INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Projet Systeme Informatique

Cristian MARTINEZ

Oussama ELJID

Table des matières

Introduction **1**

1 Compilateur **2**

1.1 Grammaire . . . 2

1.2 Table des Symboles . . . 2

1.3 Table des Instructions en Langage d'Assemblage . . . 3

1.3.1 Reconnaissance des Opérations Arithmétiques . . . 4

1.3.2 Reconnaissance des Boucles (IF - WHILE) . . . 5

1.3.2.1 Boucle IF . . . 5

1.3.2.2 Boucle WHILE . . . 6

1.3.3 Reconnaissance de la Fonction Main . . . 6

1.3.4 Reconnaissance des Pointeurs . . . 7

1.4 Reconnaissance des erreurs de compilation . . . 8

1.5 Problèmes rencontrés . . . 8

1.5.1 Gestion des Adresses Mémoire . . . 8

1.5.2 Élimination des Variables Temporaires . . . 8

1.5.3 Résolution des Ambiguïtés dans la Grammaire . . . 8

2 Microprocesseur **9**

2.1 Développement des Composants de Microprocesseur en VHDL avec Vivado . . . 9

2.1.1 Unité arithmétique et logique (UAL) . . . 9

2.1.2 Banc de registre . . . 9

2.1.3 Mémoire des données . . . 10

2.1.4 Mémoire des instructions . . . 10

2.1.5 Chemin des données . . . 10

2.2 Problèmes rencontrés . . . 10

2.2.1 Problème dans l'unité arithmétique et logique : . . . 10

2.2.2 Gestion des Aléas dans le Data Path . . . 11

2.3 Résultats . . . 11

3 Conclusion **13**

Introduction et problème étudié

L'objectif de ce projet est de concevoir un compilateur pour le langage C en suivant les différentes étapes de son développement. Tout d'abord, il a été nécessaire de créer un programme en Lex pour reconnaître les divers éléments du langage, afin de pouvoir les analyser et les traduire en un langage assembleur que nous avons développé. Ensuite, la deuxième phase a consisté à interpréter et exécuter ce langage assembleur. Pour ce faire, nous avons élaboré notre propre matériel en VHDL à l'aide du logiciel Vivado, ce qui nous a permis de le simuler avant de configurer le FPGA Xilinx Artix-7 pour implémenter le circuit numérique.

Chapitre 1

Compilateur

Objectif

Ce chapitre aborde le développement du compilateur pour l'analyse lexicale et syntaxique utilisant LEX et YACC. Notre objectif est de créer un compilateur capable d'analyser des programmes écrits en langage C, générant une table d'instructions en langage d'assemblage pouvant être traitée par un processeur. Nous décrivons ci-dessous les structures essentielles et les méthodes utilisées pour reconnaître le langage.

1.1 Grammaire

La grammaire est une description formelle de l'ensemble des règles régissant la structure d'un langage. Dans notre cas, nous définissons la grammaire du langage C à reconnaître par notre compilateur. La grammaire sert de base pour l'analyse syntaxique et la génération de code. (Insérez ici une image de la grammaire).

1.2 Table des Symboles

La table des symboles est une structure de données utilisée pour stocker des informations sur les identificateurs trouvés dans le code source, tels que les variables et les fonctions. Ces informations comprennent le type de symbole, sa valeur et son adresse en mémoire. La table des symboles est fondamentale pour l'étape d'analyse sémantique du compilateur.

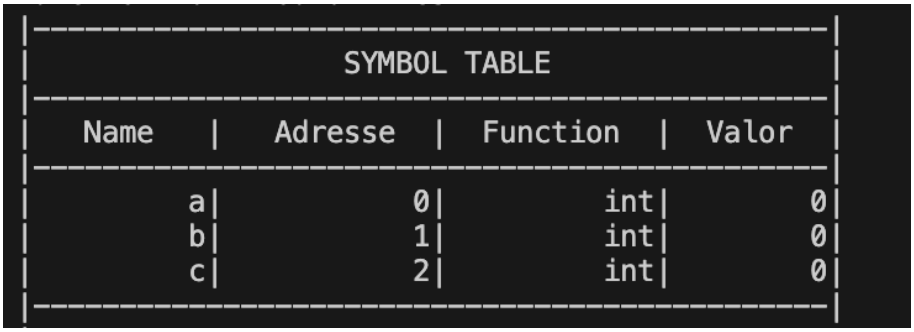
Les fonctions importantes dans cette section sont :

— `void add_symbol(char *name, char *type, int value);`

— `int find_symbol(const char *name);`

— `int find_symbol_by_number(int value);`

— `void delete_symbol(int index);`



SYMBOL TABLE

Name	Adresse	Function	Valor
a	0	int	0
b	1	int	0
c	2	int	0

FIGURE 1.1 – Table de Symboles

1.3 Table des Instructions en Langage d'Assemblage

La table des instructions en langage d'assemblage contient les instructions générées par le compilateur pour être exécutées par le processeur. Cette table est créée pendant la phase de génération de code et est fondamentale pour l'étape d'assemblage du processus de compilation.

Les fonctions importantes dans cette section sont :

— `void add_instruction(char *name, int indexInstruction, int numberRegister, int addressMemory, int addressValTMP);`

— `void update_instruction(char *name, int indexInstruction, int numberRegister, int newAddressMemory, int newAddressValTMP);`

```
void test(){
int a;
int b , c;
b = 6;
c = 8;
 a = b + c * 2;
}
```

FIGURE 1.2 – Code Exemple

```
typedef struct Instruction {
    char *name;              // Instruction Name
    int add_operand_1;       // Address Operand 1
    int add_operand_2;       // Address Operand 2
    int add_operand_3;       // Address Operand 3
    int indexInstruction;    // Address Instruction
} Instruction;
```

FIGURE 1.3 – Table d'instructions

1.3.1 Reconnaissance des Opérations Arithmétiques

Le compilateur reconnaît les opérations arithmétiques dans le code source et les traduit en instructions en langage d'assemblage, telles que ADD (addition), SUB (soustraction), DIV (division) et MUL (multiplication). La reconnaissance et la traduction correctes de ces opérations sont cruciales pour générer du code machine efficace.

Grâce à la directive %left dans la spécification de la grammaire du compilateur (par exemple, avec Yacc ou Bison), il est possible de définir la priorité et l'associativité des opérateurs arithmétiques. Cela permet de s'assurer que les expressions sont évaluées dans le bon ordre, conformément aux règles du langage source. Voici un exemple :

```
%left tADD tSUB
%left tMUL tDIV
%left tASSIGN
```

FIGURE 1.4 – Definition de Priorité

```
operationArith : var
               | operationArith tADD operationArith { in_arithmetic_operation = 1; add_operation("ADD"); }
               | operationArith tSUB operationArith { in_arithmetic_operation = 1; add_operation("SUB"); }
               | operationArith tDIV operationArith { in_arithmetic_operation = 1; add_operation("DIV"); }
               | operationArith tMUL operationArith { in_arithmetic_operation = 1; add_operation("MUL"); }
```

FIGURE 1.5 – Grammaire - Ajout d'instruction

```
void add_operation(char * operation){
    int result = address_symbol - 2;
    int operand_1 = address_symbol - 2;
    int operand_2 = address_symbol - 1;
    add_instruction(operation, address_instruction, result, operand_1, operand_2);
    delete_symbol(address_var_TMP);
};
```

FIGURE 1.6 – Fonction qui ajoute les opérations

```c
typedef struct Instruction {
    char *name;             // Instruction Name
    int add_operand_1;      // Address Operand 1
    int add_operand_2;      // Address Operand 2
    int add_operand_3;      // Address Operand 3
    int indexInstruction;   // Address Instruction
} Instruction;
```

FIGURE 1.7 – Structure d'un instruction

1.3.2 Reconnaissance des Boucles (IF - WHILE)

Pour la reconnaissance des boucles et des structures conditionnelles comme if et while, le compilateur utilise une pile supplémentaire pour stocker les adresses des instructions nécessitant des sauts conditionnels. Voici comment cela fonctionne pour les structures if et while :

Reconnaissance des Boucles et des Conditions Structure if :

```c
StackNode *jmf_stack = NULL;

// ADD A NEW INDEX TO THE STACK
void push(int index) {
    StackNode *new_node = (StackNode *)malloc(sizeof(StackNode));
    new_node->index = index;
    new_node->next = jmf_stack;
    jmf_stack = new_node;
    // print_stack();
}

// DELETE DE LAST ELEMENT AND RETURN THE NEW INDEX
int pop() {
    if (jmf_stack == NULL) {
        return -1; // STACK EMPTY
    }
    StackNode *temp = jmf_stack;
    int index = temp->index;
    jmf_stack = jmf_stack->next;
    free(temp);
    // print_stack();  print the stack
    return index;
}
```

FIGURE 1.8 – Stack pour stoker les index des boucles

```c
void update_jmf(){
    int jmf_index = pop(); // GET THE ADDRESS OF JMF ON THE INSTRUCTION TABLE - WHERE IF STARTS
    idJMF = jmf_index;
    if (jmf_index != -1) {
        update_instruction("JMF", jmf_index, idJMF, address_instruction, 0);
        // UPDATE INSTRUCTION TABLE WITH THE ADDRESS WHERE IF ENDS. LOOK FOR THE JMF + jmf_index IN THE INSTRUCTION TABLE, THEN UPDATE
    }
}
```

FIGURE 1.9 – Fonction pour metter à jour l'adresse de saut

1.3.2.1 Boucle IF

Lorsqu'un if est rencontré, le compilateur ajoute une instruction JMF (Jump if False) au début du conditionnel. Cette instruction est utilisée pour sauter à la fin du bloc if si la condition est fausse. Une fois que le compilateur atteint la fin du bloc if, il met à jour l'instruction JMF initialement définie. Cette mise à jour est réalisée grâce à une fonction update$_jmf qui dépile les éléments de la pile et recherche dans la liste des instructions l'instruction JMF correspondante.$

```
if: tIF tLPAR condition tRPAR { delete_symbol(address_var_TMP);
                                varFirstIF++;
                                add_instruction( "JMF", varFirstIF, varFirstIF , -999 , 0 );
                                idJMF = varFirstIF;
                                push(idJMF);
    }  tLBRACE structure ifStructure
;

ifStructure : tRBRACE { update_jmf();}
            | tRBRACE { update_jmf();} functionBodyReturn
            | tRBRACE tELSE tLBRACE { update_jmf();} structure tRBRACE
            | tRBRACE tELSE tLBRACE { update_jmf();} functionBodyReturn tRBRACE
            | returnStatement tRBRACE { update_jmf();}
;
```

FIGURE 1.10 – Grammaire IF pour le identifier les JMF

1.3.2.2 Boucle WHILE

La reconnaissance d'une boucle while fonctionne de manière similaire à celle de if, avec quelques différences. Au début de la boucle while, une instruction JMF est ajoutée pour gérer le cas où la condition est fausse. À la fin de la boucle while, une instruction JMP (Jump) est ajoutée pour revenir au début de la boucle. Cette instruction utilise l'adresse de la première instruction du while pour boucler.

```
while : tWHILE tLPAR condition tRPAR {  delete_symbol(address_var_TMP);
                                        varFirstWhile++;
                                        add_instruction( "JMF", varFirstWhile, varFirstWhile , -999 , 0 );
                                        idJMF = varFirstWhile;
                                        push(idJMF);
      } tLBRACE whileStructure tRBRACE { update_jmf(); add_instruction( "JMP", address_instruction, varFirstWhile , 0 , 0 ); }
;
```

FIGURE 1.11 – Grammaire WHILE pour le identifier les JMF

1.3.3 Reconnaissance de la Fonction Main

La reconnaissance de la fonction main dans un compilateur peut suivre un principe similaire à celui utilisé pour les structures de contrôle comme if et while.

Au début de la lecture du code, une instruction JMP est ajoutée. Cette instruction est temporairement dirigée vers une adresse à mettre à jour plus tard.

Toutes les fonctions doivent être déclarées avant main. Cela garantit que toutes les déclarations de fonctions sont connues avant d'exécuter main. Mise à Jour de l'Instruction JMP :

Une fois que la première instruction de main est trouvée, l'adresse de l'instruction JMP ajoutée au début est mise à jour pour pointer vers cette première instruction.

```
fun : tVOID tID {  printf("Function VOID Found : %s\n", yytext); nameFunction = $2; } Body
    | tINT tID { printf("Function INT Found : %s\n", yytext);
                 nameFunction = $2;
                 add_symbol("?ADR", nameFunction, 0);  // OK
                 add_symbol("?VAL", nameFunction, 0);
      }
      tLPAR args tRPAR { add_JMP(); }
      functionBodyReturn  tRBRACE { delete_adr_val(); delete_last_symbol(var_to_delete_arg); }
;

Body : tLPAR args tRPAR { add_JMP(); }  tLBRACE structure tRBRACE { add_instruction( "NOP", address_instruction, 0 , 0 , 0 );  }
;
```

FIGURE 1.12 – Grammaire pour ajouter le JMP

```
main: type tMAIN { printf("Function MAIN Found : %s\n", yytext);
                   mainBool = 1;
                   nameFunction = "main";
                   add_symbol("?ADR", nameFunction, 0);
                   add_symbol("?VAL", nameFunction, 0);
                   address_main = address_instruction;
         }
          tLPAR args tRPAR tLBRACE { returnBool = 0;}
          BodyMain tRBRACE {
                              delete_adr_val();
                              add_instruction( "RET", address_instruction, 0 , 0 , 0 );
                              update_instruction("JMP", varFirstJMP, address_main, 0,0 );
                              add_instruction( "NOP", address_instruction, 0 , 0 , 0 );
                              mainBool = 0;
}
;
```

FIGURE 1.13 – Grammaire pour mettre à jour l'adresse du main

1.3.4 Reconnaissance des Pointeurs

Lorsqu'une déclaration de pointeur est détectée, le compilateur ajoute la variable et l'adresse à laquelle elle pointe à la table des symboles. Ensuite, lorsque l'accès est effectué via le pointeur, le compilateur effectue l'opération appropriée pour obtenir la valeur pointée par le pointeur.

```
declaration : tINT assignationAuxList
            | tINT tMUL tID tASSIGN tPOINTER tID { nameID = $3; add_symbol($3, "POINTER", find_symbol($6)); }
```

```
assignationAux : tID {  nameID = $1;  } tASSIGN operationArith {
                    delete_symbol(address_var_TMP);
                    address_variable = find_symbol($1);
                    add_instruction("COPaa", address_instruction, address_variable, address_symbol, 0);
                  }
                | tMUL tID tASSIGN tNB {
                    find_symbol($2);
                    add_instruction("ACF", address_instruction, address_POINTER, $4, 0);
                  }     // POINTER
                | tID {  add_symbol($1, nameFunction, 0); }
;
```

1.4 Reconnaissance des erreurs de compilation

Pour le contrôle des erreurs, nous avons ajouté une vérification syntaxique qui retourne une erreur et la ligne du code lue pour indiquer si le code est correct ou non. Cela permet de détecter et de corriger les erreurs de manière plus efficace, en fournissant des messages d'erreur détaillés incluant le numéro de la ligne où l'erreur s'est produite.

```c
void yyerror(const char *s) {
    fprintf(stderr, "Error: %s at line %d - ERROR JUST BEFORE SYMBOL %s\n", s, yylineno, yytext);
}
```

1.5 Problèmes rencontrés

1.5.1 Gestion des Adresses Mémoire

Assurer que les adresses mémoire sont correctement allouées et libérées pour éviter les fuites de mémoire et les erreurs de segmentation.

1.5.2 Élimination des Variables Temporaires

S'assurer que les variables temporaires sont correctement éliminées après utilisation pour maintenir la propreté de la mémoire et éviter les conflits futurs.

1.5.3 Résolution des Ambiguïtés dans la Grammaire

Identifier et résoudre les ambiguïtés dans la grammaire pour éviter les conflits de parsing (shift/reduce) et assurer un parsing correct et prévisible du code source.

Chapitre 2

Microprocesseur

Objetif

La dernière phase de notre projet consistait à modéliser nos composants pour concevoir un FPGA en simulation, puis à les implémenter sur le FPGA Xilinx. Le microprocesseur construit inclut un pipeline capable de traiter diverses instructions assembleur : addition, soustraction, multiplication, division, copie et affectation. Notre objectif principal était d'assurer que le FPGA puisse traiter efficacement les instructions assembleur développées précédemment. Pour atteindre cet objectif, nous avons conçu un pipeline intégrant plusieurs éléments essentiels :

2.1 Développement des Composants de Microprocesseur en VHDL avec Vivado

2.1.1 Unité arithmétique et logique (UAL)

C'est un composant combinatoire essentiel, conçu pour exécuter des opérations arithmétiques (addition, soustraction, multiplication, division) sans nécessiter de clock. Elle utilise un signal de contrôle pour déterminer l'opération à effectuer et gère des drapeaux indiquant les états tels que la négativité, le débordement, la sortie nulle et la retenue. Grâce à l'UAL, les calculs sont traités immédiatement, optimisant ainsi l'efficacité du pipeline du microprocesseur.

2.1.2 Banc de registre

Nous avons conçu un banc de 16 registres de 8 bits avec un double accès en lecture et un accès en écriture. Le signal de reset (RST), actif à 0, initialise tous les registres à 0x00. Les adresses @A et @B permettent la lecture simultanée de deux registres, avec les valeurs correspondantes propagées vers les sorties QA et QB. L'écriture dans un registre se fait via les entrées @W, W et DATA, où W active l'écriture lorsqu'il est positionné à 1, synchronisée avec l'horloge.Les registres ont été intégrés afin de développer un processeur de type RISC Load/Store. Cette intégration a nécessité l'ajout des instructions Load et Store, permettant les échanges efficaces entre la mémoire et les registres. Ces améliorations ont considérablement optimisé les performances et l'efficacité du microprocesseur.

2.1.3 Mémoire des données

Nous avons conçu une mémoire des données permettant un accès en lecture et en écriture. La mémoire est structurée sous la forme d'un tableau de 256 cases, optimisée pour les instructions Load et Store. L'adresse mémoire est spécifiée par l'entrée @, avec le signal RW positionné à 1 pour les opérations de lecture et à 0 pour les écritures. En cas d'écriture, les données de l'entrée IN sont copiées à l'adresse spécifiée. Un signal de reset (RST) permet d'initialiser toute la mémoire à 0x00. Toutes les opérations de lecture, d'écriture et de reset sont synchronisées avec l'horloge (CLK), assurant une gestion efficace et ordonnée des données. Ces caractéristiques améliorent les performances et l'efficacité globale du système.

2.1.4 Mémoire des instructions

Nous avons conçu une mémoire des instructions sous la forme d'un tableau destiné à stocker les programmes à exécuter par le microprocesseur. Cette mémoire est exclusivement accessible en lecture, ce qui la rend similaire à une ROM (Read-Only Memory). Contrairement à la mémoire des données, elle est dépourvue des entrées RST, IN, et RW, et ne permet aucune modification du contenu après le stockage initial. La lecture des instructions est synchronisée avec l'horloge (CLK), assurant une exécution ordonnée des programmes. Cette mémoire des instructions joue également un rôle crucial dans la gestion des aléas, en contrôlant strictement l'accès pour prévenir toute modification non autorisée.

2.1.5 Chemin des données

Nous avons commencé par créer la structure d'un pipeline, conçu comme une unité dotée de quatre ports d'entrée et de quatre ports de sortie. Ce pipeline sera connecté à tous les éléments développés précédemment. Il a pour objectif de synchroniser les différentes étapes de notre pipeline afin qu'elles fonctionnent harmonieusement au même rythme d'horloge. Cette synchronisation permet à chaque unité de progresser en parallèle. Avec un pipeline à quatre niveaux, une nouvelle instruction est traitée à chaque cycle d'horloge, au lieu d'une instruction tous les quatre cycles, optimisant ainsi l'efficacité et la performance du microprocesseur.

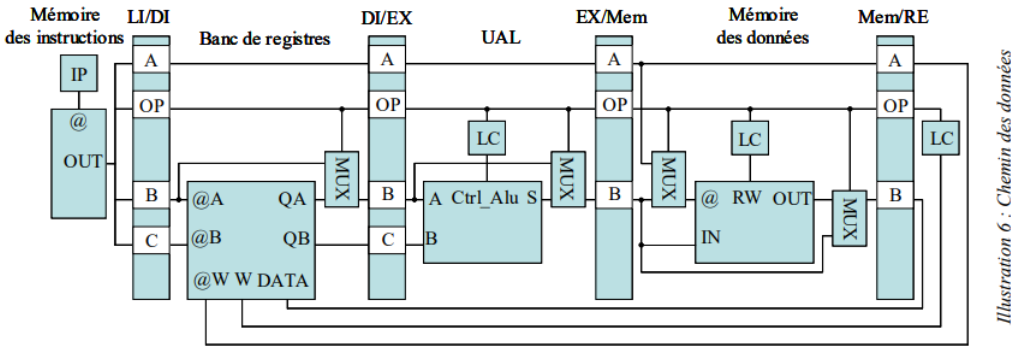


Illustration 6 : Chemin des données

2.2 Problèmes rencontrés

2.2.1 Problème dans l'unité arithmétique et logique :

L'opération de division, prévue initialement, s'est révélée impossible à implémenter dans l'ALU. La division nécessite des étapes répétitives de soustraction et de comparaison, rendant l'opération intrinsèquement plus complexe que l'addition,

la soustraction ou la multiplication. Alors que ces dernières peuvent être réalisées en un seul cycle ou en quelques cycles optimisés, la division requiert plusieurs cycles pour compléter une seule opération. En conséquence, notre code ne reconnaît pas cette opération.

2.2.2 Gestion des Aléas dans le Data Path

Dans la conception de notre unité de traitement des données (data path), nous avons rencontré plusieurs défis liés aux aléas (hazards), qui peuvent perturber le flux d'instructions dans un pipeline. Les aléas se produisent lorsque des instructions consécutives dépendent des résultats intermédiaires des unes des autres, ce qui peut provoquer des conflits et des retards dans l'exécution.

Problèmes d'Aléas Rencontrés :

1. Aléas de Données (Data Hazards) : Ces aléas surviennent lorsque les instructions successives nécessitent l'utilisation des mêmes registres. Par exemple, une instruction pourrait avoir besoin des résultats d'une instruction précédente avant que cette dernière ne soit complètement exécutée.
2. Aléas de Contrôle (Control Hazards) : Ils se produisent lorsque le pipeline doit gérer des instructions de branchement conditionnel ou inconditionnel, nécessitant une mise à jour de l'instruction pointer (IP) en fonction de la condition de branchement.

Solutions Implémentées pour la Gestion des Aléas :

1. Détection et Signalisation des Aléas : Le data path inclut des signaux dédiés pour détecter les aléas. Par exemple, les signaux lidiRead, diexWrite, et exmemWrite identifient si les instructions lisent ou écrivent dans les registres. Le signal alea est utilisé pour indiquer la présence d'un aléa, basé sur les dépendances entre les instructions en cours de traitement.
2. Contrôle des Aléas par No-Operation (NOP) : Lorsqu'un aléa est détecté, une instruction NOP (No Operation) est insérée dans le pipeline pour donner du temps à l'instruction précédente de se terminer. Cela permet de synchroniser correctement les étapes du pipeline sans conflit. Les signaux lidinop, diexnop, et exmemnop sont utilisés pour contrôler cette insertion de NOP.
3. Gestion des Branchement et Aléas de Contrôle : La logique de contrôle des branches assure que les sauts inconditionnels (uncondbranch) et les sauts conditionnels (condbranch) sont correctement gérés, avec le signal branchtaken déterminant si une branche doit être prise. Ceci synchronise le pointeur d'instruction (IP) pour qu'il pointe vers la bonne adresse en cas de branchement.
4. Synchronisation avec l'Horloge : Toutes les opérations, y compris la lecture, l'écriture, et les réinitialisations, sont synchronisées avec le signal d'horloge (CLK), garantissant une exécution ordonnée des instructions et minimisant les conflits.

2.3 Résultats

Les tests effectués sur le pipeline ont démontré son bon fonctionnement. Les instructions se propagent comme prévu à travers les différentes étapes du pipeline. L'exactitude du chemin des données a été vérifiée en examinant le contenu du banc de registres ainsi que de la mémoire des données.

Ci-dessous, un test simple illustre ce processus. Nous chargeons des valeurs dans deux registres, les additionnons, puis stockons le résultat dans le registre R8 du banc de registres. Les deux images ci-dessous montrent les instructions stockées en mémoire ainsi que l'état du chemin des données, y compris le contenu des bancs de registres après l'exécution des instructions.

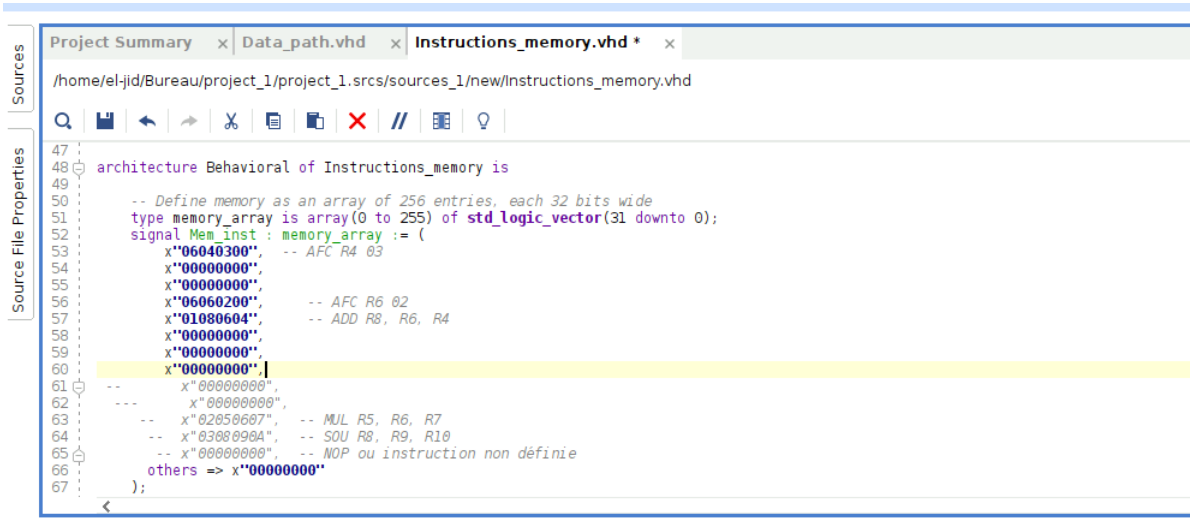


FIGURE 2.1 – Mémoire d'instructions

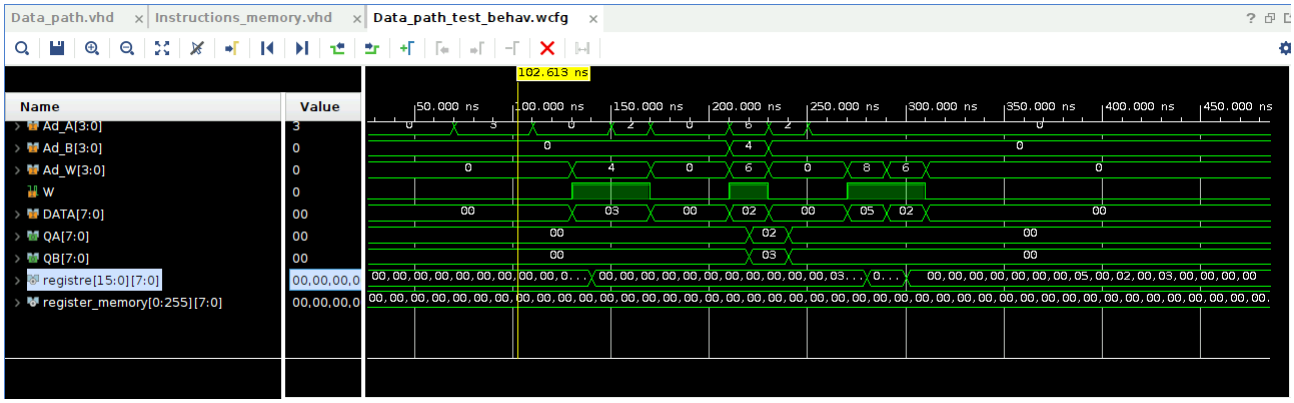


FIGURE 2.2 – Trace chemin des données

Chapitre 3

Conclusion

Ce projet a été extrêmement exigeant et enrichissant. Bien que nous ayons eu une bonne compréhension de l'architecture matérielle, la mise en pratique a nécessité des compétences multidisciplinaires dans divers langages et logiciels, ce qui a demandé une adaptation et une réflexion approfondie sur les matières informatiques. Chaque étape franchie, comme l'implémentation du pipeline, nous a apporté une grande satisfaction, nécessitant une solide compréhension de l'architecture de notre compilateur. Cependant, il a parfois été frustrant de devoir revenir sur des parties déjà implémentées, comme le YACC, lors du changement d'orientation entre registre et mémoire. Malgré ces défis, notre persévérance à poursuivre le projet et à surmonter les obstacles a été extrêmement gratifiante, reflétant les réalités actuelles des entreprises. Nous tenons à exprimer notre gratitude à M. Alata pour son soutien inestimable lors des séances de TP, en particulier durant les périodes de doute où son encouragement bienveillant nous a été d'une grande aide.