Rapport Architecture Systèmes  
-  
Processeur mono-cycle

Grégoire MAHON

Armand LELONG

EI2I-3 – II (Groupe B)

# 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](https://github.com/gregoiremahon/mono-cycle-processor-VHDL/blob/main/src/UAL.vhd) (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](https://github.com/gregoiremahon/mono-cycle-processor-VHDL/blob/main/src/banc_registres.vhd) (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](https://github.com/gregoiremahon/mono-cycle-processor-VHDL/blob/main/testbenches/testbench_banc_registres.vhd) (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](https://github.com/gregoiremahon/mono-cycle-processor-VHDL/blob/main/src/MUX.vhd) (GitHub)

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](https://github.com/gregoiremahon/mono-cycle-processor-VHDL/blob/main/testbenches/testbench_MUX.vhd) (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](https://github.com/gregoiremahon/mono-cycle-processor-VHDL/blob/main/src/sign_extension.vhd) (GitHub)

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](https://github.com/gregoiremahon/mono-cycle-processor-VHDL/blob/main/testbenches/testbench_sign_extension.vhd) (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](https://github.com/gregoiremahon/mono-cycle-processor-VHDL/blob/main/src/data_memory.vhd) (GitHub)

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](https://github.com/gregoiremahon/mono-cycle-processor-VHDL/blob/main/testbenches/testbench_data_memory.vhd) (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](https://github.com/gregoiremahon/mono-cycle-processor-VHDL/blob/main/src/Unite_Traitement.vhd) (GitHub)

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](https://github.com/gregoiremahon/mono-cycle-processor-VHDL/blob/main/testbenches/testbench_data_memory.vhd) (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](https://github.com/gregoiremahon/mono-cycle-processor-VHDL/blob/main/src/instructions_management_unit.vhd) (GitHub)

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.