INSA | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Projet Systèmes Informatiques

Rapport de projet

Tuteurs : Dragomirescu Daniela, Alata Eric

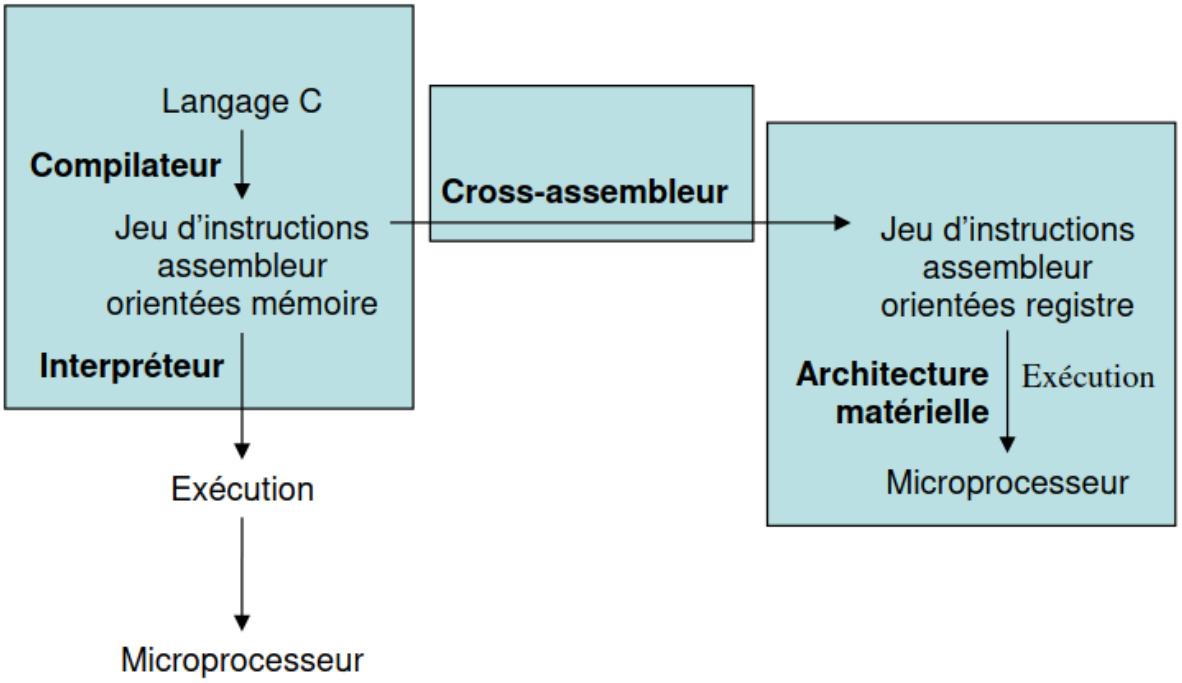
19 mai 2022

José Organista
Keziah Sorlin
4-IR A2

Introduction

Le compilateur est le programme qui permet la création d'un code objet à partir d'un code source. Il est nécessaire lors d'utilisation de langage de haut niveau et permet d'implémenter de plus en plus de fonctions et opérations de base d'un langage de programmation. Au cours de ce rapport, nous expliquerons comment nous avons conçu notre compilation ainsi que les différentes liaisons permettant de transformer un code C vers une lecture d'opération sur VHDL pour notre carte Basys 3 FPGA. Nous développerons les réflexions distinctes qui ont conduit à notre implémentation de chaque sous-parties.

Premièrement nous verrons comment nous avons implémenté notre compilateur de C par l'analyse syntaxique et sémantique en Lex et Yacc, puis l'écriture en assembleur , secondement les interpréteurs et cross-assembleur développés et troisièmement l'implémentation en VHDL des opérations du cross-assembleur pour l'application au niveau physique.



Figure[1] : Système informatique

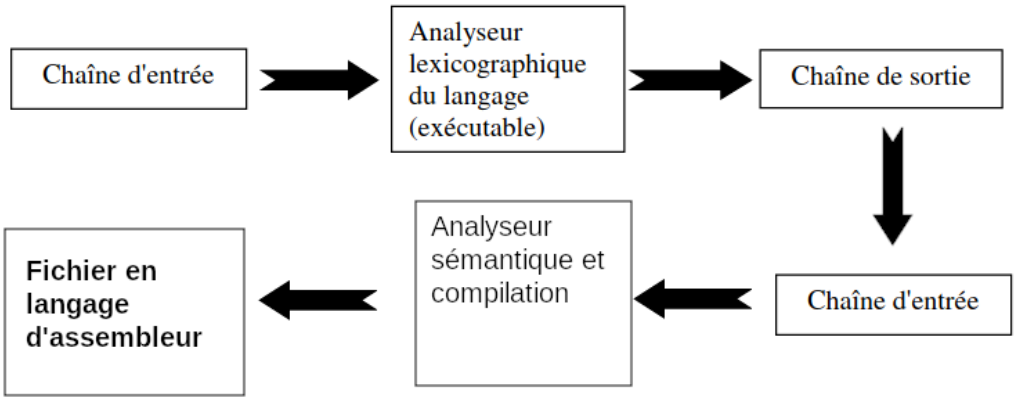
Vous trouverez ci-dessous le lien menant vers notre git contenant le code du projet.

https://github.com/jodorganistaca/Projet_SI

Compilateur

Plan de conception

Nous avons tout d'abord recherché les différentes opérations en C et l'application des syntaxes de base lues par le C.
Premièrement, nous avons créé différents tokens pour chaque syntaxe du C dans le lexique Lex. Deuxièmement, nous avons établi les règles syntaxiques associant chaque opérations de base du C comme l'attribution de valeur.



Figure[2]: Schéma de compilation Lex/Yacc

Implémentation du Compilateur

Après avoir implémenté les différents tokens pour chaque fonctionnalité et mis en place le Yacc qui reconnaît les chaînes de tokens, nous avons commencé à traduire tout cela en assembleur.

Affectation des variables

Nous nous sommes d'abord heurté au problème de retenue des variables et leur adresse pour les écrire dans le fichier assembleur. Ces dernières ont été retenues par une liste chaînée donnant pour chaque variable, son adresse, son type et sa valeur. Des variables temporaires ont été ajoutées pour gérer les affectations de valeur. Les type contiennent donc comme dans le sujet constante, pointeur, fonction, et integer.

Opérations +-/*

Puis nous avons implémenté les opérations qui nous ont demandé d'ajouter un principe de priorité dans notre reconnaissance syntaxique pour les multiplication, division et parenthèses, de plus nous avons été obligé d'augmenter notre nombre de valeur temporaire pour ne pas effacer les précédentes opérations retenues en affectation. L'exponentiel aurait pu être ajoutée en améliorant à priorité maximale.

Printf()

La fonction printf(int) permet d'afficher des variables, il suffisait simplement de récupérer l'adresse de la variable.

Si / Sinon / Sinon Si / Tant que

L'implémentation des Si / Sinon / Sinon Si / Tant que a été plus fastidieuse, elle nous a forcé à créer un tableau dans lequel nous stockons les instructions avant écriture puisque nous devions revenir sur les lignes des JUMP et des JUMP IF pour leur indiquer la ligne de saut d'instruction.
Nous avons créé un tableau faisant office de pile pour récupérer les adresses des sauts d'instructions. Les opérateurs booléens ainsi que les Ou/Et ont dû être fabriqués Nous avons ajouté les instructions SUPE / INFE

pour respectivement Supérieur ou égal et inférieur ou égal. Cela simplifie énormément le code en remplaçant des lignes de jump inutiles. Dans le cas d'imbrication de ces fonctions nous avons dû mettre en place une pile supplémentaire permettant de gérer correctement les adresses de saut.

Pointeur

La création des pointeurs n'était pas complexe en suivant notre ancienne implémentation, en ajoutant la lecture syntaxique de * et d'une variable, il nous suffisait simplement d'attribuer à la variable déclarée en tant que pointeur, l'adresse de la variable à laquelle elle est égale. Par exemple *&Variable* donne l'adresse d'une variable, ***Pointeur* donne la valeur que vise l'adresse stockée par Pointeur. Si la variable n'est pas déclarée en tant que pointeur au préalable il n'est pas possible de récupérer ***Variable*.

Fonctions

Nous avons conçu des fonctions avec ou sans paramètres et/ou valeur de retour. Etant donné que nous avons recherché a créé des fonctions imbriquées et des fonctions récursives, il était nécessaire de construire un système gardant les adresses de retour et un système de profondeur des variables associées à chaque fonctions. Pour utiliser la récursivité, il est nécessaire d'empiler toutes les variables avant de relancer la fonction pour ensuite dépiler au retour. Notre implémentation ne gère que les fonctions avec un seul integer en retour mais nous aurions pu augmenter le nombre d'adresse retour pour en avoir plusieurs en copie.
Nous avons créé un nouveau tableau contenant le nombre de paramètres de chaque fonction et sa propre profondeur.
Nous avons ajouté les instructions LR/ BJ pour respectivement Retour à la dernière adresse fonction et saut à une fonction. Cela simplifie énormément le code en le laissant visible pour l'utilisateur, LR et BJ sont traités par Python comme une pile d'adresse de retour. BJ va stocker les retours tandis que LR va les dépiler.
Cependant, par optimisation le BJ et LR aurait pu être remplacé par une pile d'adresse dans notre C mais il était plus simple de le gérer en Python et beaucoup moins coûteux en temps de production.

Gestion d'erreur

Finalement la gestion d'erreur a été élaborée pour donner à l'utilisateur la possibilité de voir où sont ses erreurs syntaxique ou sémantique. Plusieurs types d'erreur sont possibles, nous avons mis en place l'erreur d'une variable déjà définie ou non définie, une constante changée et un nombre de paramètres non respectés. Pour afficher la ligne d'une erreur nous avons utilisé l'option yylineno qui affiche en tout temps le nombre de la ligne analysée grâce aux \n. Chaque erreur déclenche un yyerror().

Make file

Pour l'automatisation du lancement des programmes un makefile à été implémenté. Dans ce makefile nous avons utilisé différents paramètres pour la simplification des modifications, comme *GRM="le fichier yacc pour la grammaire"*, *LEX="pour le fichier lexer"*, *BIN="pour le nom de l'exécutable"*. De plus, la commande make clean efface les fichiers *.c .h* et *yy.c*. Nous avons aussi créé la commande "make interpretate" et "make cross" pour l'exécution respective de l'interpréteur et du cross assembleur.

Interpréteur / Cross assembleur

Pour l'interpréteur et le cross assembleur, nous avons créé deux programmes en python pour la réalisation de chaque tâche. Le but étant pour l'interprète de comprendre le code et éxecuter seulement ce qui est nécessaire. Le cross-assembleur envoie lui sous forme de 4 octets par ligne les instructions qui doivent être lues après traitement par le VHDL.
Pour l'interpréteur nous avons utilisé le structure de données dictionary ou ligne par ligne en fonction de l'opération lue (AFC, COP, ADD, JMP, etc), nous simulons le comportement du code assembleur. Par exemple, si on lit une opération JMP, nous sautons sur la ligne indiquée par l'instruction. Étant donné que nous avons ajouté deux opérations pour gérer les fonctions (BJ, LR) nous avons utilisé une pile auxiliaire pour sauvegarder l'adresse de retour. Pour le cross assembleur le programme était pareil. Nous retrouvons un dictionnaire, un array auxiliaire

et une boucle pour la lecture ligne par ligne. De ce fait, chaque opérations qui changent ou lisent une valeur entrée ont été transformée par l'opération correspondante en VHDL sur notre processeurs, par exemple AFC => XX05XX00.

Problèmes alternatifs & solutions

Notre implémentation du compilateur n'est pas parfaite, il reste énormément de fonctionnalités à ajouter. Comme cité précédemment nous aurions pu mettre en place plusieurs variables de retour après une fonction. Il suffirait d'ajouter différentes adresses pour les retours et copier aux variables de gauche respectivement par ordre de gauche à droite.
Pour afficher toutes les erreurs syntaxiques et sémantiques , il suffirait d'écrire sur un fichier toutes les erreurs puis les lire dans un yyerror qui serait lancer en fin de lecture avant écriture par un boolean true lorsqu'il y a au moins une erreur. Pour la lecture, bien qu'il y ait des erreurs de syntaxe, il était possible de créer un token qui si rencontré lance la fonction yyerrok et écrit dans un fichier annexe l'erreur pour finalement lire toutes les erreurs une à une.
Notre implémentation du Yacc aurait pu être plus lisible si des fonctions avaient été implémentées pour l'écriture de chaque opération. Cela rend parfois notre code très long et brut.
Dans le cadre de notre projet nous travaillons avec des valeurs limitées à 8 bits soit 256, nous avons donc limité les données à ce maximum donné mais il aurait pu être possible par opérations successives sur le cross assembleur d'avoir des valeurs de taille supérieur à 256 si stockées sur plusieurs adresses.
Au niveau de notre implémentation des fonctions récursives, il aurait été préférable de travailler avec des integers en pointeur pour modifier plus facilement les variables en un nombre d'opérations moindre.
Un bloc de commentaire aurait pu être ajouté par création d'un token, ce qui aurait permis aux utilisateurs de faciliter leur compréhension de leur code.

Résultats

Nous obtenons un compilateur qui s'adapte à de nombreux cas en C basé sur les integers. Le compilateur donne des résultats corrects même après imbrications multiples ou jeu de priorité sur les diverses opérations. Il est cependant limité par sa capacité de stockage pour les valeurs temporelles et les blocs imbriqués. Ci dessous nous pouvons voir à l'exécution que notre compilateur est fonctionnel.

Code	Assembleur	Cross-Assembleur	Interpréter
Test If Else/ elsif			

`int main()`
`{`
`int a=5;`
`if (a<5){`
`a=2*2;`
`}else if(2==2){`
`a=5;`
`}`
`else{`
`a=6;`
`}`
`printf(a);`
 	
`}`
	`AFC 1 5`
	`COP 20 1`
	`AFC 2 5`
	`INF 3 20 2`
	`JMF 3 11`
	`AFC 4 2`
	`AFC 5 2`
	`MUL 6 4 5`
	`COP 20 6`
	`JMP 20`
	`AFC 7 2`
	`AFC 8 2`
	`EQU 9 7 8`
	`JMF 9 18`
	`AFC 10 5`
	`COP 20 10`
	`JMP 20`
	`AFC 11 6`
	`COP 20 11`
	`PRI 20`
		`01060500`
		`14050100`
		`02060500`
		`07060200`
		`08060200`
		`0a060500`
		`14050a00`
			`5`

Test While			

int main()
{
int a=0;
int b;
while(a<3){
b=0;
while(b<2){
b=b+1;
}
a=a+1;
printf(a);
}
printf(a);
printf(b);
 	
}
	AFC 1 0
	COP 20 1
	AFC 2 3
	INF 3 20 2
	JMF 3 20
	AFC 4 0
	COP 21 4
	AFC 5 2
	INF 6 21 5
	JMF 6 15
	AFC 7 1
	ADD 8 21 7
	COP 21 8
	JMP 9
	AFC 9 1
	ADD 10 20 9
	COP 20 10
	PRI 20
	JMP 4
	PRI 20
	PRI 21
		01060500
		14050100
		02060500
		07060200
		08060200
		0a060500
		14050a00
			1
			2
			3
			3
			2

succession d'opération			

main(){
int a =
2+3*4+10/5+(3+2)*2;
printf(a);
}
	AFC 1 2
	AFC 2 3
	AFC 3 4
	MUL 4 2 3
	ADD 5 1 4
	AFC 6 10
	AFC 7 5
	DIV 8 6 7
	ADD 9 5 8
	AFC 10 3
	AFC 11 2
	ADD 12 10 11
	AFC 13 2
	MUL 14 12 13
	ADD 15 9 14
	COP 20 15
	PRI 20
		01060200
		02060300
		03060400
		04020200
		05010100
		06060a00
		07060500
		08030600
		09010500
		0a060300
		0b060200
		0c010a00
		0d060200
		0e020c00
		0f010900
		14050f00
			26

Pointeur:			

int main()
{ int b;
b= 5;
int *o=&b;
*o=3;
printf(b);
}
	AFC 1 5
	COP 20 1
	AFC 2 20
	AFC 21 20
	AFC 4 3
	COP 20 4
	PRI 20
		01060500
		14050100
		02061400
		15061400
		04060300
		14050400
			3

Fonction :			

int sumo(int a, int b, int c){
return a+b+c;
}
 	
int main()
{ int result;
int a=2;
int b=4;
int d=5;
int c=1;
result = sumo(a,b,d);
 	
printf(result);
}
	JMP 6
	ADD 1 20 21
	ADD 2 1 22
	COP 19 2
	LR
	AFC 3 2
	COP 25 3
	AFC 4 4
	COP 26 4
	AFC 5 5
	COP 27 5
	AFC 6 1
	COP 28 6
	COP 20 25
	COP 21 26
	COP 22 27
	BJ 2
	COP 27 27
	COP 26 26
	COP 25 25
		03060200
		19050300
		04060400
		1a050400
		05060500
		1b050500
		06060100
		1c050600
		14051900
		15051a00
		16051b00
		01011400
		02010100
		13050200
		1b051b00
			11

	COP 24 19 PRI 24	1a051a00 19051900 18051300	
Fonction imbriquées			

int suma(int a, int b, int c){
 	
return a+b+c;
}
int sumo(int a, int b, int c){
int d = suma(a,b,c);
 	
printf(d);
return a+b+d;
}
int main()
{ int result;
int a=2;
int b=4;
int d=5;
int c=1;
result = sumo(a,b,c);
 	
printf(result);
}
	JMP 19
	ADD 1 20 21
	ADD 2 1 22
	COP 19 2
	LR
	COP 20 24
	COP 21 25
	COP 22 26
	BJ 2
	COP 26 26
	COP 25 25
	COP 24 24
	COP 28 19
	PRI 28
	ADD 3 24 25
	ADD 4 3 28
	COP 19 4
	LR
	AFC 5 2
	COP 30 5
	AFC 6 4
	COP 31 6
	AFC 7 5
	COP 32 7
	AFC 8 1
	COP 33 8
	COP 24 30
	COP 25 31
	COP 26 33
	BJ 6
	COP 33 33
	COP 31 31
	COP 30 30
	COP 29 19
	PRI 29
		05060200
		1e050500
		06060400
		1f050600
		07060500
		20050700
		08060100
		21050800
		18051e00
		19051f00
		1a052100
		14051800
		15051900
		16051a00
		01011400
		02010100
		13050200
		1a051a00
		19051900
		18051800
		1c051300
		03011800
		04010300
		13050400
		21052100
		1f051f00
		1e051e00
		1d051300
			7
			13

Error			

test > C t7.c

```c
int main()
{
    int a = 5;
    int b=5;
    const c=5;
    c=4;
}
```

```c
int main()
{
    int a = 5;
    int b=5;
    const c=5;
    int b = 2;
}
```

```c

int sumo(int a, int b, int c){

    return a+b+c;
}

int main()
{   int result;
    int a=2;
    int b=4;
    int d=5;
    int c=1;
    result = sumo(a,b,d,d);

    printf(result);
}
```

```
organist@insa-20549:~/Documents/repo/Projet_SI$
Start analysis
    Constante c inmodifiable ligne 6
error line 6: cannot be altered
```

```
Start analysis
    variable b déjà instanciée error ligne 6
error line 6: already instantiated
```

```
Start analysis
error line 13: nombre de paramètres non respecté, trop de paramètre
```

Processeur

Plan de conception

Nous avons utilisé la technique de “diviser pour régner” qui consiste consiste en :

Diviser (Découper le problème initial en sous-problèmes):

Pour l’implémentation en VHDL d’un microprocesseur nous avons implémenté et testé chaque composant du microprocesseur soit une unité arithmétique et logique, un banc de registres à double port de lecture, une mémoire d’instructions, une mémoire des données, un chemin des données, une unité de contrôle, unité de détection des aléas.

Régner (résoudre les sous-problèmes):

Pendant l’implémentation de chaque composant nous avons trouvé différents sous problèmes comme pendant l’implémentation de l’UAL la détection du débordement. Pour le gérer, nous avons utilisé un des signaux auxiliaires de 16 bits et pour les assignations, une redimension finale de la taille a été effectuée.

Combiner :

Nous avons calculé une solution au problème initial à partir des solutions des sous-problèmes. Une fois que chaque composant étaient testés individuellement une source “processor” été créée pour tout rassembler.

les choix d’implémentation

Pour l’implémentation nous avons suivi les recommandations du sujet:

- Une unité arithmétique et logique: des flags pour la détection de débordement, des entrées comme A, B, pour savoir l’opération un contrôle d’alu, et un sortie S. Pour la vérification de débordement l’opération s'était effectué sur 16 bits et après coupé sur 8 bits
- Un banc de registres à double port de lecture: des address pour chaque port (A, B), un signal (flag) pour l'écriture (W=1 écriture/W=0 pas d’écriture), le data à écrire un signal pour faire un reset et une clock. Pour sauvegarder le valeur sur les différents registres nous avons implémenté un registre de tableau de 16 bits dans lequel sur chaque position nous pouvons écrire 8 bits
- Une mémoire d’instructions: Nous avons implémenté une structure qui se base sur un array de 256 instructions chacune constituée de 32 bits. Étant donné que chaque instruction de 32 bits consiste en 4 valeurs A, OP, B, C respectivement register A, operation OP, register B, register C, sur 8 bits. Pour accéder à la valeur de chaque instruction, nous avons un signal add pour l’index de chaque instruction possible et un signal “OUTPUT” pour le valeur de l' instruction sur la position add.
- Une mémoire des données: L'implémentation était faite de manière similaire à la mémoire d'instructions. Par contre, contrairement à la mémoire d’instructions des signaux comme l’entrée, le reset et un signal pour lire/écrire étaient ajoutés. Un signal 8 bits “INPUT” est utilisé comme entrée pour écrire sur le tableau de 256 instructions sur position add. RST est le signal pour nullifier une position dans la mémoire des données. RW comme signal pour savoir si une écriture ou une lecture s’effectue.
- Un chemin des données/Architecture pipe-line sur 5 étages: Pour la communication de chaque composante, différentes configurations ont dû être réalisées en fonction de l'opération à effectuer. Nous avons créé une entité auxiliaire "Buffer Pipeline" pour la détection de chaque étage. Le buffer est constitué de 4 entrées A, Op, B, C qui, à chaque rising edge de l’horloge, copie les entrées dans le sorties. Chaque buffer est connecté, c'est-à-dire que les sorties d’un buffer sont les entrées d’un autre. De cette façon nous savions que sur la première clock les sorties de la mémoire des instructions sont dans le premier étage (Banc de registres) , à la deuxième clock sont dans le deuxième étage (UAL), etc.
- Une unité de contrôle: En cours d’exécution différent multiplexors était implémenté 4 pour le contrôle en fonction des instructions + 1 pour la gestion des aléas.

- Une unité de détection des aléas: Pour la détection des aléas une entité counter a été créée et un multiplexeur est mis en place, de plus un signal auxiliaire est ajouté à chaque fois qu'un aléa est détecté le signal prend la valeur 1. Le compteur est utilisé pour augmenter l'indice du la mémoire des instructions. De ce fait, à chaque fois qu'il y a un aléa le compteur n'augmente pas tant que l'opération n'est pas effectuée.

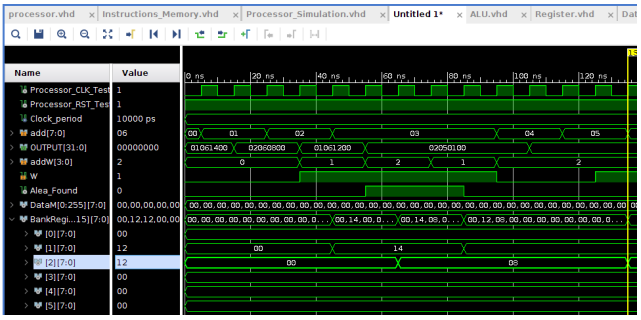
Problème & Solutions

Pendant le développement nous avons trouvé différents problèmes surtout dans la partie d'assemblage.
Le premier était sur la détection contrôlée à chaque étage du processeur la solution était l'implémentation des 4 buffers interconnectés.Une fois ce problème réglé, un nouveau problème sur les connexions entre les sorties du buffers sur les composants est apparu. Nous avons donc implémenté des multiplexeurs. Cependant, nous nous sommes trompés pendant le développement car nous changions les sorties des buffers. Cela écrasait le programme car les sorties ne peuvent changer que sur front d'horloge. Nous avons ajouté des signaux auxiliaires pour changer l'entrée de chaque buffer si besoin.
Finalement, Il fallait effectuer la gestion des aléas s'il y avait un opération de lecture sur un registre qui suit une instruction d'écriture. L'écriture met 4 coups d'horloge à se faire donc si on effectue l'opération trop rapidement la modification sera faussée car la lecture sera effectuée avec un mauvaise valeur. Dans un premier temps, nous avons implémenté un autre clock pour attendre l'écriture toutes les 4 clocks, mais le problème de cette approche c'était que 4 clocks implique que si d'autres opérations qui n'impliquent pas des aléas prennent plus de temps. Donc, nous avons fait une entité compteur et un multiplexeur a été mis en place, de plus un signal auxiliaire a été ajouté chaque fois qu'un aléa est détecté le signal prendra la valeur 1. De cette manière, si il y avait un aléa le compteur n'augmente pas dans l'attente que l'opération s'effectue tout en envoyant des opérations nop.

Résultats

Pour tester, nous avons fait différentes séquences des opérations directement sur le mémoire des instructions. Par exemple, pour tester le fonctionnement AFC et COP.

```
0=> x"01061400", -- AFC 01 14
1=> x"02060800", -- AFC 02 08
2=> x"01061200", -- AFC 01 12
3=> x"02050100", -- COP 04 01
```



Nous obtenons tout en haut, l'horloge que nous utilisons dans tous les composants, l'adresse de la mémoire des instructions, le signal W pour savoir si nous pouvions écrire sur le banc registre .

Conclusion

Pour conclure, au cours de ce projet, nous avons eu l'opportunité de créer un compilateur en LEX/YACC ainsi que de créer un interpréteur et cross-assembleur dans l'optique de l'exécuter de manière physique sur VHDL. Ce projet nous a permis d'avoir des compétences moteurs pour la compréhension globale des machines et des systèmes de l'embarqué. Nous avons été confrontés à de nombreux problèmes comme l'aléa de données ou l'imbrication de fonctions auxquelles nous avons su trouver des solutions. De plus, en travaillant en binôme nous avons amélioré nos connaissances en github et notre organisation d'équipe.