École nationale supérieure de techniques avancées Bretagne

—

Wrapper

Étudiant :
Bilal LATRACH

Encadré par :
Loïc Lagadec
Théotime Bollengier

1er septembre 2024

Table des matières

1 Introduction

Le wrapper a pour fonction de contrôler les requêtes échangées entre les maîtres et les esclaves dans un système numérique. Il intègre une mémoire interne où sont stockées des règles, chaque règle étant définie par un numéro (rule_number) et un contenu spécifique (data_rule). L'écriture d'une règle dans cette mémoire est déclenchée en activant le signal w_rule_enable (mis à 1).

Outre la gestion des règles, le wrapper utilise des signaux pour identifier les requêtes des maîtres et vérifier leur conformité par rapport aux règles stockées en mémoire. Si une requête respecte au moins une règle, le wrapper envoie un signal de sortie à 1, indiquant que la requête est autorisée. Deux signaux de sortie sont utilisés : l'un pour la réponse en lecture et l'autre pour l'écriture. Les identifiants des maîtres sont fournis par les signaux MID_W (écriture) et MID_R (lecture), permettant de traiter simultanément une requête de lecture et une d'écriture. Un autre signal, x_enable, sert à indiquer si le maître demande l'exécution de l'opération.

Les adresses d'écriture et de lecture (add_w et add_r) sont obtenues à partir des signaux d'adresse de l'interface AXI (S_AXI_AWADDR pour l'écriture et S_AXI_ARADDR pour la lecture). Le wrapper et la mémoire sont synchronisés avec le système via les signaux S_AXI_ACLK (horloge) et S_AXI_ARESETN (réinitialisation) de l'interface AXI, garantissant une coordination correcte des opérations.

2 Wrapper

2.1 Schèma

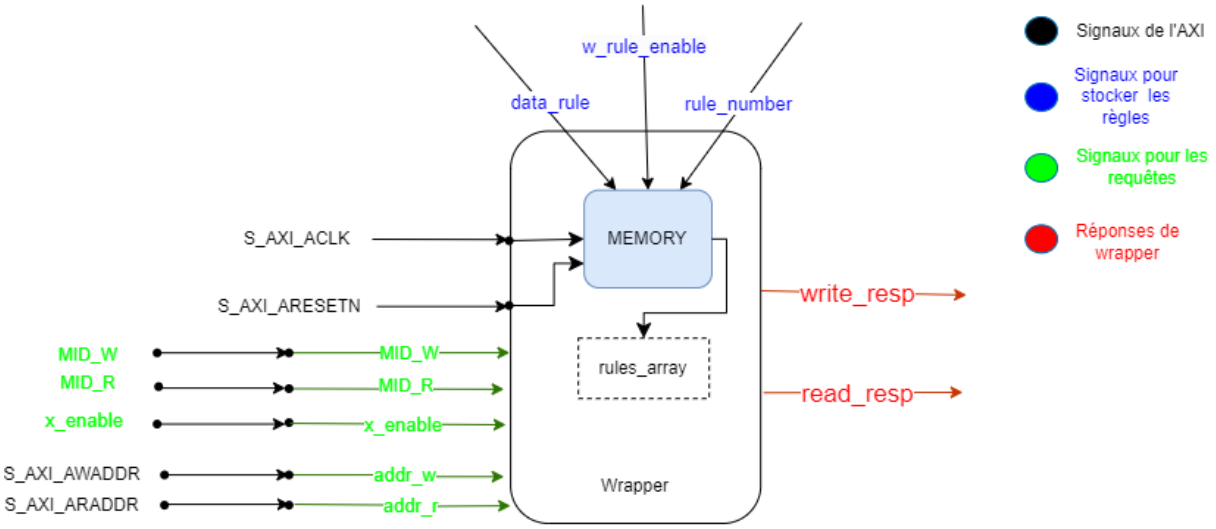

FIGURE 1 – Fonctionnement de wrapper

Accéder au code source

2.2 Package Memory_type

Le package Memory_type définit un type de tableau en VHDL, MemoryArrayType, qui est un tableau de vecteurs de logique (std_logic_vector). Ce type est utilisé pour stocker et manipuler des données dans des modules tels que rules_array et wrapper. Il définit une mémoire à n emplacements, chacun pouvant contenir un vecteur de m bits, permettant de stocker des données de manière structurée.

2.3 Composant rules_array

L'entité rules_array est un module de mémoire qui stocke les règles de lecture et d'écriture dans un tableau de vecteurs logique. Cette entité utilise une horloge (clk) et un signal de réinitialisation (reset) pour gérer les opérations de lecture et d'écriture. Lorsqu'un signal de validation d'écriture (w_enable) est actif, les données entrantes sont stockées dans le tableau à l'emplacement spécifié par rule_number. Le tableau est réinitialisé à zéro lorsque le signal de réinitialisation est actif. Les données stockées dans la mémoire sont disponibles en sortie sous la forme d'un tableau.

2.4 Composant wrapper

L'entité wrapper est responsable de l'implémentation de la logique qui vérifie les permissions de lecture et d'écriture en fonction des règles stockées dans le module rules_array.

Elle prend en entrée des signaux tels que les données de règle (data_rule), le numéro de règle (rule_number), ainsi que les adresses de lecture et d'écriture. Le module compare ces informations avec les règles stockées pour déterminer si les transactions sont autorisées. Les résultats des vérifications sont retournés sous forme de signaux de réponse wrapper_write_response et wrapper_read_response.
Le wrapper contient une instance du module rules_array qui stocke les règles de transaction. Un processus est utilisé pour vérifier, à chaque front montant de l'horloge (clk), si les transactions de lecture ou d'écriture sont autorisées en fonction des règles stockées. Les résultats de cette vérification sont affectés aux signaux de réponse appropriés. Si le signal w_rule_enable est actif, les réponses sont mises à zéro, sinon, elles sont calculées en fonction des règles et des adresses.

3 Script Python pour la génération du wrapper

3.1 Objectif du Script

Ce script Python a été développé pour générer automatiquement des fichiers VHDL nécessaires à l'implémentation d'une mémoire, d'une wrapper et d'une interface AXI. La génération automatique permet de créer ces fichiers avec des tailles de mémoire personnalisées (profondeur et largeur) en fonction des besoins spécifiques du projet. Cela évite de devoir écrire manuellement les fichiers VHDL, ce qui pourrait être source d'erreurs, surtout lors de changements fréquents des paramètres de la mémoire.

3.2 Paramètres du Script

— **MEM_DEPTH** : La profondeur de la mémoire, c'est-à-dire le nombre de cellules de mémoire disponibles.
— **MEM_WIDTH** : La largeur de la mémoire, correspondant à la largeur de chaque cellule mémoire en bits.

Ces paramètres peuvent être ajustés directement dans le code ou via des arguments en ligne de commande lors de l'exécution du script. *(voir la partie exemple d'utilisation)*

3.3 Explication du Fonctionnement

Calculs Préliminaires

— **ID_width** : Nombre de bits nécessaires pour représenter les adresses dans la mémoire, calculé en fonction de `MEM_DEPTH`.
— **rwx_width** : Largeur du champ RWX (Read/Write/Execute), fixé à 3 bits.
— **adress_width** : Largeur de l'adresse à utiliser pour chaque mémoire, calculée en fonction de `MEM_WIDTH`, `ID_width`, et `rwx_width`.

Génération des Fichiers VHDL

Le script génère trois fichiers VHDL :
— **rules_array.vhd** : Contient la déclaration et la définition de la mémoire dans laquelle les règles sont stockées.

— **wrapper.vhd** : Définit un wrapper qui interagit avec l'instance de `rules_array` et gère les accès en lecture et en écriture, selon les droits et adresses spécifiées.
— **interface_AXI.vhd** : Implémente une interface AXI qui utilise le wrapper pour lire et écrire des données en fonction des transactions AXI reçues.

Chaque fichier est généré avec un contenu spécifique en fonction des paramètres de la mémoire (profondeur et largeur) passés au script.

Utilisation des Arguments de Ligne de Commande

Le script utilise le module `argparse` pour permettre à l'utilisateur de spécifier les valeurs de `MEM_DEPTH` et `MEM_WIDTH` via la ligne de commande. Cela rend le script flexible et facile à utiliser dans différents contextes de développement. Par défaut, `MEM_DEPTH` est fixé à 8 et `MEM_WIDTH` à 16, mais ces valeurs peuvent être modifiées avec les options `-d` et `-w`.

3.4 Exemple d'Utilisation

Pour générer les fichiers VHDL avec une profondeur de mémoire de 16 et une largeur de 32, le script peut être exécuté comme suit :

```
python3 wrapper.py -d 16 -w 32
```

Cela produira trois fichiers VHDL (`rules_array.vhd`, `wrapper.vhd`, et `interface_AXI.vhd`) adaptés à une mémoire de 16 cellules de 32 bits.

3.5 Avantages de ce Script

— **Automatisation** : La génération automatique des fichiers VHDL réduit le risque d'erreurs humaines lors de la configuration de la mémoire.
— **Flexibilité** : Les utilisateurs peuvent ajuster les paramètres de la mémoire à la volée, sans avoir à modifier manuellement le code VHDL.
— **Réutilisabilité** : Ce script peut être utilisé pour divers projets nécessitant des configurations de mémoire différentes, simplifiant ainsi le processus de développement.

Ce script est un outil puissant pour les développeurs travaillant avec VHDL et des architectures AXI, car il simplifie grandement la configuration et la génération de code VHDL personnalisé pour différentes applications. Grâce à cette approche, le développement de systèmes numériques devient plus modulable, rapide, et moins sujet aux erreurs.

4 Script Python pour la génération du test bench

4.1 Fichier de configuration de la mémoire

Ce fichier "memory_configuration.txt" contient la configuration de la mémoire, Chaque ligne du fichier représente une règle de sécurité, exprimé en mode hexadécimal. Les règles sont stockées en mémoire et associées à des identifiants de maîtres spécifiques. Le fichier inclut également des commentaires, marqués par le symbole #, qui offrent des explications ou des annotations pour faciliter la compréhension des règles inscrites.

4.2 Fichier de requêtes

Ce fichier de requêtes "request.txt" décrit les transactions effectuées dans un système basé sur une interface AXI. Chaque ligne correspond à une requête spécifique, avec plusieurs champs indiquant les détails de la transaction.

MID : L'identifiant maître, qui désigne la source de la requête.
RWX : Un champ de trois bits qui décrit les droits d'accès pour la transaction.
addr : L'adresse mémoire hexadécimale où l'opération doit être effectuée.
response_expected : Ce champ indique la réponse attendu du test.

4.3 Script python pour générer le test bench

Ce script Python génère un banc d'essai (test bench) en VHDL. Il commence par définir des constantes et des fonctions auxiliaires, comme celles permettant de convertir des nombres décimaux et hexadécimaux en binaire. Le script utilise ces fonctions pour générer des signaux de contrôle et de données basés sur des fichiers de configuration externes (memory_configuration.txt et request.txt). Le script crée ensuite un fichier de banc d'essai VHDL (Interface_AXI_tb.vhd) qui simule des opérations de lecture et d'écriture sur une mémoire, en vérifiant la validité des réponses en comparant les résultats attendus avec ceux obtenus, et en consignant les erreurs dans un fichier test_bench.log. Ce processus aide à valider le bon fonctionnement de l'interface AXI en simulant divers scénarios et en vérifiant leur conformité avec les règles définies.

5 Structure des fichiers

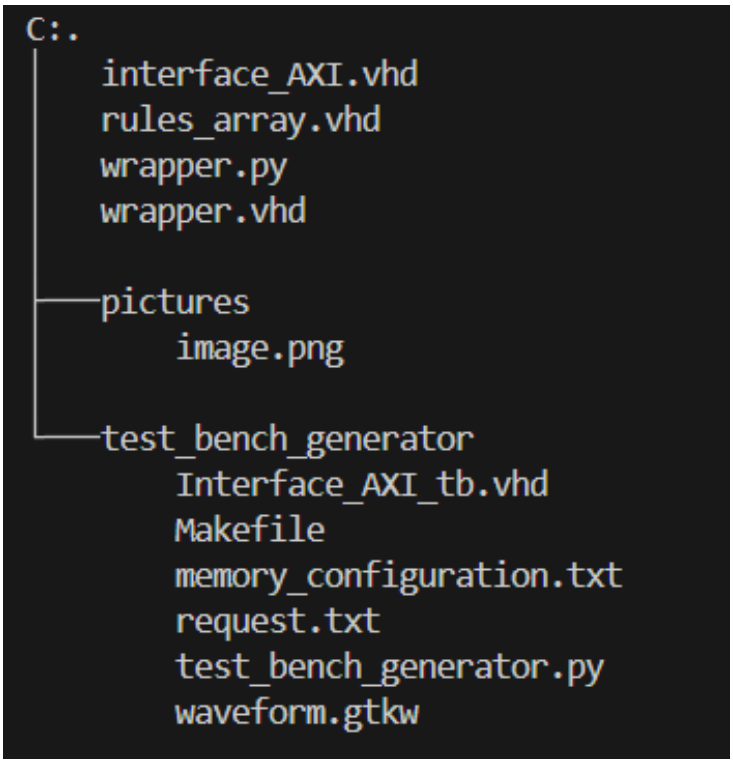

FIGURE 2 – Structure des fichiers

6 Exécutions

6.1 Exécution des scripts

Vous pouvez exécuter les deux scripts simultanément en utilisant le Makefile dans le répertoire wrapper_generator. Pour ce faire, il suffit de lancer la commande make, qui exécutera les deux scripts avec les paramètres par défaut MEM_DEPTH = 8 et MEM_WIDTH = 16. Si vous souhaitez utiliser d'autres paramètres, vous pouvez les spécifier comme ceci : make MEM_DEPTH=4 MEM_WIDTH=12.

Vous avez également la possibilité d'exécuter chaque script séparément. Pour cela, naviguez vers le répertoire où le script se trouve et exécutez la commande python3 nom_de_script.py. Vous pouvez également utiliser les options -d et -w pour modifier les valeurs par défaut des paramètres.

6.2 Exécution de test bench

Pour lancer le test bench généré, exécutez le Makefile dans le répertoire test_bench_generator. Ce Makefile utilise GHDL pour synthétiser et simuler le code VHDL des composants, ainsi que Gtkwave pour visualiser les signaux du système.

L'exécution de ce Makefile utilise un fichier waveform.gtkw pour afficher un ensemble de signaux. Ce fichier a été généré avec Gtkwave v3.3.104. Si vous utilisez une autre version, il se peut qu'aucun signal ne soit affiché. Pour simplifier, après la première exécution,

cliquez sur les signaux affichés dans la fenêtre de Gtkwave pour les afficher, puis enregistrez l'état de Gtkwave sous la forme d'un fichier waveform.gtkw. Lors des exécutions suivantes, les signaux seront affichés dès le début.

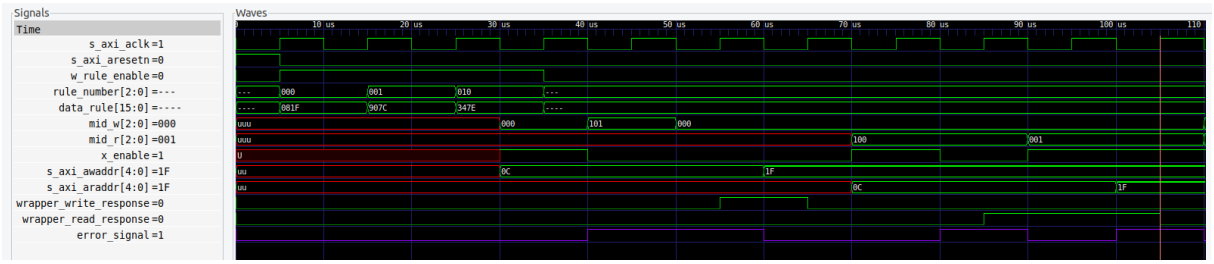

FIGURE 3 – Affichage des signaux avec Gtkwave

Cliquez ici pour accéder au code complet du projet.