 :0, 1 :20
AFC 5 0	5 :0, 0 :0, 1 :20
COP 3 5	5 :0, 0 :0, 1 :20, 3 :0
...	...

Une version plus élaborée permettant d'afficher graphiquement (en CLI) le code étape par étape avec les registres sur le côté a également été développée. *(package : rich, v13.3.5, textual v0.26.0)*

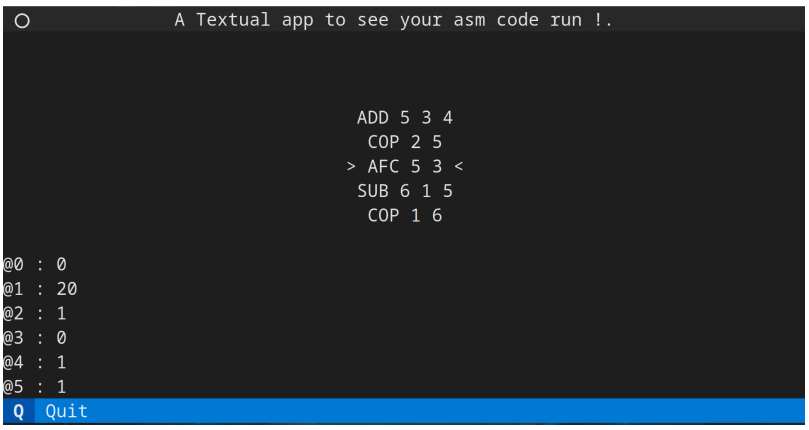

FIGURE 1 – Capture d'écran de l'interpréteur

3.2.2 Le Cross-Assembleur

Nous avons fait le choix d'utiliser un cross-compilateur et non de modifier notre grammaire.

Le cross-compilateur permet en effet, à partir de règles simples de correspondance des *instruction sets*, de générer un assembleur pour une autre cible très facilement (une quinzaine de lignes de code à changer). Toutefois, l'optimisation du code obtenu n'est pas garantie, ce qui fait de cette solution un remplacement critiquable à la compilation directe.

Ci-dessous les 10 premières lignes de l'assembleur généré par le yacc, son équivalent orienté registre et ce dernier sous format utilisable dans du code VHDL.

Assembleur Mémoire	Assembleur Registres	Assembleur pour VHDL
AFC 1 5	AFC 0 5	(x"06000500")
COP 0 1	STORE 1 0	(x"08010000")
AFC 1 20	LOAD 0 1	(x"07000100")
INF 2 0 1	STORE 0 0	(x"08000000")
JMF 2 9	AFC 0 20	(x"06001400")
AFC 2 2	STORE 1 0	(x"08010000")
ADD 1 0 2	LOAD 0 0	(x"07000000")
COP 0 1	LOAD 1 1	(x"07010100")
JMP 2	INF 2 0 1	(x"09020001")
NOP	STORE 2 2	(x"08020200")

3.3 CPU

3.3.1 Opérations supportées

Voici la liste de toutes les opérations assembleurs supportées par notre microprocesseur avec la liste de leurs codes hexadécimaux utilisés durant le développement :

Instructions	Codes hexadécimaux	Instructions	Codes hexadécimaux
ADD	x"01"	SUP	x"0A"
MUL	x"02"	EQ	x"0B"
SUB	x"03"	NOT	x"0C"
DIV	x"04"	AND	x"0D"
COP	x"05"	OR	x"0E
AFC	x"06"	JMP	x"0F"
LOAD	x"07"	JMF	x"10"
STORE	x"08"	PRI	x"13"
INF	x"09"	NOP	x"FF"

Nous avons fait le choix d'implémenter seulement les instructions INF, SUP, EQ et NOT pour les instructions booléennes. L'opération "$\geq$" peut être réalisée avec un NOT suivi d'un INF. De même pour l'instruction "$\leq$" avec un NOT suivi d'un SUP.

3.3.2 Gestion des aléas

Lorsqu'une instruction qui écrit dans un registre est suivie d'une instruction qui lit dans ce même registre, un aléa de données se produit. Cela est dû au fait que l'écriture se passe durant le 5^e étage de notre pipeline (Write Back) alors que la lecture se fait au deuxième étage. La deuxième instruction lira alors la mauvaise donnée.

Pour éviter que cela ne se produise, nous avons besoin de laisser un temps suffisant pour que la première instruction ait le temps d'écrire avant que la deuxième instruction n'arrive pour lire.

Nous avons dans un premier temps répertorié toutes les instructions pouvant poser problèmes ensemble. Il s'agit de toutes les instructions qui écrivent dans le banc de registre (opérations arithmétiques et booléennes, COP et AFC) lorsqu'elles sont suivies d'instructions qui peuvent lire dans le banc de registre (opérations arithmétiques et booléennes, COP et STORE). Une fois ces instructions identifiées, nous avons implémenté, dans notre AleaControler la vérification que ces instructions ne se trouvent pas à la suite, ou à 2 ou 3 d'écart l'une de l'autre. Si cela s'avère être le cas, on bloque le pointeur d'instruction pour qu'il ne continue pas d'avancer et on remplace l'instruction lue par l'étage Li (le premier étage du pipeline) par un NOP pour que cette instruction prématurée qui va lire dans le banc de registre ne soit pas prise en compte par notre processeur.

4 Problèmes rencontrés et solutions trouvées

4.1 Difficulté à déboguer le yacc

La phase de rédaction du lex et du yacc fut mouvementée et marquée par des moments d'incompréhension des erreurs. Toutefois, en nous appuyant sur la documentation, nous avons trouvé (et recensé sur la page 4IR du Wiki) les directives suivantes :

- **bison ... -Wcounterexamples** pour pouvoir afficher un contre-exemple aux règles ce qui nous a permis de ne pas tester à l'aveuglette
- **%define parse.error detailed** qui permet de donner des détails sur l'erreur en question plutôt qu'un "syntax error" qui nous bloquait souvent.

4.2 Gestion des sauts

Notre processeur supporte les boucles while et les if. On peut distinguer de types de sauts d'instructions : les JMP (sauts à l'instruction indiquée) et les JMF (sauts à l'instruction suivante si la condition est fausse).

Les JMP ont été assez faciles à mettre en place. Lorsqu'un JMP est détecté, on envoie l'adresse à laquelle sauté au pointeur d'instruction et on met le signal jump à 1 pour indiquer au pointeur d'instruction qu'il doit utiliser l'adresse qu'il a reçu en entrée.

La mise en place des JMF a elle été plus compliquée. Premièrement, il a fallu gérer toutes les opérations booléennes de conditions, mais il fallait également trouver un moyen de savoir si la condition testée était vraie ou fausse. On aurait pu stocker ce résultat dans un registre particulier, par exemple R2 (comme c'est le cas dans notre assembleur), mais nous avons trouvé qu'il serait plus judicieux de lever un flag dès que l'ALU effectuait une opération donnant un résultat booléen. De ce fait, dès qu'on détecte un JMF, comme le calcul a été réalisé à l'instruction précédente, nous avons directement le résultat de la condition qui est stocké dans le flag. Si la condition est vraie, on continue d'exécuter le code, sinon, on fait comme pour le JMP, on vide le pipeline avec des NOP et on met à jour le pointeur d'instruction.

En détectant le saut conditionnel lors du deuxième étage de pipeline, on trouve le bon compris pour ne pas perdre trop de cycles d'horloge. Si le saut n'était pas nécessaire alors, on a fait une exécution spéculative sur une instruction (sans effectuer aucune opération non-réversible comme une écriture), on ne perd aucun cycle d'horloge. Si le saut était nécessaire, on n'aurait perdu qu'un seul cycle d'horloge, car le saut est détecté rapidement grâce au flag levé par l'instruction précédente dans l'ALU, sans devoir attendre que ce résultat soit transmis par l'écriture dans un registre.

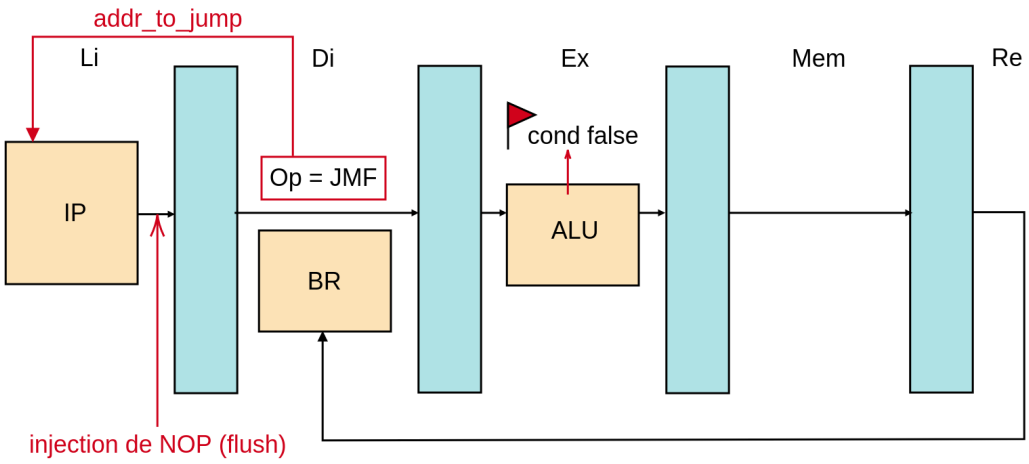

FIGURE 2 – Schéma représentant le cas où un JMF doit être effectué

4.3 Mémoire de données

Nous avons rencontré un problème lors de la simulation au niveau de la mémoire de donnée. Nous nous sommes rendus compte lorsque nous avions un STORE suivi d'un LOAD que la lecture ne lisait pas la donnée, car celle-ci n'avait pas encore été écrite dans la mémoire par l'instruction précédente. Ce problème est dû au fait que l'écriture et la lecture dans la mémoire soit synchrone. Pour régler ce problème, nous avons passé la lecture en asynchrone. Il nous semblait important de laisser l'écriture en synchrone pour éviter qu'une mauvaise valeur soit inscrite en mémoire suite au changement asynchrone des signaux (ex : si on change le signal de DataIn mais que le signal d'écriture reste à '1' pendant un petit laps de temps).

5 Résultats obtenus

Nous avons pu constater durant nos différents tests que notre microprocesseur fonctionne comme prévu. Voici un exemple d'une exécution qui montre les différentes opérations et leur impact sur le microprocesseur.

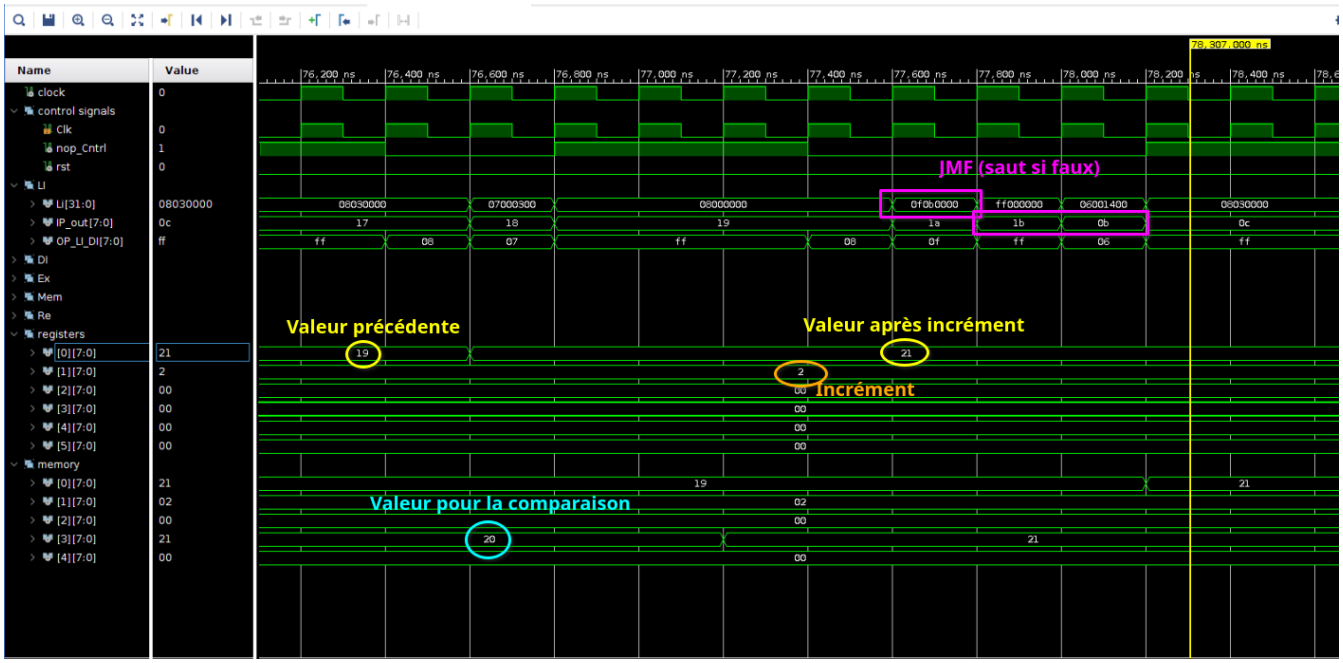

FIGURE 3 – Résultats d'une simulation d'instructions

Cette simulation montre l'exécution du code C suivant :

```
int main() {
    int a;
    a = 5;
    while(a < 20){
        a = a+2;
    }
}
```

Ainsi, on peut voir que la valeur de l'incrément "+2" est situé dans un registre, de même pour la valeur courante de a (en jaune). On constate que celle-ci a bien été augmentée par 2. On remarque également que la valeur de comparaison pour le while (20) est stockée en mémoire. En violet, on peut voir un saut effectué lorsque la condition du while devient fausse. Le numéro d'instruction est changé afin de sortir de la boucle.

Après avoir réalisé la synthèse, on obtient les résultats suivants :

- La fréquence max de fonctionnement est de 58,824MHz (période de 17ns).
- Notre circuit est composé de 342 bascules flips flops.
- L'étage qui limite la fréquence de fonctionnement est l'étage de l'ALU (chemin critique). En effet, certains calculs comme la multiplication peuvent être très long à faire.

L'étage limitant après l'ALU a une slack de 3,498ns (contre 0,539ns pour l'ALU). Si nous arrivons à optimiser l'étage de l'ALU, nous pourrons grandement augmenter les performances de notre microprocesseur. Pour cela, on pourrait par exemple subdiviser cet étage.

6 Amélioration

6.1 Implémentation de fonctions

L'implémentation de fonctions impliquerait notamment :

- De réorganiser la mémoire sous un fonctionnement de piles sur lesquelles on empilera le contexte (valeurs des registres, adresse de retour, valeur de l'EBP précédent), les variables passées en argument et les variables locales.
- De spécialiser un registre (EBP) dans le circuit du FPGA pour accéder aux variables de la frame avec une adresse relative (ne gérant que les unsigned int, il faudra que toutes les informations du contexte et des arguments soient placées *au-dessus* de l'adresse pointée par EBP).
- De gérer les appels et retours de fonctions dans le yacc, en les traduisant par des instructions du langage déjà défini (CALL → STORE des informations, JMP au code de la fonction ; RET → LOAD des informations, STORE de la valeur de retour, JMP à l'adresse de retour). Il n'y a pas besoin de rajouter d'instruction particulière (d'autant qu'un CALL ne pourrait pas être résumé en une seule instruction).

6.2 Amélioration du Cross Assembleur

Pour le Cross-Assembleur, nous proposerions l'amélioration suivante avec pour but de retirer les STORE/LOAD de la même variable :

```
ASMLinesRegisterBis = ASMLinesRegister[:]
for i, l in enumerate(ASMLinesRegister):
    try :
        a = l.split()
```

```
        b = ASMLinesRegister[i+1].split()
        if a[0] == "STORE" and b[0] == "LOAD" and a[1] == b[2] and a[2] == b[1]:
            ASMLinesRegisterBis[i:i+2] = []
    except :
        pass
print("regs opti: ", ASMLinesRegister)
```

Cette solution n'étant pas parfaite, (en effet, on peut stocker m(1) dans r0, m(2) dans r1 puis, voulant lire m(1), le restocker dans r0 alors qu'aucune valeur n'a été écrite par-dessus) il faudrait en fait tenir compte à tout instant de quel registre est utilisé pour minimiser le nombre d'accès mémoire et maximiser l'utilisation des registres (ce qui n'est absolument pas le cas actuellement vu que nous n'en utilisons que 2 sur 16).

6.3 Amélioration sur le microprocesseur

Une des premières améliorations possibles sur le microprocesseur serait de faire la détection du JMP dès la sortie de la mémoire d'instruction, et non après la sortie du premier étage de pipeline qui est synchronisé avec la clock. En faisant cela, on gagnerait un tic d'horloge lorsque le saut serait effectif.

Pour l'instant, nous n'avons pas implémenté la fonctionnalité PRINT. Nous voulions porter notre code sur FPGA de manière à ce que l'on puisse choisir les registres que nous voulons voir. Les LEDs nous auraient donné les valeurs contenues à l'intérieur de ces registres. Ainsi, nous aurions pu voir pas à pas l'avancée du code en appuyant sur un bouton du FPGA pour faire défiler la clock. Malheureusement, après plusieurs heures d'essais, nous n'avons pas réussi à porter notre code sur FPGA.

La fonctionnalité PRINT aurait pu servir par la suite à afficher les valeurs des registres en temps réel sur le FPGA. Le PRINT aurait été une instruction pour faire sortir un signal du banc de registre vers l'écran du FPGA.

7 Conclusion

Ce projet a été très formateur. Il nous a permis de mieux comprendre le fonctionnement d'un compilateur, d'un parseur ainsi que l'architecture d'un pipeline à cinq étages et le fonctionnement général des systèmes informatiques matériels. Nous avons pu nous familiariser avec un nouveau langage avec tous les concepts propres à la conception d'un système informatique.

Nous sommes fiers d'avoir réussi à faire fonctionner notre implémentation en partant de la compilation d'un langage C simplifié vers un microprocesseur sur FPGA. Certaines fonctionnalités peuvent toutefois être encore implémentées dans le but d'augmenter la liste des instructions supportées ou d'optimiser le tout.

INSA Toulouse
135, Avenue de Rangueil
31077 Toulouse Cedex 4 - France
www.insa-toulouse.fr

Université Fédérale Toulouse Midi-Pyrénées

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE
MINISTÈRE
DE L'ÉDUCATION NATIONALE,
DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE