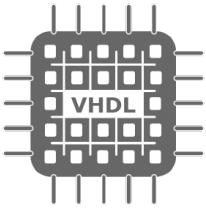
Université Euro Méditerranéenne Fès
EuroMed University of Fez

Ecole d'Ingénierie Digitale et d'Intelligence Artificielle
(EIDIA)

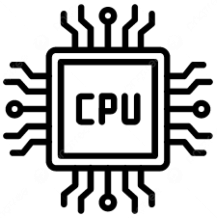


Projet de Fin de module

Filière : Robotique & Cobotique

Semestre : 7

Module : Circuit Logiques Programmables et Programmation VHDL



Thème :

La conception d'un processeur RISC 16 bits utilisant VHDL

Encadré par :

Pr. AMMOUR Alae

Préparé par les étudiants:

- **Bouarfa LAHMAR**
- **Mohamed BOFARHA**

Année Universitaire : 2024-2025

Table des Matières

1. Introduction

1.1 Contexte et Motivation

La conception de processeurs est un domaine clé de l'ingénierie électronique et informatique, avec des applications dans de nombreux systèmes embarqués, informatiques et industriels. Le processeur RISC (Reduced Instruction Set Computing) est une architecture qui simplifie l'exécution des instructions et améliore l'efficacité en termes de vitesse et de consommation d'énergie. Le choix d'un processeur RISC 16 bits repose sur le besoin d'une architecture suffisamment simple pour une implémentation dans des systèmes embarqués tout en restant performante pour des applications spécifiques. La motivation de ce projet est de concevoir et d'implémenter un processeur RISC 16 bits en utilisant VHDL, un langage de description matériel, afin de mieux comprendre les principes de conception des processeurs et d'acquérir des compétences pratiques en VHDL.

1.2 Objectifs du Projet

Le projet a pour objectif de concevoir un processeur RISC 16 bits en utilisant VHDL. Cela inclut :

- La définition d'un ensemble d'instructions adapté aux besoins du processeur.
- La conception de l'architecture matérielle, incluant les composants essentiels tels que l'ALU (Unité Arithmétique et Logique), le décodeur d'instructions et l'unité de contrôle...La mise en œuvre des instructions arithmétiques, logiques, de contrôle et de mémoire.
- La simulation et validation de la conception à l'aide d'outils de simulation pour assurer son bon fonctionnement.

1.3 Portée du Projet

Ce projet vise à concevoir et implémenter un processeur RISC 16 bits simplifié, dans un cadre éducatif, afin de mettre en pratique les concepts théoriques abordés durant la formation. Il a pour objectif de consolider la maîtrise des langages de description matérielle, en particulier le VHDL, ainsi que de renforcer la compréhension des architectures des systèmes embarqués. Le développement sera entièrement réalisé en VHDL, et la validation fonctionnelle sera assurée par des simulations à l’aide de l’environnement ModelSim.

2. Méthodologie de Conception :

2.1 Vue d'ensemble de l'ensemble d'instructions :

L'ensemble d'instructions (ISA, Instruction Set Architecture) définit l'ensemble des opérations que le processeur est capable d'exécuter. Pour notre processeur RISC 16 bits, l'ISA est conçu pour être simple, efficace et adapté à des applications éducatives. Il se compose principalement des catégories d'instructions suivantes :

- **Instructions arithmétiques et logiques** : Ces instructions permettent d'effectuer des opérations de base telles que l'addition, la soustraction, ainsi que des opérations logiques comme AND, OR, XOR et NOT.
- **Instructions de contrôle et de saut** : Ces instructions permettent de transférer des données entre la mémoire et les registres du processeur. Elles incluent des opérations de chargement (LDR - Load Register) pour lire des données depuis la mémoire, et de stockage (STR - Store Register) pour écrire des données en mémoire.

2.2 Flux de conception et processus :

Le processus de conception suit une approche structurée :

1. **Analyse des exigences** : Définition des spécifications du processeur et des fonctionnalités requises.
2. **Conception modulaire** : Division du processeur en sous-systèmes (ALU, décodeur, unité de contrôle, etc.).
3. **Implémentation en VHDL** : Codage de chaque module en VHDL.
4. **Simulation et tests** : Validation des modules à l'aide de bancs de test VHDL.

2.3 Approche de conception modulaire :

La conception modulaire consiste à diviser le projet en sous-composants autonomes, chacun étant responsable d'une fonction spécifique du processeur. Cette approche permet de :

- Simplifier la conception et la gestion du projet.
- Faciliter les tests unitaires de chaque module.
- Assurer la réutilisabilité du code pour des projets futurs.

2.4 Tests et Validation :

Chaque module est testé individuellement avant d'être intégré dans l'architecture globale. Les simulations sont réalisées dans ModelSim pour vérifier le comportement du processeur et garantir sa conformité aux spécifications.

3. Architecture de l'Ensemble d'Instructions :

3.1 Vue d'ensemble de l'ensemble d'instructions :

L'ensemble d'instructions (ISA, Instruction Set Architecture) décrit les opérations que le processeur peut exécuter. Pour ce processeur RISC 16 bits, chaque instruction est représentée par un opcode unique. Le tableau des instructions (voir tableau fourni) regroupe toutes les instructions disponibles. Elles sont classées en trois catégories principales :

1. **Instructions arithmétiques et logiques** : Effectuent des calculs et des manipulations logiques sur les données.
2. **Instructions de contrôle et de saut** : Permettent de gérer le flux d'exécution.
3. **Instructions de chargement et de stockage** : Gèrent le transfert de données entre la mémoire et les registres.

ALU_Opcode	Opération
0000	ADD (Addition)
0001	SUB (Soustraction)
0010	NOT (Non)
0011	AND (ET)
0100	OR (OU)
0101	XOR (OU exclusif)
0110	LSL (Décalage gauche logique)
0111	LSR (Décalage droit logique)
1000	CMP (Comparaison)
1001	B (Branchement)
1010	BEQ (Branchement si égal)
1011	IMM (Immédiat)
1100	LDR (Chargement)
1101	STR (Stockage)

3.2 Instructions arithmétiques et logiques

Les instructions arithmétiques et logiques sont essentielles pour effectuer des calculs et manipuler des données au niveau binaire. Elles permettent au processeur de réaliser des opérations de base sur les données stockées dans les registres. Voici les principales opérations :

- **ADD (Addition)** :
  Cette instruction additionne deux valeurs provenant de registres ou d'une valeur immédiate et un registre. Le résultat est stocké dans un registre de destination.
  Exemple : ADD R1, R2, R3 (R1 = R2 + R3).
- **SUB (Soustraction)** :
  Cette instruction soustrait une valeur d'une autre. Elle peut être utilisée entre deux registres ou entre un registre et une valeur immédiate.
  Exemple : SUB R1, R2, R3 (R1 = R2 - R3).
- **NOT (Non logique)** :
  Cette instruction effectue une inversion binaire (complément à un) sur la valeur d'un registre.
  Exemple : NOT R1, R2 (R1 = ~R2).
- **AND (ET logique)** :
  Cette instruction effectue une opération logique ET entre deux valeurs. Chaque bit du résultat est mis à 1 si les bits correspondants des deux opérandes sont à 1.
  Exemple : AND R1, R2, R3 (R1 = R2 & R3).

- **OR (OU logique)** :
  Cette instruction effectue une opération logique OU entre deux valeurs. Chaque bit du résultat est mis à 1 si au moins l'un des bits correspondants des deux opérandes est à 1.
  Exemple : OR R1, R2, R3 (R1 = R2 | R3).

- **XOR (OU exclusif)** :
  Cette instruction effectue une opération logique OU exclusif entre deux valeurs. Chaque bit du résultat est mis à 1 si les bits correspondants des deux opérandes sont différents.
  Exemple : XOR R1, R2, R3 (R1 = R2 ^ R3).

- **LSL (Décalage gauche logique)** :
  Cette instruction décale les bits d'un registre vers la gauche, en remplissant les bits de droite avec des zéros. Cela équivaut à une multiplication par 2 pour chaque décalage.
  Exemple : LSL R1, R2, #2 (R1 = R2 << 2).

- **LSR (Décalage droit logique)** :
  Cette instruction décale les bits d'un registre vers la droite, en remplissant les bits de gauche avec des zéros. Cela équivaut à une division entière par 2 pour chaque décalage.
  Exemple : LSR R1, R2, #2 (R1 = R2 >> 2).

- **CMP (Comparaison)** :
  Cette instruction compare deux valeurs en soustrayant la seconde de la première, sans stocker le résultat. Elle met à jour les drapeaux de condition (flags) pour indiquer si les valeurs sont égales, supérieures ou inférieures.
  Exemple : CMP R1, R2.

3.3 Instructions de contrôle et de saut

Les instructions de contrôle et de saut permettent de modifier le flux d'exécution du programme en fonction de conditions spécifiques. Elles sont essentielles pour implémenter des boucles, des conditions et des appels de fonctions.

- **B (Branchement inconditionnel)** :
  Cette instruction provoque un saut inconditionnel vers une adresse spécifiée. Elle est utilisée pour transférer le contrôle à une autre partie du programme.

- **BEQ (Branchement si égal)** :
  Cette instruction provoque un saut conditionnel si les deux valeurs comparées par une instruction précédente (comme CMP) sont égales. Elle est utilisée pour implémenter des conditions.

3.4 Instructions de chargement et de stockage

Les instructions de chargement et de stockage permettent de transférer des données entre la mémoire et les registres. Elles sont cruciales pour accéder aux données stockées en mémoire et les manipuler.

- **LDR (Chargement)** :
  Cette instruction charge une valeur depuis la mémoire dans un registre. Elle est utilisée pour accéder aux données stockées à une adresse spécifique.
  Exemple : LDR R1, [R2] (Charge la valeur à l'adresse contenue dans R2 dans R1).

- **STR (Stockage)** :
  Cette instruction stocke une valeur d'un registre dans la mémoire. Elle est utilisée pour sauvegarder des données à une adresse spécifique.
  Exemple : STR R1, [R2] (Stocke la valeur de R1 à l'adresse contenue dans R2).

3.5 Instructions immédiates (IMM) :

Les instructions immédiates permettent d'utiliser des valeurs constantes directement dans les opérations, sans avoir à les charger depuis la mémoire.

- **IMM (Valeur immédiate)** :
  Cette instruction charge une valeur constante directement dans un registre. Elle est utile pour initialiser des registres avec des valeurs fixes.
  Exemple : IMM R1, #42 (Charge la valeur 42 dans R1).

3.6 Format des Instructions et Encodage :

Le tableau ci-dessous présente l'ensemble des instructions supportées par le processeur RISC 16 bits, détaillant leur format, leur implémentation, le bit de condition associé et leur code opération (OPCODE). Cette structure permet de comprendre comment chaque instruction est encodée et exécutée par le processeur.

Instruction	Format	Implémentation	Bit de condition (c)	OPCODE
ADD	RRR	rD = rM + rN	1/0 = signé/non signé	0000
SUB	RRR	rD = rM - rN	1/0 = signé/non signé	0001
NOT	RRU	rD = not rN	N/A	0010
AND	RRR	rD = rM and rN	N/A	0011
OR	RRR	rD = rM or rN	N/A	0100
XOR	RRR	rD = rM xor rN	N/A	0101
LSL	RRI(5)	rD = rM << rN	N/A	0110
LSR	RRI(5)	rD = rM >> rN	N/A	0111
CMP	RRR	rD = cmp(rM, rN)	1/0 = signé/non signé	1000
B	UI(8)	PC = rM ou immédiat 8 bits	1/0 = rM/immédiat 8 bits	1001
BEQ	URR	PC = rM si condition sur rN	N/A	1010
IMMEDIATE	RI(8)	rD = immédiat 8 bits	1/0 = bits supérieurs/inf.	1011
LD	RRU	rD = mémoire(rM)	N/A	1100
ST	URR	mémoire(rM) = rN	N/A	1101

Termes :

- **R :** Registre
- **U :** Non utilisé (*Unused*)
- **rD :** Registre de destination
- **rM :** Premier registre opérande
- **rN :** Deuxième registre opérande
- **c :** Bit de condition
- **I(8)/(5) :** Donnée immédiate sur 8 ou 5 bits

Le tableau suivant montre comment les bits sont organisés pour chaque format d'instruction du processeur :

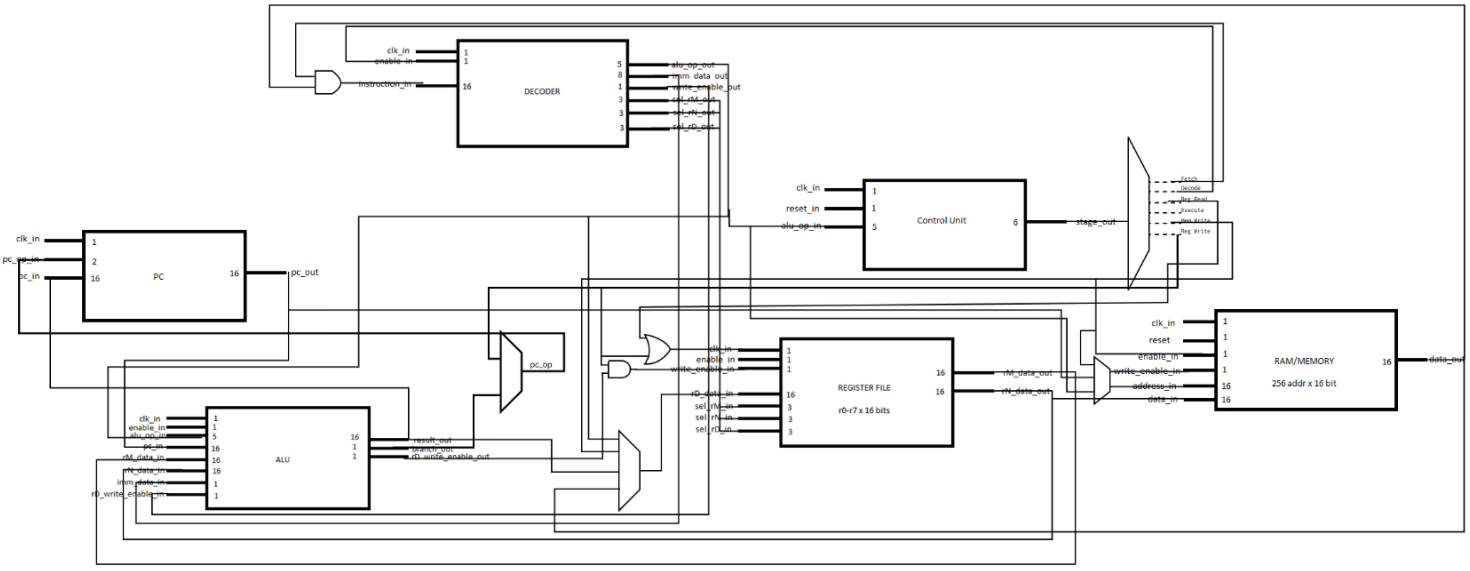
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RRR	**C**	**opcode**				**rD**			**rM**			**rN**			**U**	
RRU	**C**	**opcode**				**rD**			**rM**			**U**				
URR	**C**	**opcode**				**U**			**rM**			**rN**			**U**	
RRI(5)	**C**	**opcode**				**rD**			**rM**			**5-bit immediate**				
RI(8)	**C**	**opcode**				**rD**			**8-bit immediate**							
UI(8)	**C**	**opcode**				**U**			**8-bit immediate**							

- **RRR** : Opération entre trois registres.
- **RRU** : Opération entre deux registres, un champ inutilisé.
- **URR** : Opération entre deux registres, un champ inutilisé.
- **RRI(5)** : Opération entre deux registres et une valeur immédiate de 5 bits.
- **RI(8)** : Opération entre un registre et une valeur immédiate de 8 bits.
- **UI(8)** : Opération avec une valeur immédiate de 8 bits, champ registre inutilisé.

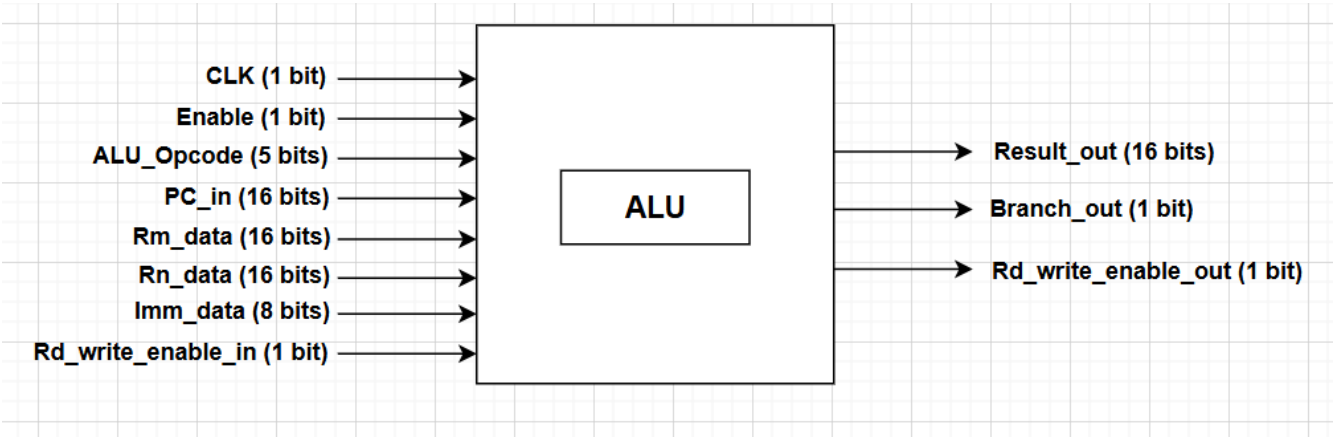
4. Architecture du Processeur :

4.1 Architecture générale du processeur :

Pour réaliser ce processeur RISC 16 bits, nous avons adopté une architecture modulaire et simplifiée, conforme aux principes RISC. Cette architecture est composée de plusieurs unités fonctionnelles interconnectées, chacune dédiée à une tâche spécifique, afin d'assurer une exécution efficace et optimisée des instructions.



4.2 Unité Arithmétique et Logique (ALU) :



Entrées de l'ALU :

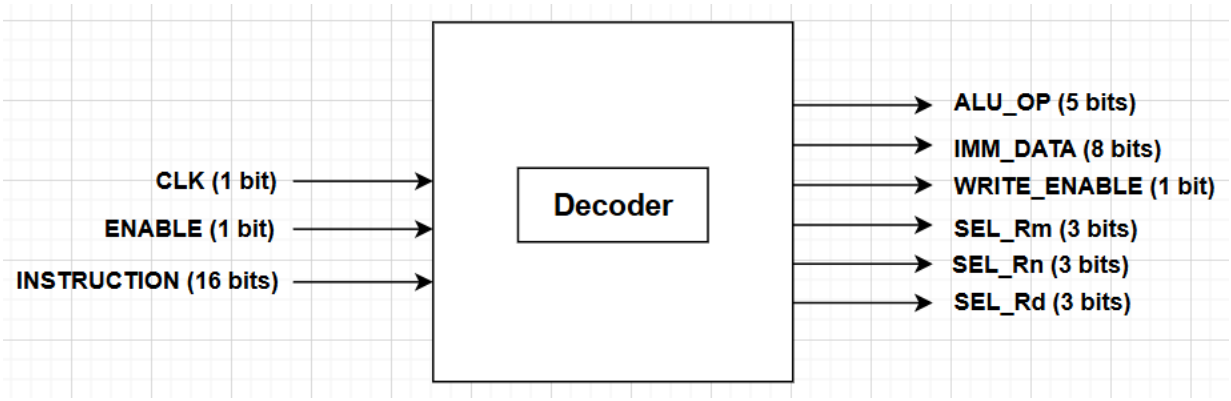
1. **CLK** : Signal d'horloge pour synchroniser les opérations.
2. **Enable** : Active l'ALU lorsqu'il est à '1'.
3. **ALU_Opcode** : Code d'opération (5 bits) qui détermine l'opération à effectuer (addition, soustraction, etc.).
4. **PC_in** : Valeur du compteur de programme (16 bits) utilisée pour les opérations de branchement.
5. **Rm_data** : Donnée du premier registre opérande (16 bits).
6. **Rn_data** : Donnée du deuxième registre opérande (16 bits).
7. **Imm_data** : Donnée immédiate (8 bits) pour les opérations utilisant une valeur constante.
8. **Rd_write_enable_in** : Signal indiquant si le résultat doit être écrit dans un registre de destination.

Sorties de l'ALU :

1. **Result_out** : Résultat de l'opération (16 bits).
2. **Branch_out** : Signal indiquant si une instruction de branchement doit être exécutée.
3. **Rd_write_enable_out** : Signal indiquant si le résultat doit être écrit dans un registre de destination

L'ALU exécute des opérations arithmétiques, logiques, de décalage et de branchement en fonction de l'opcode, et génère un résultat ainsi que des signaux de contrôle pour guider l'exécution des instructions.

4.3 Décodeur d'instructions :



Entrées :

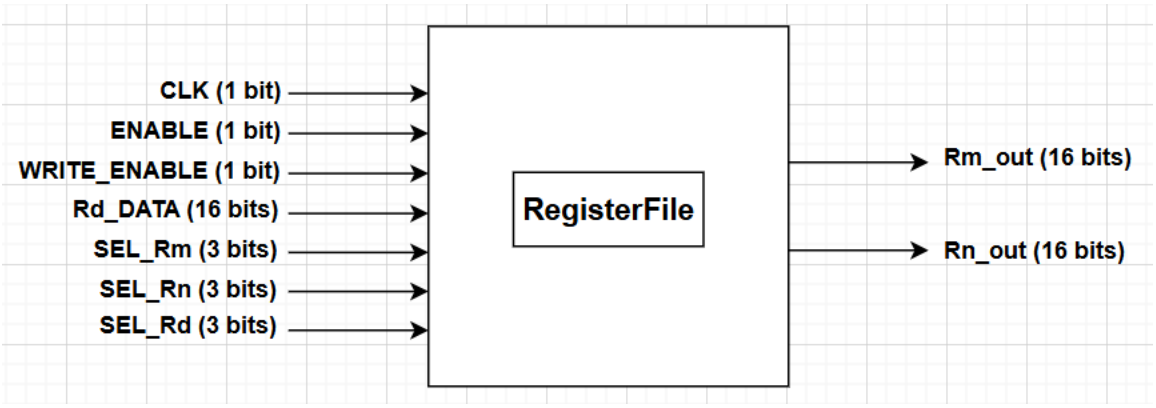
- **INSTRUCTION** : L'instruction de 16 bits à décoder.
- **CLK** et **ENABLE** : Synchronisation et activation du décodeur.

Sorties :

- **ALU_OP** : Code d'opération pour l'ALU.
- **SEL_Rm**, **SEL_Rn**, **SEL_Rd** : Sélection des registres.
- **IMM_DATA** : Donnée immédiate extraite de l'instruction.
- **WRITE_ENABLE** : Signal indiquant si un registre doit être écrit.

Le décodeur interprète l'instruction et extrait les informations nécessaires pour exécuter l'opération demandée, en générant les signaux de contrôle appropriés.

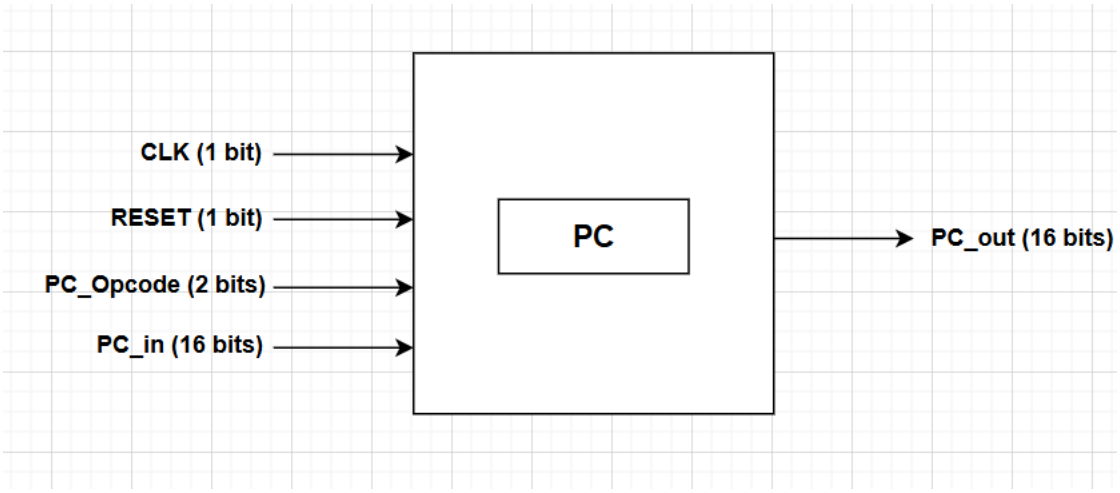
4.4 Registre de fichiers :



- **Entrées** :
  - **SEL_Rm**, **SEL_Rn**, **SEL_Rd** : Sélection des registres à lire ou à écrire.
  - **Rd_DATA** : Donnée à écrire dans le registre cible.
  - **WRITE_ENABLE** : Active l'écriture dans le registre cible.
  - **CLK** et **ENABLE** : Synchronisation et activation du module.
- **Sorties** :
  - **Rm_out**, **Rn_out** : Données lues à partir des registres sélectionnés.

Le RegisterFile stocke et gère les données temporaires du processeur en permettant la lecture et l'écriture dans les registres, en fonction des signaux de contrôle.

4.5 Compteur Programme (PC) :

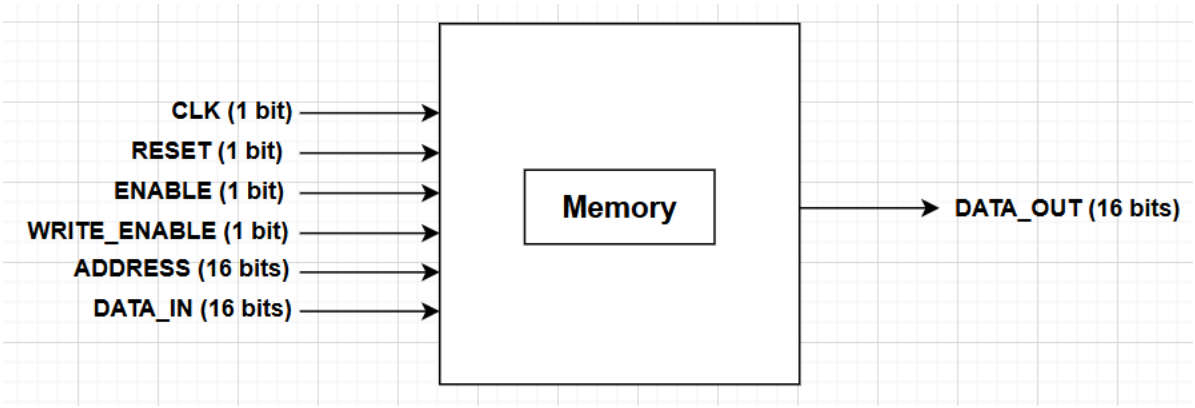


- **Entrées :**
  - **PC_Opcode :** Détermine l'opération à effectuer (réinitialisation, incrémentation, branchement, ou aucune opération).
  - **PC_in :** Nouvelle adresse à charger en cas de branchement.
  - **CLK et RESET :** Synchronisation et réinitialisation du compteur.

- **Sorties :**
  - **PC_out :** Adresse de la prochaine instruction à exécuter.

Le PC maintient et met à jour l'adresse de la prochaine instruction, en fonction des opérations demandées (réinitialisation, incrémentation ou branchement).

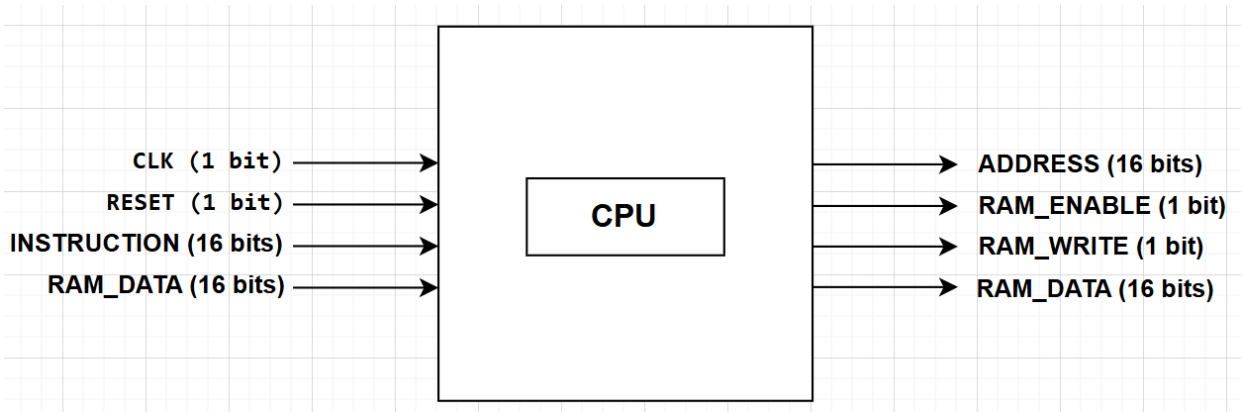
4.7 Mémoire :



- **Entrées :**
  - **ADDRESS :** Adresse mémoire à lire ou à écrire.
  - **DATA_IN :** Donnée à écrire en mémoire.
  - **WRITE_ENABLE :** Active l'écriture en mémoire.
  - **ENABLE :** Active le module mémoire.
  - **CLK et RESET :** Synchronisation et réinitialisation de la mémoire.
- **Sorties :**
  - **DATA_OUT :** Donnée lue à partir de l'adresse spécifiée.

Le module Memory stocke et fournit des données pour le processeur, en permettant des opérations de lecture et d'écriture synchronisées sur l'horloge.

4.8 Unité centrale de traitement (CPU) :



- **Entrées** :
  - **INSTRUCTION** : Instruction de 16 bits à exécuter.
  - **CLK** : Signal d'horloge pour synchroniser les opérations.
  - **RST** : Signal de réinitialisation pour remettre le CPU dans un état initial.
- **Sorties** :
  - **ADDRESS** : Adresse mémoire à lire ou à écrire.
  - **RAM_ENABLE** : Active l'accès à la mémoire.
  - **RAM_WRITE** : Active l'écriture en mémoire.
  - **RAM_DATA** : Données bidirectionnelles pour la lecture/écriture en mémoire.

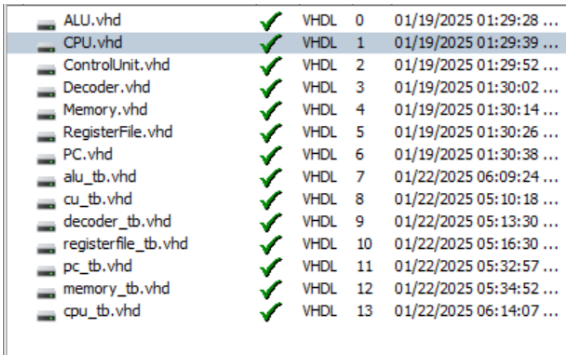
Le module CPU coordonne l'exécution des instructions en interconnectant les sous-modules (PC, ALU, Decoder, RegisterFile, Memory) pour gérer le flux de données, effectuer des calculs, et interagir avec la mémoire, tout en synchronisant les opérations sur le signal d'horloge.

5. Implémentation de la Conception

L'implémentation de la conception s'est déroulée en plusieurs étapes, incluant le développement modulaire, l'intégration, et la validation par simulation. Chaque étape a été réalisée avec soin pour garantir un processeur fonctionnel et conforme aux spécifications initiales.

5.1 Codage VHDL pour les Modules

Tous les composants de l'architecture décrite dans la section précédente ont été développés individuellement en VHDL. Chaque unité fonctionnelle, comme l'ALU, l'unité de contrôle (CU) … a été implémentée sous forme de modules distincts. Cette approche modulaire a permis de simplifier le processus de conception et de faciliter le test de chaque composant indépendamment avant leur intégration.

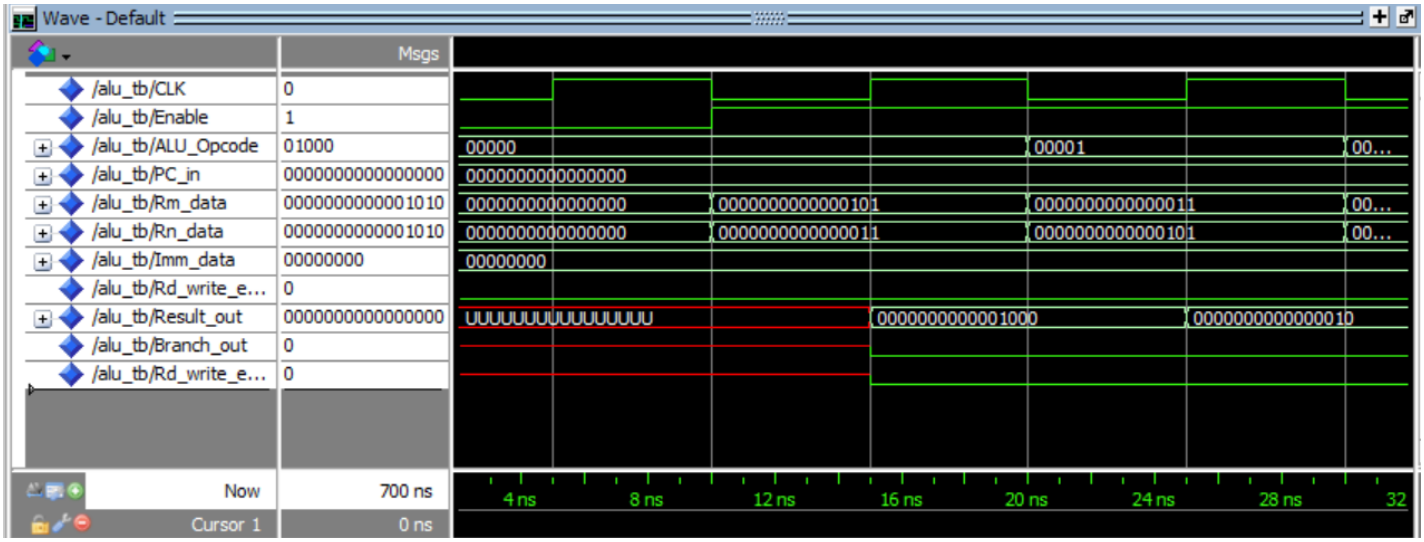


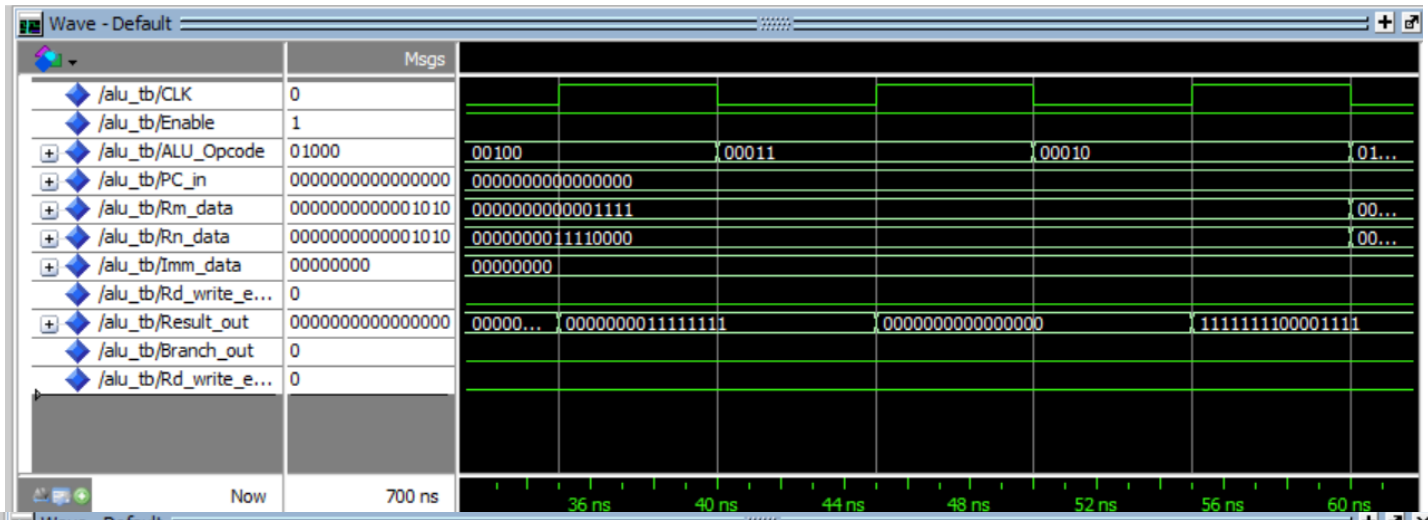
5.2 Intégration des Modules

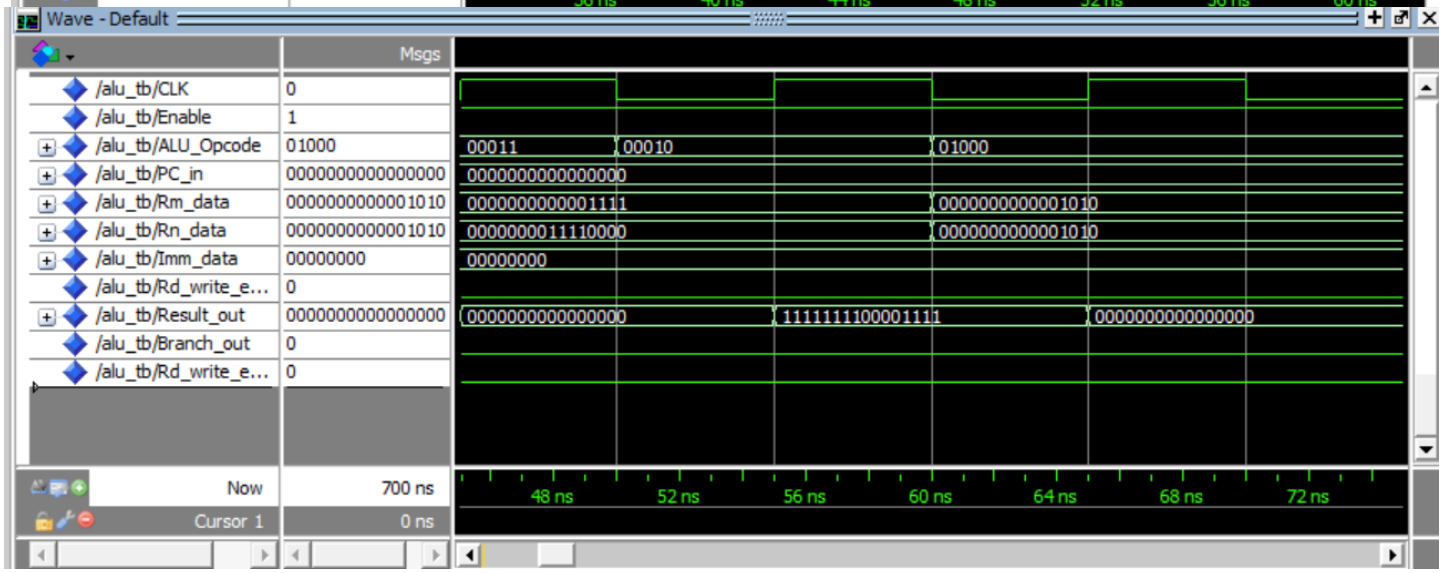
Après avoir codé et validé chaque module individuellement, les composants ont été intégrés dans un fichier principal appelé **cpu.vhdl**, qui représente l'unité centrale de traitement complète. Cette étape a consisté à connecter les différents modules entre eux en respectant le flux de données et les signaux de contrôle définis dans l'architecture. L'intégration a permis de vérifier que les modules interagissent correctement pour exécuter les instructions du processeur.

6. Résultats et Analyse

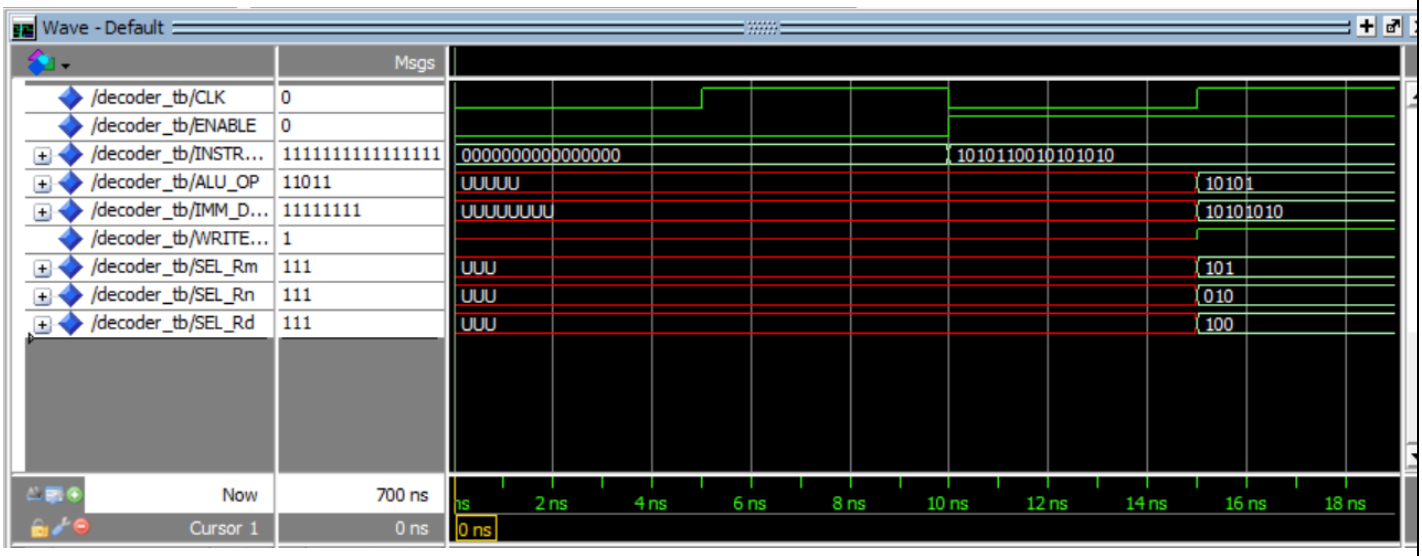
6.1 Testbench de l'ALU :

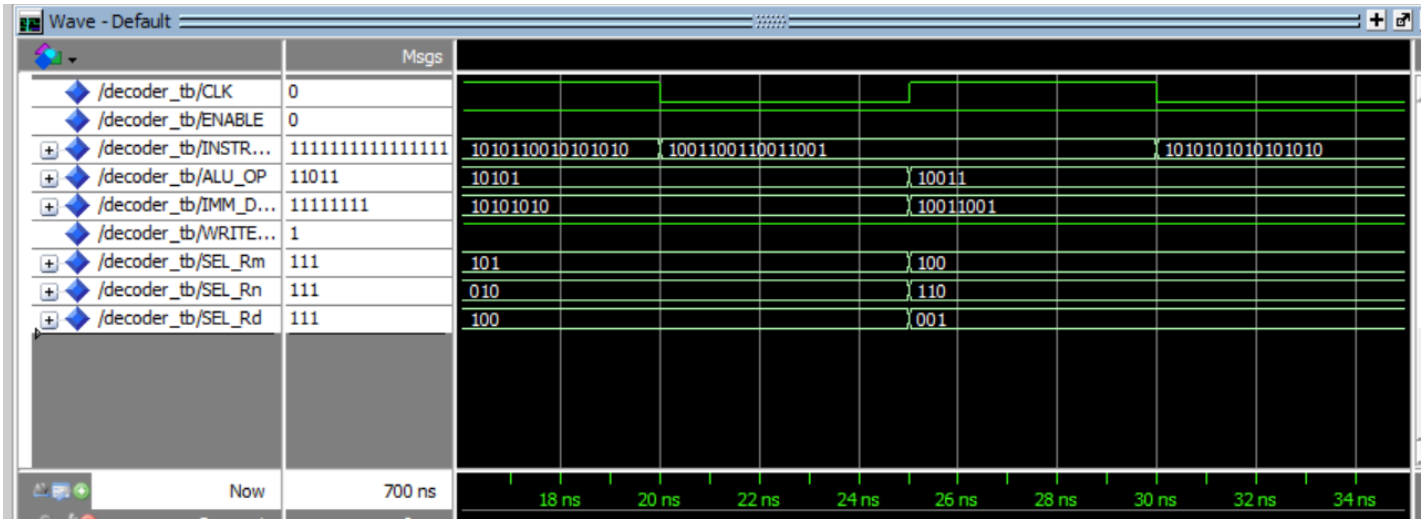


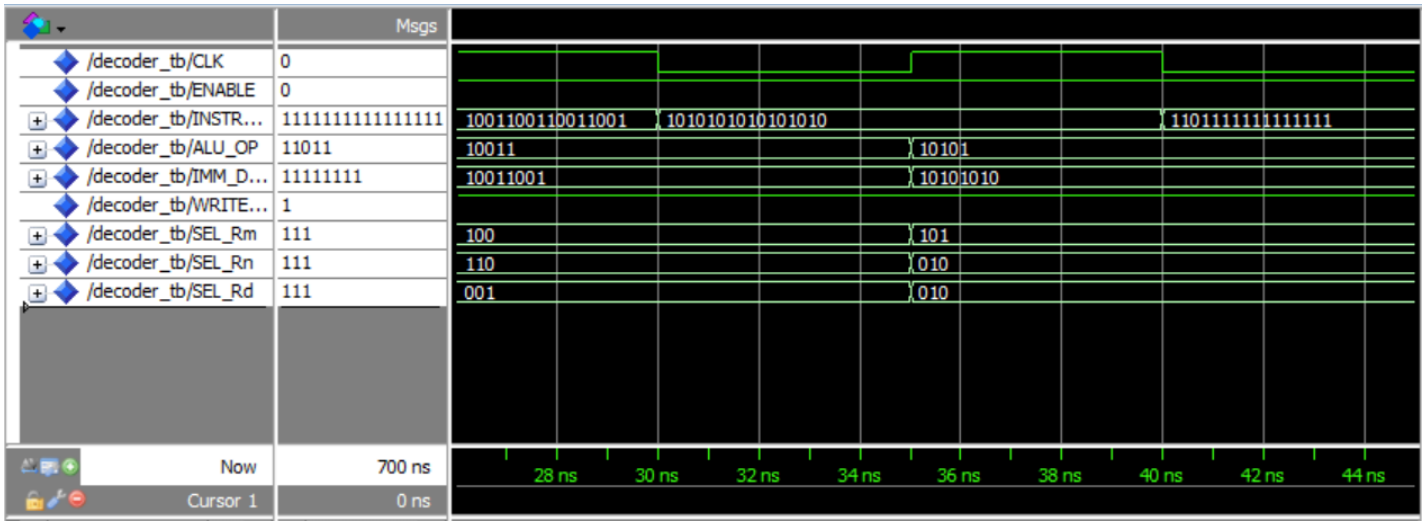




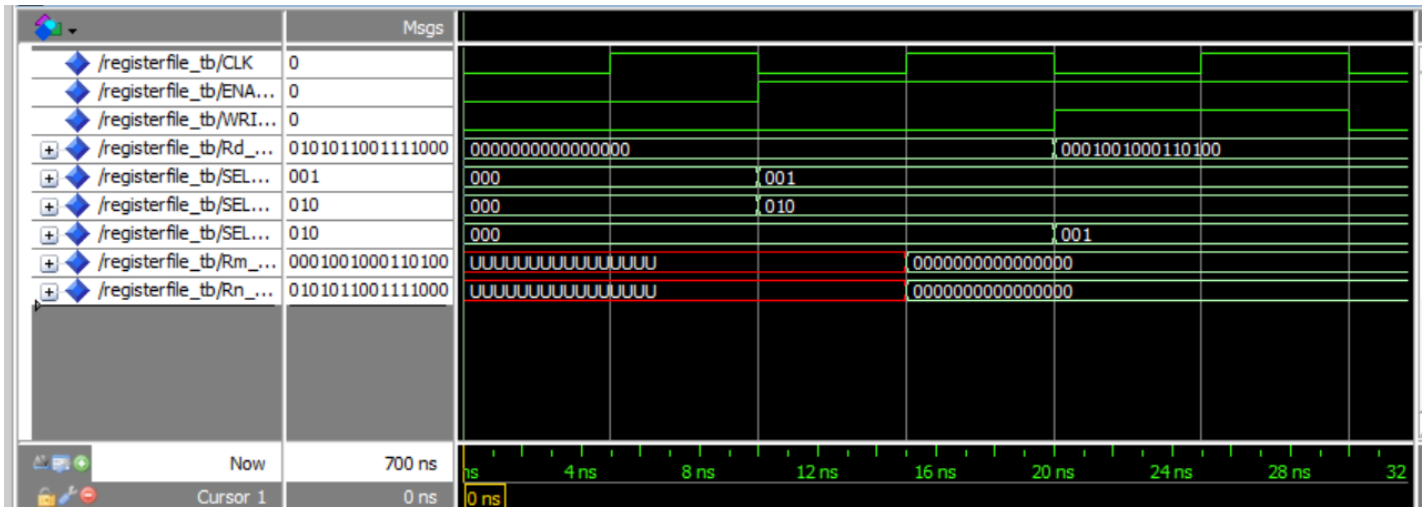
6.2 Testbench du Decoder

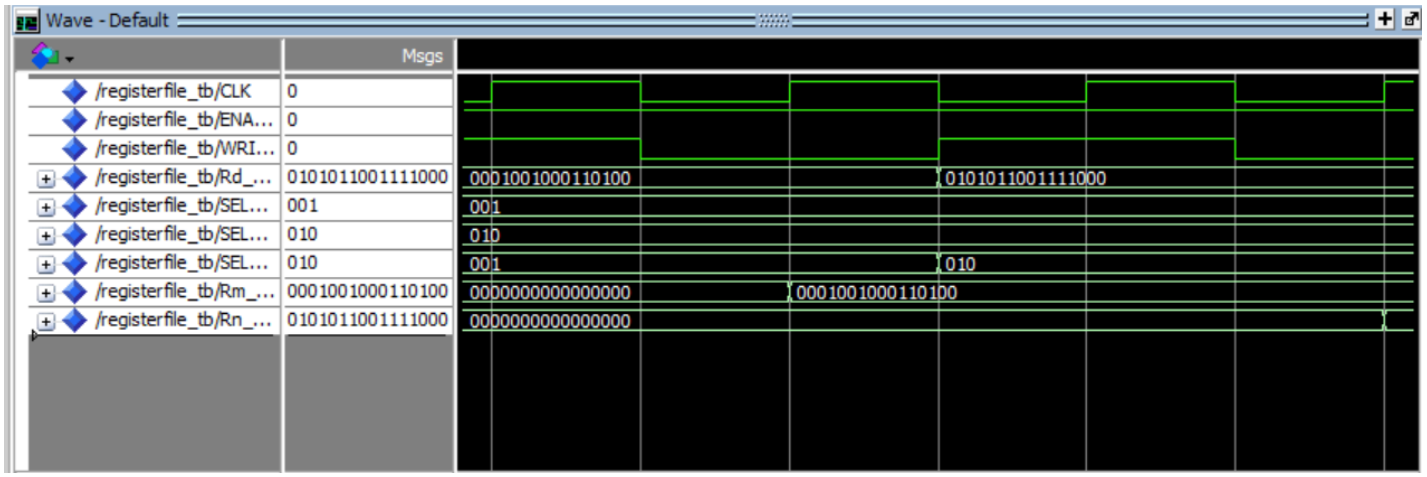




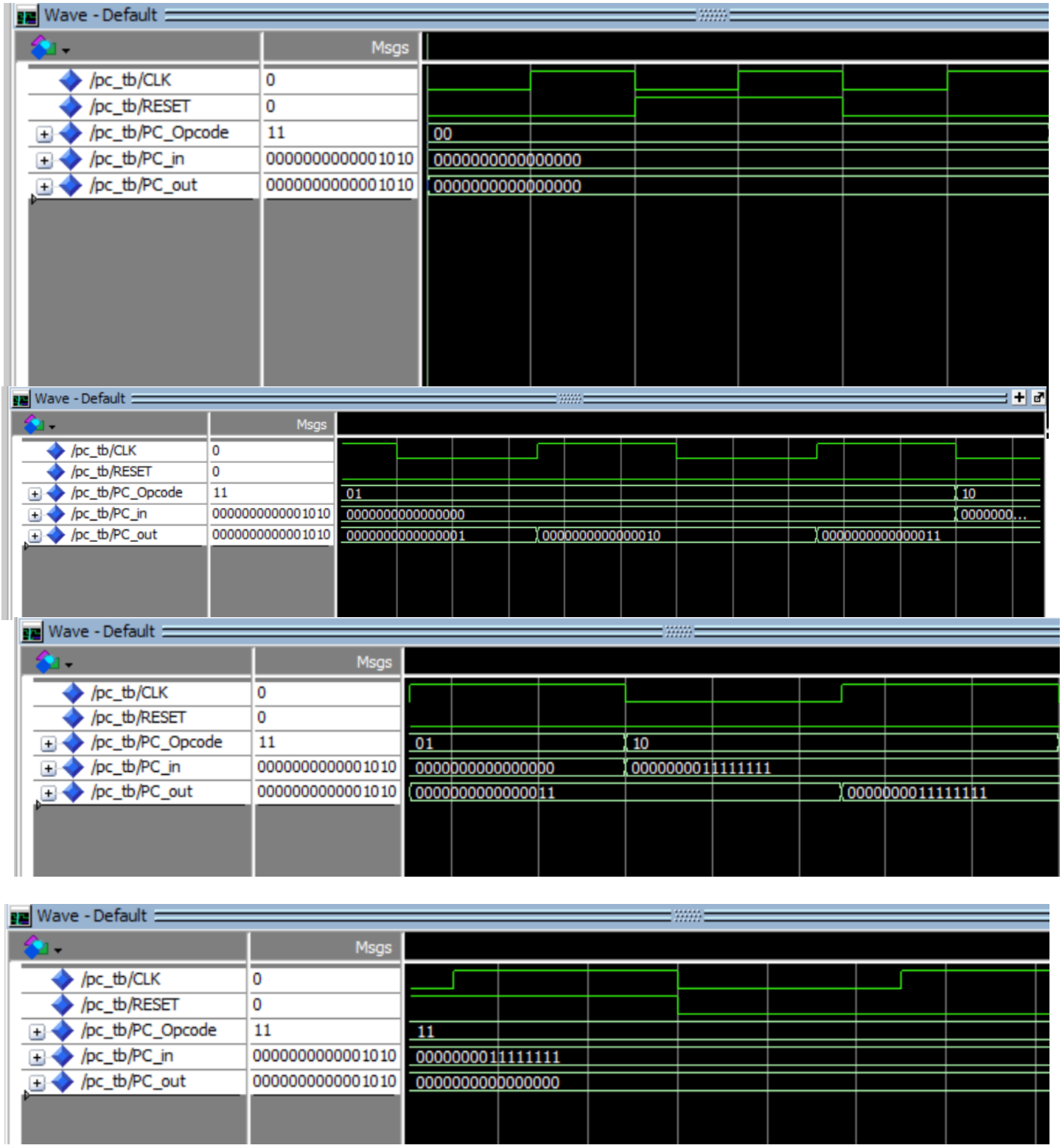


6.3 Testbench du RegisterFile :

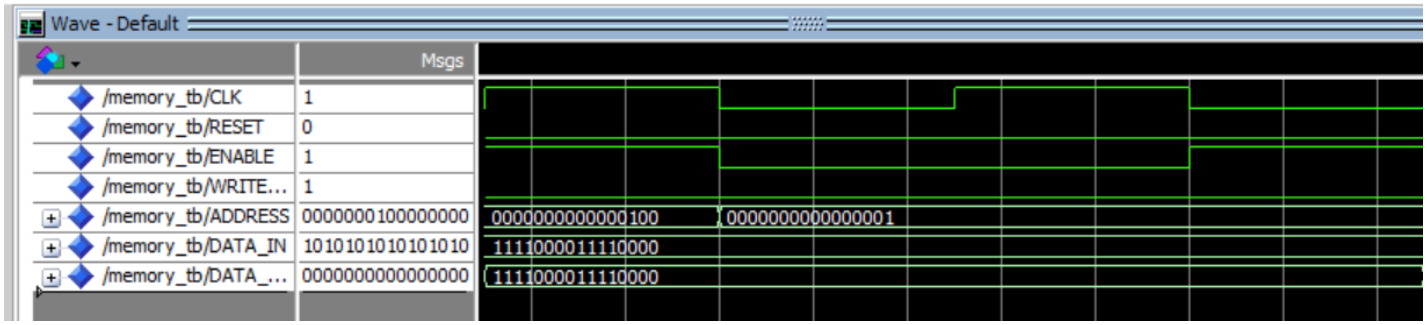


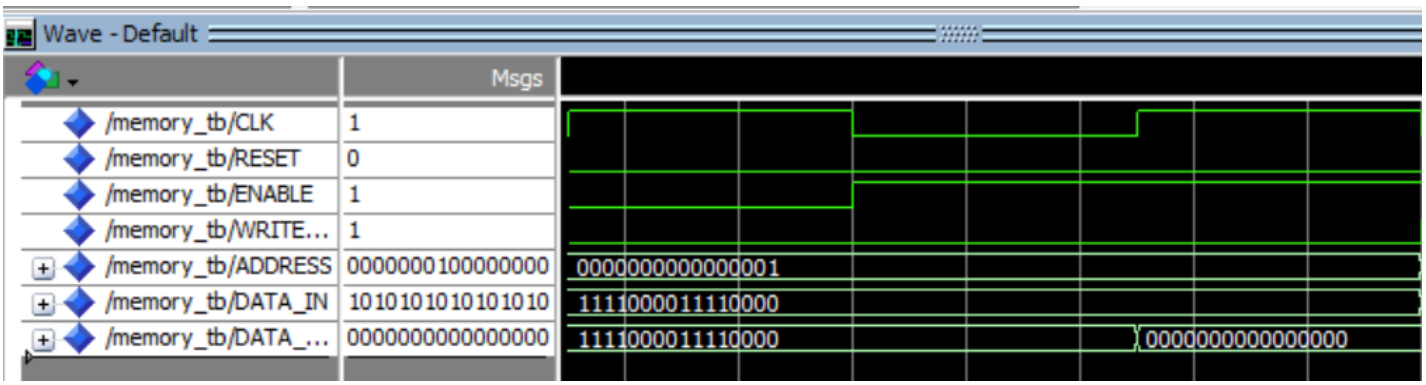


6.4 Testbench du PC (Compteur Programme) :

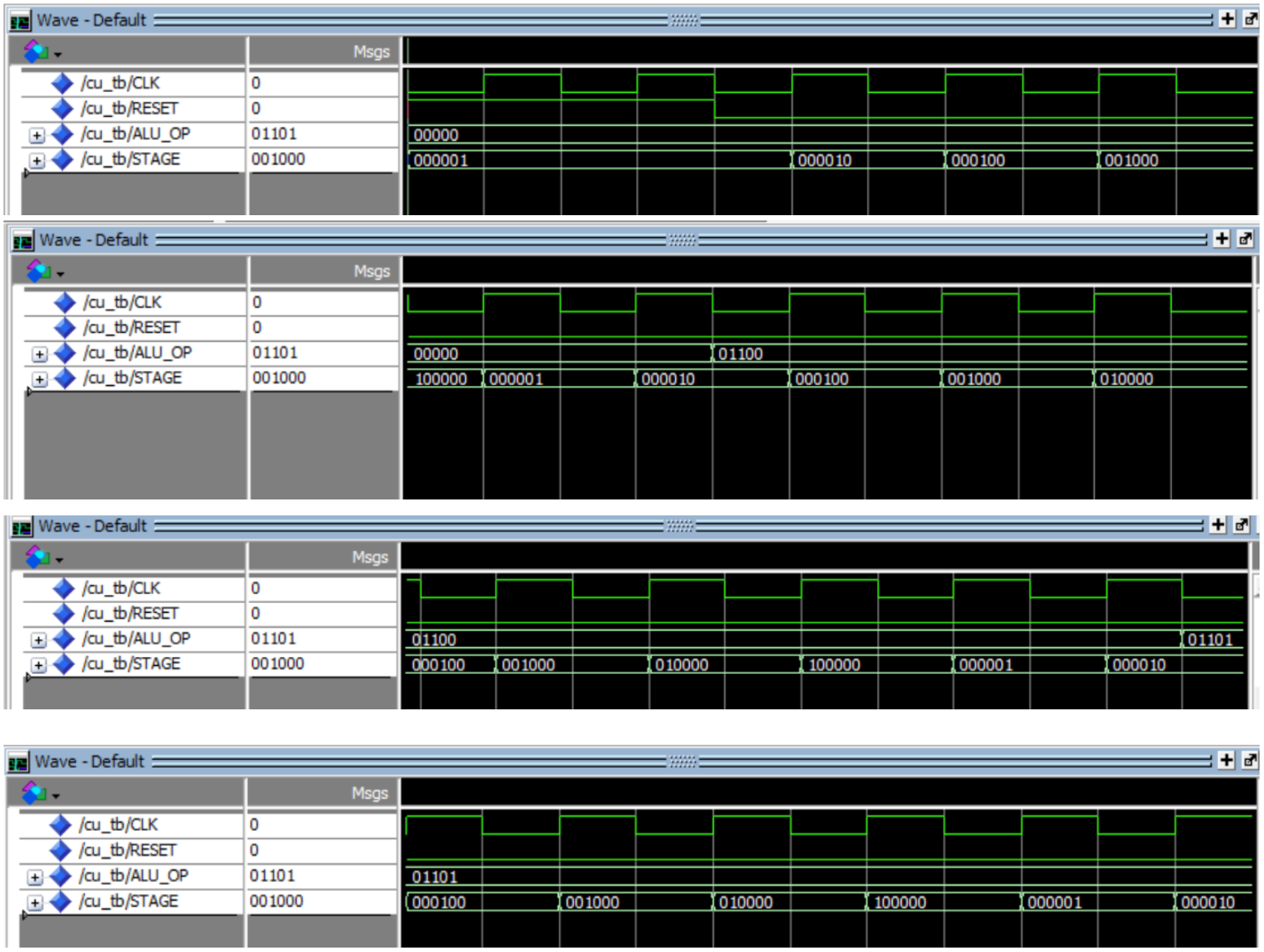


6.5 Testbench de la Memory :

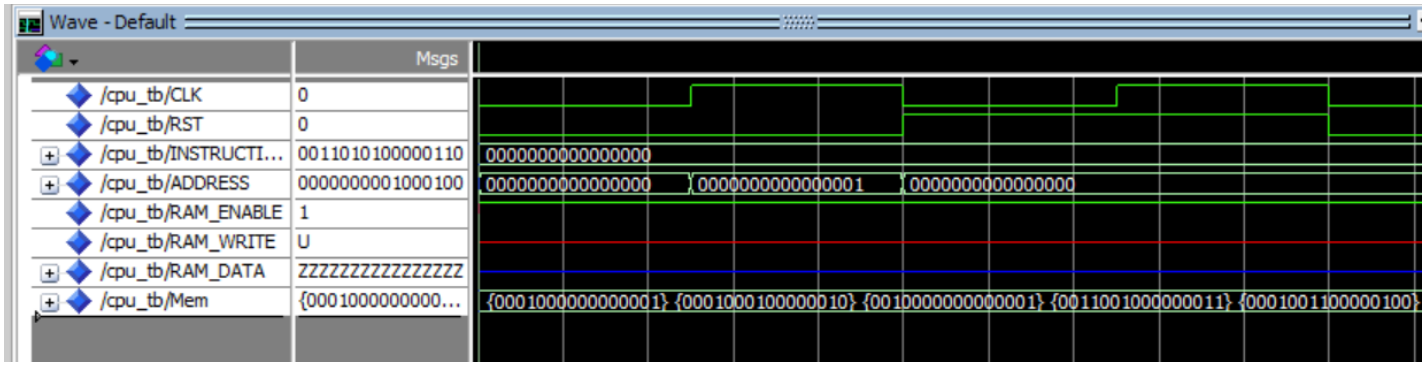


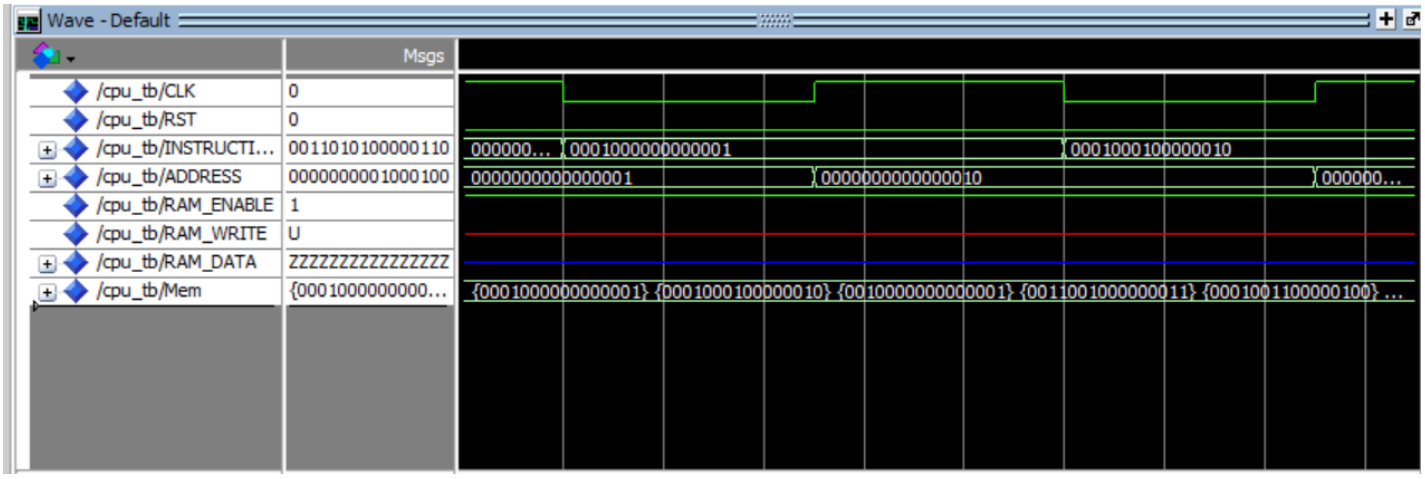


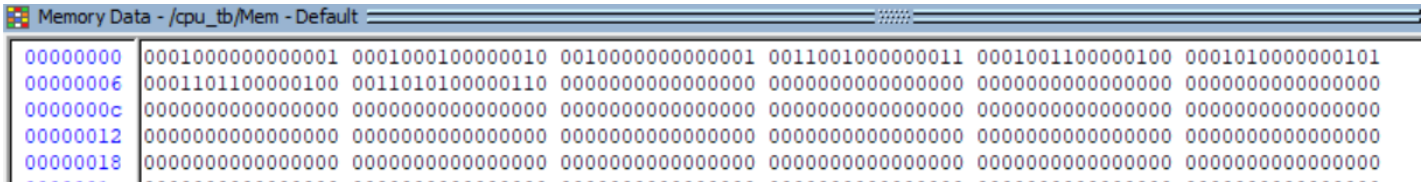
6.6 **Testbench de ControlUnit :**



6.7 Testbench du CPU :







Programme 1 : Addition de deux nombres

Instructions :

1. **LOAD A : Charge la valeur 00000001 (1 en décimal) dans le registre A.**
2. **LOAD B : Charge la valeur 00000010 (2 en décimal) dans le registre B.**
3. **ADD A, B : Additionne A et B, résultat dans C.**
4. **STORE C : Stocke le résultat à l'adresse mémoire 3.**

Programme 2 : Opération logique AND

Instructions :

1. **LOAD X** : Charge la valeur 00000100 (4 en décimal) dans le registre X.
2. **LOAD Y** : Charge la valeur 00000101 (5 en décimal) dans le registre Y.
3. **AND X, Y** : Effectue l'opération logique AND entre X et Y, résultat dans un registre temporaire.
4. **STORE Result** : Stocke le résultat à l'adresse mémoire 6.

Résultats :

Programme 1 :

- **Adresse 0** : 0001000000000001 (LOAD A).
- **Adresse 1** : 0001000100000010 (LOAD B).
- **Adresse 2** : 0010000000000001 (ADD A, B).
- **Adresse 3** : 0011001000000011 (STORE C) → Résultat attendu : 00000011 (3 en décimal).

Programme 2 :

- **Adresse 4** : 0001001100000100 (LOAD X).
- **Adresse 5** : 0001010000000101 (LOAD Y).
- **Adresse 6** : 0001101100000100 (AND X, Y).
- **Adresse 7** : 0011010100000110 (STORE Result) → Résultat attendu : 00000100 (4 en décimal).

7. Conclusion

7.1 Réalisations du projet

Ce projet a permis de concevoir et de simuler un processeur RISC 16 bits fonctionnel en utilisant VHDL. Bien que limité à une simulation logicielle, le processeur a démontré sa capacité à exécuter un ensemble réduit d'instructions, ce qui constitue une étape importante dans la compréhension des concepts fondamentaux de l'architecture des processeurs.

7.2 Difficultés rencontrées

Plusieurs défis ont été rencontrés au cours du projet. Parmi ceux-ci figurent un manque initial de connaissances approfondies sur l'architecture des processeurs, des contraintes de temps importantes pour la réalisation des différentes étapes, des problèmes techniques liés à l'utilisation de ModelSim pour la simulation, des difficultés lors de l'intégration des différents modules, et des complications dans la mise en place et l'exécution des testbenches.

7.3 Travaux futurs et améliorations possibles

Pour améliorer et étendre ce projet, plusieurs perspectives peuvent être envisagées. L'une des améliorations principales serait de transformer le processeur en une architecture 32 bits, permettant ainsi l'exécution d'un jeu d'instructions plus complexe et plus étendu. Cette évolution offrirait une meilleure flexibilité et des performances accrues, rendant le processeur adapté à une plus grande variété d'applications. D'autres améliorations pourraient inclure l'ajout de fonctionnalités avancées comme la gestion de la mémoire cache ou le développement de modes de fonctionnement optimisés pour des tâches spécifiques.