Rapport Architecture Systèmes

-

Processeur mono-cycle

Grégoire MAHON
Armand LELONG
EI2I-3 – II (Groupe B)

README.md

VHDL Monocycle Processor

This repository contains the VHDL implementation of a monocycle processor based on the MIPS (Microprocessor without Interlocked Pipelined Stages) instruction set architecture. This project was created as part of the Polytech Sorbonne EI2I-3 project by Grégoire Mahon and Armand Lelong.

The processor is composed of several components, including a register file, an arithmetic logic unit (ALU), a control unit, an instruction decoder, and a processing unit. Each component is implemented as a separate VHDL entity, allowing for modularity and easy integration.

Components

The following components are used in the processor:

- `processor`: The main component of the processor, integrating all other components.
- `banc_registres`: A 16x32-bit register file that can read and write values to and from registers.
- `ALU`: Arithmetic logic unit performing arithmetic and logic operations on two 32-bit inputs.
- `control_unit`: Generates control signals for the other components based on the instruction code.
- `Instruction_Decoder`: Decodes the incoming instruction and generates control signals for the registers and operators.
- `processing_unit`: A component that integrates the register file, ALU, control unit, and instruction decoder.

Usage

To use the processor, instantiate the `processor` entity and connect it to the other components. The processor is clocked by the `clk` input, and instructions are loaded into the `instr` input.

Testbenches

The processor can be tested using the provided testbenches, which simulate various instruction sequences and verify the output values. Each testbench is named after the instruction sequence it tests.

Learning Outcomes

This project provided us with a deep understanding of the inner workings of a monocycle processor and a strong proficiency in VHDL, a hardware description language widely used in the industry. We learned how to design modular components and integrate them into a larger architecture, understanding how different parts of a system can interact to perform complex operations. This project also allowed us to develop our teamwork and communication skills, essential competencies for any engineer.

Contributors

This project was created by Grégoire Mahon and Armand Lelong [@armagrad]. Please feel free to contribute to the project by creating pull requests or reporting issues.

Figure 1 : Capture d'écran du readme (informations sur le projet, dans la page d'accueil du projet sur GitHub)

Table des matières

Introduction

Dans le domaine en constante évolution de l'informatique, la conception et la réalisation de processeurs jouent un rôle crucial. Les processeurs sont le cœur battant de chaque système informatique, exécutant les instructions qui permettent à nos ordinateurs, smartphones et autres appareils numériques de fonctionner. Dans le cadre de ce projet, nous avons exploré la conception et la réalisation d'un processeur mono-cycle en utilisant le langage de description matériel VHDL (VHSIC Hardware Description Language).

Le VHDL est un langage de programmation utilisé pour décrire les systèmes numériques à différents niveaux d'abstraction. Il est largement utilisé dans l'industrie et l'académie pour la modélisation, la simulation et la synthèse de circuits numériques. Dans ce rapport, nous présentons notre travail sur la conception et la réalisation d'un processeur mono-cycle en utilisant le VHDL.

Un processeur mono-cycle est un type de processeur où chaque instruction est exécutée en un seul cycle d'horloge. Cela contraste avec les processeurs pipelinés, où plusieurs instructions sont exécutées simultanément à différents stades de leur exécution. Bien que les processeurs mono-cycle soient généralement plus simples à concevoir et à comprendre, ils présentent des défis uniques en termes de performance et d'efficacité.

Dans ce rapport, nous décrirons en détail notre approche pour concevoir et réaliser un processeur mono-cycle en VHDL. Nous présenterons notre code, expliquerons les choix techniques que nous avons faits, et discuterons des défis que nous avons rencontrés et comment nous les avons surmontés. Notre objectif est de fournir un aperçu de notre processus de conception et de réalisation, et d'offrir des perspectives sur les leçons que nous avons apprises en cours de route.

L’ensemble des testbenches présentés dans ce rapport sont conçus avec des assertions, ce qui explique qu’il n’y a pas de captures d’écran des graphiques de simulations ModelSim, car les assertions renvoient uniquement des validations sous forme de « PASS » ou « FAIL » (validation ou erreur) dans la console.

L’ensemble des entités contiennent des testbenches et ces derniers ont validé le fonctionnement général de notre processeur monocycle, notamment par l’ensemble des testbenches validés, ainsi qu’un testbench final qui valide l’entité processeur(qui regroupe l’ensemble du processeur monocycle), ne renvoyant aucune erreur.

Unité Arithmétique Logique - UAL.vhd

Lien vers le code de l'UAL (GitHub)

- *Entité : ual*

L'Unité Arithmétique Logique, ou ALU, est un composant essentiel de tout processeur. Elle est responsable de l'exécution de toutes les opérations arithmétiques et logiques qui sont nécessaires pour le traitement des instructions.

L'ALU prend en entrée deux opérandes a et b, ainsi qu'un code d'opération op. En fonction de la valeur de op, l'ALU effectue une addition, une soustraction, ou simplement transmet l'une des entrées à la sortie. Le résultat de l'opération est ensuite renvoyé à travers le port de sortie s.

En outre, l'ALU génère un signal de sortie n qui indique si le résultat de l'opération est négatif. Ce signal est utilisé pour des opérations de comparaison et de saut conditionnel.

La conception de l'ALU est assez simple, mais elle est essentielle pour le fonctionnement du processeur. Elle illustre bien l'importance de la modularité et de l'abstraction dans la conception des systèmes numériques. Chaque composant du processeur a un rôle spécifique à jouer, et en combinant ces composants de manière judicieuse, nous pouvons construire un système complexe et fonctionnel à partir de parties relativement simples.

Banc de Registres - banc_registres.vhd

Lien vers le code du banc de registres (GitHub)

- *Entité : banc_registres*

Le banc de registres est un autre composant essentiel de tout processeur. Il s'agit d'un ensemble de registres qui peuvent être lus et écrits par le processeur pour stocker temporairement des données pendant l'exécution des instructions.

Le banc de registres prend en entrée un signal d'horloge (clk), un signal de réinitialisation (rst), une entrée d'écriture (w), trois adresses (ra, rb, rw) et un signal d'écriture (we). Les adresses sont utilisées pour sélectionner les registres à lire ou à écrire. Si we est '1', alors le registre à l'adresse rw est écrit avec la valeur de w.

En sortie, le banc de registres fournit les valeurs des registres aux adresses ra et rb. Ces valeurs sont utilisées par d'autres parties du processeur pour effectuer des opérations.

Le banc de registres est initialisé à l'aide de la fonction init_banc, qui met tous les registres à 0, à l'exception du dernier (index 15) qui est mis à 0x00000030. Cette initialisation est effectuée à chaque fois que le signal de réinitialisation est '1'.

La conception du banc de registres illustre l'importance de la gestion de l'état dans les systèmes numériques. Les registres permettent au processeur de stocker temporairement des données, ce qui est essentiel pour l'exécution de nombreuses instructions.

En outre, la capacité de réinitialiser le banc de registres à un état connu est importante pour la fiabilité et la prévisibilité du système.

Testbench du banc de registres

Lien vers le code du testbench du banc de registres (GitHub)

Un testbench est un environnement de test conçu pour vérifier le fonctionnement d'un module de conception. Dans le contexte de la conception de circuits numériques, un testbench est généralement un programme qui génère des stimuli pour le dispositif sous test (DUT), vérifie les réponses du DUT et signale toute différence entre les réponses observées et les réponses attendues.

Dans notre projet, le banc de registres est un composant crucial qui doit fonctionner correctement pour que le processeur puisse exécuter des instructions. Par conséquent, il est essentiel de tester le banc de registres pour s'assurer qu'il répond correctement aux entrées et qu'il produit les sorties attendues.

Le testbench commence par déclarer une instance de l'entité banc_registres et la connecter à un ensemble de signaux qui seront utilisés pour tester son comportement. Ces signaux incluent un signal d'horloge (clk), un signal de réinitialisation (rst), une entrée d'écriture (w), trois adresses (ra, rb, rw), un signal d'écriture (we), et deux sorties (a, b).

Le testbench génère ensuite un signal d'horloge en utilisant un processus séparé (clk_gen). Ce signal d'horloge est utilisé pour synchroniser les opérations du banc de registres.

Le comportement du banc de registres est testé à l'aide d'un autre processus (test). Ce processus commence par initialiser le banc de registres en activant le signal de réinitialisation. Il lit ensuite les registres aux adresses 0 et 12 et vérifie qu'ils contiennent tous deux la valeur 0. Ces vérifications sont effectuées à l'aide d'instructions **assert** qui signalent une erreur si la condition spécifiée n'est pas satisfaite.

Le processus de test écrit ensuite la valeur 0xA dans les registres aux adresses 0 et 12. Il vérifie ensuite que ces écritures ont été effectuées correctement en lisant à nouveau les registres et en vérifiant qu'ils contiennent la valeur attendue.

L'importance de ce testbench réside dans sa capacité à vérifier de manière exhaustive et systématique le comportement du banc de registres. En testant le banc de registres avec une variété de stimuli et en vérifiant soigneusement ses réponses, nous pouvons nous assurer que le banc de registres fonctionne correctement dans toutes les conditions prévues. Cela est essentiel pour garantir la fiabilité et la précision du processeur dans son ensemble.

En outre, le testbench fournit une documentation précieuse sur le comportement attendu du banc de registres. En lisant le code du testbench, un autre ingénieur peut comprendre comment le banc de registres est censé fonctionner et comment interagir avec lui. Cela peut être particulièrement utile pour le débogage et la maintenance du processeur à l'avenir.

Multiplexeur 2 vers 1 - MUX.vhd

Lien vers le code du multiplexeur 2 vers 1 (GitHub)

- *Entité : multiplexeur*

Un multiplexeur est un composant essentiel dans de nombreux systèmes numériques, y compris les processeurs. Il s'agit d'un dispositif qui peut acheminer l'un de ses multiples signaux d'entrée vers une seule sortie, en fonction de la valeur d'un ou plusieurs signaux de commande.

Le multiplexeur prend deux entrées, A et B, qui sont des vecteurs de logique standard de taille N. La taille N est un paramètre générique qui peut être spécifié lors de l'instanciation du multiplexeur. Par défaut, N est défini sur 32, ce qui signifie que le multiplexeur peut acheminer des vecteurs de 32 bits.

En plus des entrées A et B, le multiplexeur prend également une entrée de commande COM. Cette entrée détermine quelle entrée est acheminée vers la sortie. Si COM est '0', alors l'entrée A est acheminée vers la sortie. Si COM est '1', alors l'entrée B est acheminée vers la sortie.

La sortie du multiplexeur est S, qui est également un vecteur de logique standard de taille N.

La conception du multiplexeur est un exemple de la manière dont les composants numériques peuvent être conçus pour être flexibles et réutilisables.

En utilisant un paramètre générique pour spécifier la taille des vecteurs d'entrée et de sortie, le même code VHDL peut être utilisé pour créer des multiplexeurs qui manipulent des vecteurs de différentes tailles.

Cela permet de réutiliser le même code dans différentes parties du processeur, ce qui peut simplifier la conception et la maintenance du système, ce qui est très pratique dans un tel projet.

La conception du multiplexeur illustre également l'importance de la modularité dans la conception de circuits numériques. Le multiplexeur est un composant simple qui effectue une seule tâche, mais il peut être combiné avec d'autres composants pour construire des systèmes numériques plus complexes. En concevant des composants modulaires qui peuvent être réutilisés et combinés de différentes manières, nous pouvons simplifier la conception de systèmes numériques complexes et rendre ces systèmes plus faciles à comprendre et à maintenir.

Testbench du multiplexeur 2 vers 1

Lien vers le code du testbench du multiplexeur 2 vers 1 (GitHub)

Dans notre projet, le multiplexeur est un composant crucial qui doit fonctionner correctement pour que le processeur puisse exécuter des instructions. Par conséquent, il est essentiel de tester le multiplexeur pour s'assurer qu'il répond correctement aux entrées et qu'il produit les sorties attendues.

Le testbench commence par déclarer une instance de l'entité multiplexeur et la connecter à un ensemble de signaux qui seront utilisés pour tester son comportement. Ces signaux incluent deux entrées A et B, une entrée de commande COM et une sortie S.

Le testbench génère ensuite une série de stimuli pour le multiplexeur en utilisant un processus séparé (stimulus). Ce processus génère quatre cas de test différents, chacun testant le comportement du multiplexeur avec différentes valeurs d'entrée et de commande.

Dans chaque cas de test, le processus stimulus définit les valeurs des entrées A et B et du signal de commande COM, puis attend un certain temps pour que le multiplexeur réagisse. Il vérifie ensuite que la sortie S du multiplexeur est égale à l'entrée attendue (A si COM est '0', B sinon). Si la sortie n'est pas égale à l'entrée attendue, le testbench signale une erreur à l'aide d'une instruction **assert**.

Les instructions **assert** sont un outil puissant pour la vérification de la conception de circuits numériques. Elles permettent de spécifier des conditions qui doivent être vraies à certains points du testbench. Si une condition spécifiée par une instruction **assert** n'est pas vraie, le simulateur VHDL signalera une erreur et fournira un message spécifié par le concepteur. Cela permet de localiser rapidement et précisément les erreurs dans la conception.

Dans ce testbench, les instructions **assert** sont utilisées pour vérifier que la sortie du multiplexeur est correcte pour chaque cas de test. Si la sortie n'est pas correcte, l'instruction **assert** signalera une erreur et le message d'erreur indiquera quel cas de test a échoué.

Enfin, le testbench se termine par une instruction assert qui signale toujours une erreur. Cette instruction est utilisée pour indiquer la fin du testbench. Lorsque cette instruction est exécutée, le simulateur VHDL signalera une note indiquant "Fin du testbench". Cela permet de savoir que tous les cas de test ont été exécutés et que le testbench s'est terminé normalement.

En conclusion, ce testbench est un outil précieux pour vérifier le fonctionnement du multiplexeur. Il teste le multiplexeur avec une variété de stimuli et vérifie soigneusement ses réponses, ce qui permet de s'assurer que le multiplexeur fonctionne correctement dans toutes les conditions prévues. Cela est essentiel pour garantir la fiabilité et la précision du processeur dans son ensemble.

Extension de signe - sign_extension.vhd

Lien vers le code de l'extension de signe (GitHub)

- *Entité : sign_extension*

L'extension de signe est une opération courante en informatique qui est utilisée pour augmenter la taille d'un nombre tout en conservant sa valeur signée. C'est une étape essentielle dans de nombreux systèmes numériques, y compris les processeurs, car elle permet de manipuler des nombres de différentes tailles de manière cohérente.

L'entité sign_extension prend en entrée un vecteur de logique standard E de taille N et produit une sortie S de taille 32. La taille N est un paramètre générique qui peut être spécifié lors de l'instanciation de l'entité. Par défaut, N est défini sur 16, ce qui signifie que l'entité peut étendre des nombres de 16 bits à 32 bits.

L'architecture de l'entité sign_extension contient un processus qui est déclenché par des changements dans l'entrée E. Ce processus vérifie le bit de signe (le bit le plus significatif) de l'entrée E. Si le bit de signe est '1' (ce qui indique un nombre négatif), le processus étend le signe en concaténant 16 bits '1' à l'entrée E, convertit le résultat en une valeur signée et l'assigne à la sortie S. Si le bit de signe est '0' (ce qui indique un nombre positif), le processus étend le signe en concaténant 16 bits '0' à l'entrée E, convertit le résultat en une valeur signée et l'assigne à la sortie S.

L'extension de signe est un composant fondamental dans la conception de circuits numériques. Elle est utilisée pour adapter les nombres de différentes tailles aux opérations arithmétiques et logiques. Dans le contexte de notre processeur mono-cycle, l'extension de signe est utilisée pour adapter les nombres de 16 bits aux opérations sur 32 bits, ce qui est essentiel pour le fonctionnement correct du processeur.

Testbench de l'extension de signe

Lien vers le code du testbench de l'extension de signe (GitHub)

Le testbench pour l'extension de signe est conçu pour vérifier que l'entité ***sign_extension*** fonctionne correctement. Il génère une série de stimuli pour l'extension de signe et vérifie que la sortie correspond à ce qui est attendu pour chaque entrée.

Dans ce testbench, le processus stimulus génère cinq cas de test différents. Chaque cas de test définit une valeur pour l'entrée E, attend un certain temps pour que l'extension de signe réagisse, puis vérifie que la sortie S est égale à la valeur attendue. Si la sortie n'est pas égale à la valeur attendue, le testbench signale une erreur à l'aide d'une instruction **assert**.

Les instructions **assert** sont un outil puissant pour la vérification de la conception de circuits numériques. Elles permettent de spécifier des conditions qui doivent être vraies à certains points du testbench. Si une condition spécifiée par une instruction **assert** n'est pas vraie, le

simulateur VHDL signalera une erreur et fournira un message spécifié par le concepteur dans la console. Cela permet de localiser rapidement et précisément les erreurs dans la conception.

Dans ce testbench, les instructions **assert** sont utilisées pour vérifier que la sortie de l'extension de signe est correcte pour chaque cas de test.

Si la sortie n'est pas correcte, l'instruction **assert** signalera une erreur et le message d'erreur indiquera quel cas de test a échoué.

Enfin, le testbench se termine par une instruction **assert** qui signale toujours une erreur. Cette instruction est utilisée pour indiquer la fin du testbench. Lorsque cette instruction est exécutée, le simulateur VHDL signalera une note indiquant "Fin du testbench".

En conclusion, ce testbench est un outil précieux pour vérifier le fonctionnement de l'extension de signe. Il teste l'extension de signe avec une variété de stimuli et vérifie soigneusement ses réponses, ce qui permet de s'assurer que l'extension de signe fonctionne correctement dans toutes les conditions prévues. Cela est essentiel pour garantir la fiabilité et la précision du processeur dans son ensemble.

Mémoire de données – data_memory.vhd

Lien vers le code de la mémoire de données (GitHub)

- *Entité : data_memory*

La mémoire de données est un composant essentiel de notre processeur. Elle est responsable du stockage et de la récupération des données pendant l'exécution des instructions. La mémoire de données est conçue pour stocker 64 mots de 32 bits, ce qui est suffisant pour la plupart des applications de notre processeur.

L'entité data_memory a cinq ports : CLK, DataIn, DataOut, Addr et WrEn. CLK est l'entrée d'horloge qui synchronise l'écriture de données dans la mémoire. DataIn est l'entrée de données à écrire dans la mémoire. DataOut est la sortie de données lues à partir de la mémoire. Addr est l'adresse à laquelle les données doivent être écrites ou lues. Enfin, WrEn est le signal d'activation d'écriture qui, lorsqu'il est à '1', permet l'écriture de données dans la mémoire.

L'architecture de data_memory définit un type memory_array qui est un tableau de 64 vecteurs de logique standard de 32 bits. Un signal memory de ce type est déclaré pour représenter la mémoire de données.

L'architecture définit ensuite un processus qui est déclenché par le front montant du signal d'horloge CLK. Si le signal WrEn est à '1', ce processus écrit la valeur de DataIn à l'adresse spécifiée par Addr dans la mémoire.

Enfin, la sortie DataOut est assignée à la valeur de la mémoire à l'adresse spécifiée par Addr. Cette assignation est combinatoire, ce qui signifie qu'elle se produit immédiatement chaque fois que la valeur de Addr change.

En somme, la mémoire de données est un composant clé de notre processeur qui permet le stockage et la récupération de données. Sa conception en VHDL est simple et efficace, ce qui facilite son intégration dans le reste du processeur.

Testbench de la mémoire de données

Lien vers le code du testbench de la mémoire de données (GitHub)

Le testbench pour la mémoire de données est conçu pour vérifier que l'entité **data_memory** fonctionne correctement. Il génère une série de stimuli pour la mémoire de données et vérifie que la sortie correspond à ce qui est attendu pour chaque entrée.

Dans ce testbench, le processus stimulus génère trois cas de test différents. Chaque cas de test définit une valeur pour DataIn, Addr et WrEn, attend un certain temps pour que la mémoire de données réagisse, puis vérifie que la sortie DataOut est égale à la valeur attendue. Si la sortie n'est pas égale à la valeur attendue, le testbench signale une erreur à l'aide d'une instruction assert.

Les instructions assert sont un outil puissant pour la vérification de la conception de circuits numériques. Elles permettent de spécifier des conditions qui doivent être vraies à certains points du testbench. Si une condition spécifiée par une instruction assert n'est pas vraie, le simulateur VHDL signalera une erreur et fournira un message spécifié par moi. Cela me permet de localiser rapidement et précisément les erreurs dans la conception.

Dans ce testbench, les instructions assert sont utilisées pour vérifier que la sortie de la mémoire de données est correcte pour chaque cas de test. Si la sortie n'est pas correcte, l'instruction assert signalera une erreur et le message d'erreur indiquera quel cas de test a échoué.

Enfin, le testbench se termine par une instruction assert qui signale toujours une erreur. Cette instruction est utilisée pour indiquer la fin du testbench. Lorsque cette instruction est exécutée, le simulateur VHDL signalera une note indiquant "Fin du testbench". Cela me permet de savoir que tous les cas de test ont été exécutés et que le testbench s'est terminé normalement.

En conclusion, ce testbench est un outil précieux pour vérifier le fonctionnement de la mémoire de données. Il teste la mémoire de données avec une variété de stimuli et vérifie soigneusement ses réponses, ce qui me permet de m'assurer que la mémoire de données fonctionne correctement dans toutes les conditions prévues. C'est essentiel pour garantir la fiabilité et la précision de notre processeur dans son ensemble.

Unité de traitement - Unite_Traitement.vhd

Lien vers le code de l'unité de traitement (GitHub)

- *Entité : traitement_unit*

L'unité de traitement est le cœur de notre processeur ! Elle est responsable de l'exécution des opérations arithmétiques et logiques, de la gestion de la mémoire de données, de l'extension de signe et de la sélection des données appropriées pour l'écriture et la lecture à partir de la mémoire.

L'entité **traitement_unit** comporte plusieurs ports d'entrée et de sortie. Les entrées comprennent l'horloge **CLK**, les signaux de contrôle **We**, **WrEn**, **COM1**, **COM2**, **RESET**, le vecteur d'instruction immédiate **imm**, les adresses de registre **RW**, **RA**, **RB** et l'opération à effectuer **OP**. Les sorties sont le drapeau **flag** et le bus de sortie **BusW**.

L'architecture de **traitement_unit** définit plusieurs signaux internes pour connecter les différentes entités qui composent l'unité de traitement. Ces signaux sont **Bus_W**, **Bus_A**, **Bus_B**, **EXS_OUT**, **Mux_F_OUT**, **ALU_OUT** et **Data_OUT**.

L'architecture connecte ensuite ces signaux aux entités appropriées. L'entité **banc_registres** est connectée à **Bus_W**, **Bus_A** et **Bus_B**. L'entité **ual** (Unité Arithmétique et Logique) est connectée à **Bus_A**, **Mux_F_OUT**, **ALU_OUT** et **flag**. L'entité **data_memory** (Mémoire de Données) est connectée à **CLK**, **Bus_B**, **Data_OUT**, **ALU_OUT** et **WrEn**. L'entité **sign_extension** (Extension de Signe) est connectée à **imm** et **EXS_OUT**. Enfin, deux instances de l'entité **multiplexeur** sont utilisées pour sélectionner les données appropriées pour l'écriture et la lecture à partir de la mémoire.

L'unité de traitement est donc une entité complexe qui intègre plusieurs autres entités pour réaliser les différentes opérations nécessaires au fonctionnement du processeur. Elle est conçue pour être flexible et modulaire, de sorte que les différentes entités peuvent être modifiées ou remplacées si nécessaire, sans affecter le fonctionnement global de l'unité de traitement. C'est un exemple de la manière dont la conception modulaire peut être utilisée pour créer des systèmes numériques complexes et flexibles, ce qui nous a permis de découvrir l'assemblage de plusieurs blocs complexes en un seul afin de l'utiliser ensuite dans notre processeur.

Testbench de l'unité de traitement

Lien vers le code du testbench de l'unité de traitement (GitHub)

Le testbench de l'unité de traitement est conçu pour vérifier le bon fonctionnement de cette dernière. Il simule une série de scénarios d'entrée et vérifie que les sorties sont correctes.

Dans ce testbench, j'ai défini plusieurs cas de test pour vérifier le bon fonctionnement de l'unité de traitement. Chaque cas de test écrit une valeur dans une adresse spécifique de la mémoire, puis lit cette valeur et vérifie qu'elle est correcte. Si la valeur lue ne correspond pas à la valeur écrite, une erreur est signalée.

Le testbench utilise également un processus pour générer un signal d'horloge, qui est nécessaire pour le fonctionnement de l'unité de traitement.

Le signal d'horloge est initialisé à '0', puis bascule entre '0' et '1' toutes les 10 ns pour simuler le passage du temps.

- Le premier cas de test écrit la valeur **x"12345678"** à l'adresse **5** de la mémoire, puis lit cette valeur et vérifie qu'elle est correcte.
- Le deuxième cas de test fait de même avec la valeur **x"9ABCDEF0"** à l'adresse **10**.
- Le troisième cas de test lit la valeur à l'adresse **63**, qui est censée être non initialisée et donc égale à **0**.

Si l'un de ces tests échoue, une erreur est signalée et le message d'erreur indique le cas de test qui a échoué. Cela permet de localiser rapidement le problème et de le corriger.

Enfin, une note d'information est émise à la fin du testbench pour indiquer que tous les tests ont été exécutés. C'est une bonne pratique pour s'assurer que le testbench s'est exécuté jusqu'au bout et n'a pas été interrompu en cours de route.

En conclusion, ce testbench est un outil essentiel pour vérifier le bon fonctionnement de l'unité de traitement. Il permet de s'assurer que l'unité de traitement fonctionne comme prévu et de détecter rapidement tout problème qui pourrait survenir.

Unité de gestion des instructions - instructions_management_unit.vhd

Lien vers le code de l'unité de gestion des instructions (GitHub)

- *Entité : instruction_unit*

L'Unité de Gestion des Instructions (UGI) est une composante essentielle du processeur. Elle est responsable de la gestion des instructions 32 bits. L'UGI possède une mémoire d'instruction de 64 mots de 32 bits, similaire à celle de l'unité de traitement. Elle ne dispose pas de bus de données en écriture ni de Write Enable.

L'UGI comprend également un registre 32 bits (Registre PC) qui possède une horloge et un reset asynchrone (actif à l'état haut) non représentés sur le schéma. Elle contient une unité d'extension de 24 à 32 bits signés similaire au module décrit précédemment. Enfin, elle dispose d'une unité de mise à jour du compteur de programme PC suivant le signal de contrôle nPCsel. Si nPCsel = 0 alors PC = PC + 1, et si nPCsel = 1 alors PC = PC + 1 + SignExt(offset).

Le code VHDL fourni décrit l'architecture de l'UGI. Elle comprend un processus qui gère le registre PC en fonction du signal de reset et de l'horloge. Un autre processus gère la mise à jour du PC en fonction du signal nPCsel, du PC actuel et de l'offset signé étendu. Enfin, l'instruction de sortie est extraite de la mémoire d'instruction en fonction du PC.

Cette entité est une composante clé du processeur, car elle gère le flux d'instructions et contrôle le déroulement du programme. Elle est essentielle pour le bon fonctionnement du processeur.

Nous n’avons pas réalisé de testbench pour cette entité, par soucis de gain de temps. Nous nous sommes donc assurés lors de la réalisation du code que cette entité fonctionnerait correctement, et cela a été validé lors du test final du processeur.

Registre 32 bit avec commande de chargement - control_unit.vhd

Lien vers le code de l’unité de contrôle (GitHub)

- *Entité : control_unit*

L'Unité de Contrôle (UC) est une composante cruciale du processeur. Elle est responsable de la gestion des signaux de contrôle qui orchestrent le fonctionnement des autres composants du processeur. L'UC reçoit des instructions du bus de données d'entrée (DATAIN) et génère des signaux de contrôle qui sont transmis aux autres composants du processeur.

Dans notre code, l’unité de contrôle est définie comme une entité avec cinq ports : DATAIN, RST, CLK, WE et DATAOUT.
DATAIN est une entrée de 32 bits qui reçoit les instructions du bus de données. RST est une entrée de réinitialisation qui, lorsqu'elle est active (à '1'), réinitialise l'état de l'UC. CLK est l'entrée d'horloge qui synchronise les opérations de l'UC. WE est un signal d'activation d'écriture qui, lorsqu'il est actif (à '1'), permet à l'UC de mettre à jour son état interne. Enfin, DATAOUT est une sortie de 32 bits qui transmet l'état actuel de l'UC au reste du processeur.

L'architecture de l'UC est définie dans le bloc "behavior". Elle comprend un registre d'état du processeur (PSR) de 32 bits qui stocke l'état actuel de l'UC. Un processus est défini pour gérer le PSR. Si le signal de réinitialisation est actif, le PSR est réinitialisé à zéro. Sinon, à chaque front montant de l'horloge, si le signal d'activation d'écriture est actif, le PSR est mis à jour avec la valeur actuelle de DATAIN. Enfin, la valeur actuelle du PSR est transmise à DATAOUT.

L'UC joue un rôle central dans le fonctionnement du processeur. Elle détermine l'opération à effectuer en fonction de l'instruction actuelle et génère les signaux de contrôle appropriés pour les autres composants du processeur. Sans l'UC, le processeur ne serait pas en mesure d'exécuter correctement les instructions.

Encore une fois, par soucis de gain de temps, nous n’avons pas réalisé de testbench pour cette entité, mais nous avons validé son fonctionnement lors du test final du processeur.

Décodeur d'Instructions - instruction_decoder.vhd

Lien vers le code du décodeur d'instructions (GitHub)

- *Entité : Instruction_Decoder*

L'entité **Instruction_Decoder** est une partie essentielle du processeur. Elle est responsable de la décomposition des instructions entrantes en signaux de contrôle qui dirigent le fonctionnement des autres parties du processeur.

L'entité **Instruction_Decoder** a plusieurs entrées et sorties.

Les entrées comprennent l'instruction à décoder (**Instruction**) et le registre d'état du processeur (**PSR -> Processor State Register**).

Les sorties sont les signaux de contrôle générés par le décodeur.

Le décodeur d'instructions est conçu pour gérer un ensemble spécifique d'instructions, y compris **MOV**, **ADDi**, **ADDr**, **CMP**, **LDR**, **STR**, **BAL**, et **BLT**. Chaque instruction est associée à un ensemble spécifique de signaux de contrôle qui sont générés lorsque l'instruction est décodée, à partir des valeurs données en annexe du sujet.

Le décodeur d'instructions utilise une architecture comportementale, ce qui signifie qu'il est conçu pour décrire le comportement du décodeur plutôt que sa structure physique. Il utilise deux processus pour accomplir cela. Le premier processus détermine l'instruction courante en fonction de l'instruction entrante. Le deuxième processus génère les signaux de contrôle appropriés en fonction de l'instruction courante.

Le décodeur d'instructions est conçu pour fonctionner avec des instructions de 32 bits. Cependant, il est également capable de gérer des instructions plus courtes grâce à l'utilisation de l'extension de signe.

C'est donc, encore une fois, une partie essentielle du processeur qui décode les instructions entrantes et génère les signaux de contrôle appropriés pour diriger le fonctionnement du processeur.

Assemblage du processeur - processor.vhd

Lien vers le code du processeur assemblé (GitHub)

- *Entité : processor*

L'entité **processor** est le cœur de notre implémentation VHDL du processeur monocycle. Elle est le composant principal qui intègre tous les autres composants du processeur, y compris le décodeur d'instructions, l'unité arithmétique et logique (UAL) et l'unité de contrôle.

Interface

L'interface de l'entité **processor** est composée de plusieurs signaux d'entrée et de sortie :

- **clock** : C'est l'entrée d'horloge qui synchronise l'ensemble du processeur. Toutes les opérations du processeur sont synchronisées sur les fronts montants de ce signal d'horloge.
- **reset** : C'est le signal de réinitialisation. Lorsqu'il est activé (niveau haut), il remet le processeur à son état initial.
- **Instruction** : C'est une entrée de 32 bits qui fournit les instructions à exécuter par le processeur. Chaque instruction est codée sur 32 bits selon l'architecture MIPS.
- **PSR** : Le registre d'état du processeur (Processor State Register) est un signal bidirectionnel de 32 bits. Il contient des informations sur l'état actuel du processeur, comme les indicateurs de condition.
- **DATAIN** : C'est une entrée de données de 32 bits qui fournit les données à traiter par le processeur.
- **DATAOUT** : C'est une sortie de données de 32 bits qui renvoie les résultats des opérations du processeur.
- **WE** : C'est le signal d'activation d'écriture (Write Enable). Lorsqu'il est activé (niveau haut), il autorise l'écriture dans les registres du processeur.

Architecture

L'architecture de l'entité **processor** décrit comment ces composants sont interconnectés. Elle définit également plusieurs signaux internes qui sont utilisés pour le contrôle et la communication entre les différents composants.

- **Décodeur d'instructions** : Ce composant, instancié en tant qu'**inst_dec**, décode l'instruction entrante et génère des signaux de contrôle pour les registres et les opérateurs. Il est connecté directement à l'entrée **Instruction** et au registre d'état du processeur **PSR**.
- **Unité Arithmétique et Logique (UAL)** : Cette unité, instanciée en tant qu'**ual**, effectue diverses opérations arithmétiques et logiques en fonction de l'instruction décodée. Elle est connectée aux signaux **a** et **b** pour ses entrées et produit un signal **s** pour sa sortie.

- **Unité de contrôle** : Cette unité, instanciée en tant que **ctrl_unit**, gère l'état du processeur et contrôle le flux de données entre les différents composants du processeur. Elle est connectée à l'entrée **DATAIN**, à l'horloge **clock**, au signal de réinitialisation **reset**, au signal d'activation d'écriture **WE** et à la sortie **DATAOUT**.

Cette architecture permet une grande modularité et une séparation claire des responsabilités entre les différents composants du processeur.

Voici un schéma (donné en annexe du sujet) de l'architecture complète du processeur :

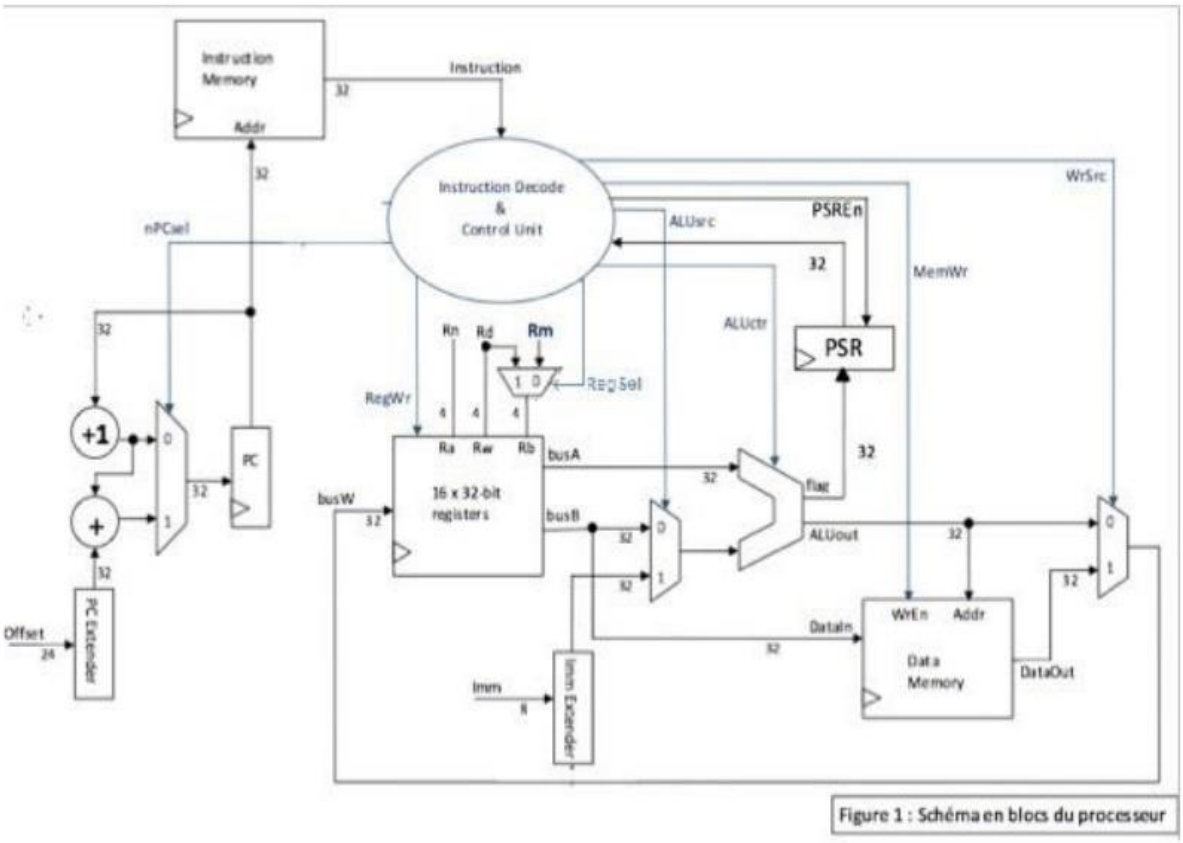

Figure 2 : Schéma en blocs de l'architecture du processeur

Ce schéma illustre parfaitement le processeur que nous avons décrit précédemment en VHDL.

Testbench du processeur

Lien vers le code du testbench du processeur (GitHub)

Le testbench du processeur, nommé **processor_tb**, a été conçu pour tester la fonctionnalité de l'entité **processor**. Il simule le fonctionnement du processeur avec une série d'instructions spécifiées dans l'entité **instruction_unit**.

Chaque instruction est testée individuellement avec une assertion pour vérifier que le résultat est correct. Si une assertion échoue, un message d'erreur est affiché. Le test se termine lorsque toutes les instructions ont été testées.

Le testbench comprend deux processus principaux : un processus d'horloge et un processus de test.

Processus d'horloge

Le processus d'horloge génère un signal d'horloge pour simuler le fonctionnement du processeur en temps réel. Le signal d'horloge est un signal carré avec une période de 20 ns, ce qui signifie que l'état du signal change toutes les 10 ns.

Processus de test

Le processus de test simule les instructions qui sont envoyées au processeur. Il initialise les signaux d'entrée du processeur, puis exécute une série d'instructions en écrivant les codes d'instruction appropriés sur le signal d'instruction. Après chaque instruction, le processus de test vérifie que le processeur a produit le résultat attendu en utilisant une assertion.

Par exemple, pour tester l'instruction **ADD**, le processus de test écrit le code d'instruction pour **ADD** sur le signal d'instruction, puis vérifie que le processeur a correctement ajouté les valeurs des registres d'entrée.

Assertions

Les assertions sont utilisées pour vérifier que le processeur produit les résultats attendus. Chaque assertion compare la sortie du processeur à une valeur attendue. Si la sortie du processeur ne correspond pas à la valeur attendue, l'assertion échoue et un message d'erreur est affiché.

Ces assertions permettent de vérifier automatiquement que le processeur fonctionne correctement, ce qui facilite grandement le processus de test.

Voici les différentes opérations (chargées dans l'entité instructions_unit) testées dans ce testbench :

```
0x0   _main :    MOV R1,#0x20     ; --R1 <= 0x20
0x1              MOV R2,#0        ; --R2 <= 0
0x2   _loop :    LDR R0,0(R1)     ; --R0 <= DATA_MEM[R1]
0x3              ADD R2,R2,R0     ; --R2 <= R2 + R0
0x4              ADD R1,R1,#1     ; --R1 <= R1 + 1
0x5              CMP R1,0x2A      ; --? R1 = 0x2A
0x6              BLT loop         ; --branchement à _loop si
R1 inferieur a 0x2A
      _end  :
0x7              STR R2,0(R1)     ; --DATA_MEM[R1] <= R2
0x8              BAL main         ; --branchement à _main
```

Figure 3 : Instructions chargées pour le programme de test

Voici un exemple d'assertions, qui sont effectuées toutes les 20ns (à chaque front montant d'horloge), pour tester différentes opérations :

```
60          -- Test process
61          test_process : process
62          begin
63              -- Reset the processor
64              reset <= '1';
65              wait for 20 ns;
66              reset <= '0';
67
68              -- Test the MOV instruction
69              -- R1 <= 0x20;
70              wait for 20 ns;
71              assert PSR = "00000000000000000000000000100000"
72              report "MOV instruction failed"
73              severity error;
74
75              -- Test the ADDi instruction
76              -- R2 <= 0x0;
77              wait for 20 ns;
78              assert PSR = "00000000000000000000000000000000"
79              report "ADDi instruction failed"
80              severity error;
81
82              -- Test the LDR instruction
83              -- R0 <= DATA_MEM[R1];
84              wait for 20 ns;
85              assert PSR = "00000000000000000000000000000000"
86              report "LDR instruction failed"
87              severity error;
88
89              -- Test the ADDr instruction
90              -- R2 <= R2 + R0;
91              wait for 20 ns;
92              assert PSR = "00000000000000000000000000000000"
93              report "ADDR instruction failed"
94              severity error;
```

Figure 4 : Exemple d'assertions dans le testbench du processeur

En conclusion, le testbench **processor_tb** est un outil essentiel pour vérifier la fonctionnalité du processeur. Il permet de tester automatiquement chaque instruction et de vérifier que le processeur produit les résultats attendus.

Cela nous a donc permis de valider le fonctionnement de notre système complet.

Conclusion - projet de réalisation d’un processeur mono-cycle en VHDL

Au terme de ce projet, nous avons acquis une compréhension approfondie du fonctionnement interne d'un processeur monocycle, ainsi qu'une solide compétence en VHDL, un langage de description de matériel largement utilisé dans l'industrie.

La conception et l'implémentation de chaque composant du processeur nous ont permis de comprendre en détail comment les instructions sont décodées et exécutées dans un processeur. Nous avons appris à concevoir des composants modulaires et à les intégrer dans une architecture plus large, ce qui nous a permis de comprendre comment les différentes parties d'un système peuvent interagir pour réaliser des opérations complexes.

L'unité arithmétique et logique (UAL), par exemple, nous a permis de comprendre comment les opérations arithmétiques et logiques sont réalisées au niveau du matériel. La conception de l'UAL nous a également permis de nous familiariser avec les différentes structures de VHDL, telles que les entités, les architectures et les processus.

Le décodeur d'instructions, quant à lui, nous a permis de comprendre comment les instructions sont décodées et comment les signaux de contrôle sont générés pour les autres composants du processeur. La conception du décodeur d'instructions nous a permis de nous familiariser avec les concepts de décodage d'instructions et de génération de signaux de contrôle.

L'unité de contrôle, enfin, nous a permis de comprendre comment l'état du processeur est géré et comment le flux de données est contrôlé. La conception de l'unité de contrôle nous a permis de nous familiariser avec les concepts de gestion de l'état du processeur et de contrôle du flux de données.

En outre, nous avons appris à travailler efficacement en binôme, en partageant les tâches et en collaborant étroitement tout au long du projet. Cette expérience nous a permis de développer nos compétences en matière de travail d'équipe et de communication, des compétences essentielles pour tout ingénieur.

En somme, ce projet a été une expérience d'apprentissage précieuse et enrichissante. Nous sommes convaincus que les compétences et les connaissances que nous avons acquises grâce à ce projet seront extrêmement utiles pour notre future carrière en tant qu'ingénieurs.