HEIG VD HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

Laboratoire n°03

Détection de spike - Traitement de signal

Département : TIC

Unité d'enseignement VSE

Auteur: **Andrè Costa & Alexandre Iorio**

Professeur: **Yann Thoma**

Assistant : **-**

Salle de labo : **A07**

Date : **13.12.2024**

Table des matières

1. Introduction

Ce laboratoire à pour but de valider l'interaction entre un logiciel et un matériel de détection de signaux électriques émis par des neurones réels. Le système utilise un FPGA connecté via une interface Avalon pour détecter et stocker des fenêtres d'intérêt autour des spikes.

Il nous est demandé de refactoriser le code fourni pour le rendre plus lisible et plus modulaire. Ensuite, il faut ajouter la possibilité d'arrêter et de redémarrer l'acquisition de données. Pour finir, il faut ajouter la possibilité de définir un fichier de simulation pour tester le DUV.

2. Refactorisation

Actuellement, la classe `FPGAAccess` mélange des fonctionnalités de bas niveau permettant de faire des accès directs aux registres du FPGA, et de plus haut niveau, permettant de gerer les commandes et les interruptions. Cette architecture n'est pas optimale et ne permet pas la séparation et les généralisations du code.

Pour résoudre ce problème, nous avons décidé de factoriser le code en créant une interface `FpgaAccess` qui définit les méthodes nécessaires à la communication avec la `FPGA`, `FpgaAccessRemote` qui implément `FpgaAccess` qui s'occupe de l'accès au FPGA via un serveur TCP par le biais de la simulation, et pour terminer, une classe `SpikeDetector` qui s'occupe de la détection des spikes.

2.1. FpgaAccess

Cette interface permet de définir les méthodes nécessaires pour s'interfacer avec une FPGA. Notamment, des méthodes telle que `void setup()` permettant de configurer la `FPGA`, demmarer le serveur sur un port specifique et instancier un thread pour la réception des données.

Quant aux methodes `void write_register(uint16_t reg, uint16_t value)` et `uint16_t read_register(uint16_t reg)` permettent respectivement d'écrire et de lire des registres de la `FPGA`.

Enfin, la méthode `void set_callback(irq_handler_t)` permet de définir une fonction de callback pour les interruptions.

```
typedef void (*irq_handler_t)(const std::string &);

class FpgaAccess {
  public:
    virtual void setup() = 0;
    virtual void write_register(uint16_t reg, uint16_t value) = 0;
    virtual uint16_t read_register(uint16_t reg) = 0;
    virtual void set_callback(irq_handler_t) = 0;
};
```

Nous avons créé une interface `FpgaAccess` qui est responsable de fournir les méthodes nécessaires pour s'interfacer avec une FPGA. Notamment, des méthodes pour configurer l'accès (`setup`), lire et écrire des registres (`read/write_register`) et encore pour définir une fonction de callback pour les interruptions.

2.2. FpgaAccessRemote

La classe `FpgaAccessRemote` implémente l'interface `FpgaAccess` et est responsable de fournir les méthodes de l'interface avec le protocole custom TCP fourni.

De plus, elle possède une méthode `void set_simulation_file(const char *path)` qui permet de définir un fichier de simulation pour tester le DUV. En effet, il est intéressant de pouvoir executer les tests avec différents fichiers de données comprenants des données différentes. Cela permet de garantir que le spike detector fonctionne correctement. Cette methode permet alors d'envoyer des fichiers de simulation à la FPGA.

Nous parlerons de cette méthode plus en détail dans la section 5. Tests d'intégration.

```
struct SetupOptions {
  bool wait_for_connection;
  uint16_t port;
};

class FpgaAccessRemote : public FpgaAccess {
  public:
    FpgaAccessRemote(SetupOptions opts);
    ~FpgaAccessRemote();

    void setup();
    void write_register(uint16_t reg, uint16_t value);
    uint16_t read_register(uint16_t reg);
    void set_callback(irq_handler_t);
    void set_simulation_file(const char *path);
}
```

2.3. SpikeDetector

Grâce à cette classe, nous avons la possibilité de nous abstraire de l'accès au DUV. Ici, on ne parle plus de registres, nous sommes maintenant dans un concept plus haut niveau. On parle d'`Acquisition`, `DataReady`, etc.

La classe `SpikeDetector` est construite avec un objet de type `FpgaAccess`, elle pourra ainsi faire abstraction du protocole de communication utlisé pour accéder à la FPGA.

```
typedef void (*on_message_cb)(const std::string &);
class SpikeDetector {
  public:
    SpikeDetector(std::shared_ptr<FpgaAccess> access, on_message_cb cb);
    ~SpikeDetector() = default;

    void start_acquisition();
    void stop_acquisition();
    bool is_acquisition_in_progress();
    bool is_data_ready();

    uint16_t get_status();
    uint16_t get_window_address();

    bool read_window(SpikeWindow &data);
    void set_on_new_data_callback(on_message_cb);
    void set_simulation_file(const char *path);
};
```

2.4. Utilisation du code refactorisé

Maintenant que nous avons une architecture plus modulaire, l'utilisation du code est plus simple et devient notablement plus élégante.

En voici un exemple:

```
// main
detector.start_acquisition();

// spike_detector
void SpikeDetector::start_acquisition()
{
  access->write_register(1, 1);
}

// fpga_access_remote
void FpgaAccessRemote::write_register(uint16_t reg, uint16_t value)
{
  std::stringstream stream;
  stream << "wr " << reg << " " << value << std::endl;
  this->do_send(stream.str());
}
```

3. Intéractions avec l'acquisition des données

Dans cette partie, nous allons modifier le code pour permettre d'arrêter et de redémarrer l'acquisition des données.

3.1. Détecter la demande d'arrêt de l'acquisition

Dans un premier temps, il est nécessaire de pouvoir détecter une demande d'arrêt d'acquisition.

Afin de résoudre ce problème, tout d'abord, il faut détecter les demandes.

Pour cela, nous écrivons une valeur `0` ou `1` dans le registre situé à l'adresse `0x1`, respectivement pour arrêter et redémarrer l'acquisition.

```
task avalon_write(int address, int data);
 ...
  if (address == 1) begin
    if (data == 0) begin
      $display("%t Stopping acquisition", $time);
      is_active = 0;
    end else if (data == 1) begin
      $display("%t Starting acquisition", $time);
      is_active = 1;
      ->start_record;
    end
  end
```

Comme on peut le voir, nous mettons à jour le flag `is_active. Ce flag est utilisé dans la procedure suivante qui va attendre que l'événement start_record` soit déclenché pour continuer l'acquisition.

```
while (!$feof(
    fd
)) begin
  ret = $fscanf(fd, "%d", val);
  if (!is_active) begin
    $display("%t Acquisition Stopped. Waiting...", $time);
    wait (start_record.triggered);
    $display("%t Acquisition Restarted", $time);
  end
  ...
end
```

4. Tests unitaires

Cette partie décrie les tests unitaires que nous avons implémentés pour valider le bon fonctionnement du système. Grâce à la décomposition décrite au chapitre 2. Refactorisation, il est très simple de mettre en place des tests unitaires.

4.1. Spike Detector

Pour tester le Spike Detector, il n'est pas forcément nécessaire d'avoir un accès à la `FPGA`. Nous allons donc créer une classe `MockFpgaAccess` qui implémente l'interface `FpgaAccess` et qui permet de simuler les accès aux registres et ainsi tester le bon fonctionnement de la classe `SpikeDetector`.

4.1.1. Implémentation de MockFpgaAccess

Dans ce mock, nous avons besoin de deux `struct`, la première, `Access` permet de stocker les accès aux registres, la deuxième, `Register` permet de stocker les valeurs à retourner lors de la lecture.

```
struct Access {
  bool is_read;
  int reg;
  int value;
};

struct Register {
  uint16_t address;
  uint16_t value;
};

class MockFpgaAccess : public FpgaAccess {
  public:
    MockFpgaAccess(const std::vector<Register> &registers);
    ~MockFpgaAccess() = default;

    void setup();
    void write_register(uint16_t reg, uint16_t value);

    uint16_t read_register(uint16_t reg);
    void set_callback(irq_handler_t);

    void set_simulation_file(const char *path);
    std::vector<Access> access;
    const std::vector<Register> &registers;
    const char *file_set;
    bool setup_called;
    irq_handler_t handler;
};
```

De ce Mock, les appels à `write_register` et `read_register` vont stocker les accès dans un vecteur `access`.

Pour `write_register`, on stocke l'adresse du registre et la valeur passée en paramètre.

```
void MockFpgaAccess::write_register(uint16_t reg, uint16_t value)
{
  access.push_back(Access{
    .is_read = false,
    .reg = reg,
    .value = value,
  });
}
```

pour `read_register`, on retourne la valeur stockée dans le vecteur `registers`.

```
uint16_t MockFpgaAccess::read_register(uint16_t reg)
{
  for (const auto &r : registers) {
    if (r.address == reg) {
      access.push_back(Access{
        .is_read = true,
        .reg = reg,
        .value = r.value,
      });
      return r.value;
    }
  }
  return 0xFFFF;
}
```

4.1.2. Tests unitaires de Spike Detector

Comme nous avons un contrôle complet sur la classe `MockFpgaAccess` nous pouvons vérifier la totalité des méthodes de l'interface `FpgaAccess`.

Voici l'explication des tests unitaires que nous avons implémentée pour valider le bon fonctionnement de la classe `SpikeDetector`.

`TestSpikeDetector.SetupGetsCalledAndHandlerGetsSet`

- **Objectif** : Vérifier que l'initialisation du `SpikeDetector` appelle `setup()` et que le gestionnaire d'interruption (`handler`) est correctement défini.
- **Méthode** : Instancie `SpikeDetector` et vérifie que `setup_called` est vrai et que `handler` est correctement enregistré.

`TestSpikeDetector.TestThrowsErrorOnNullArgs`

- **Objectif** : Vérifier que la création d'un `SpikeDetector` avec des arguments `nullptr` génère une exception.
- **Méthode** : Utilise `ASSERT_THROW` pour s'assurer qu'une exception `std::invalid_argument` est levée lorsqu'un pointeur nul est passé.

`TestSpikeDetector.TestSetNewDataCallback`

- **Objectif** : Vérifier que la modification du gestionnaire de données (`set_on_new_data_callback`) fonctionne correctement.
- **Méthode** : Définit un premier gestionnaire, le modifie, et s'assure que la modification a bien été prise en compte.

`TestSpikeDetector.TestThrowsErrorOnNullCallback`

- **Objectif** : Vérifier que l'affectation d'un gestionnaire de données `nullptr` génère une exception.
- **Méthode** : Utilise `ASSERT_THROW` pour s'assurer qu'une exception `std::invalid_argument` est levée.

`TestSpikeDetector.StartStopAcquisition`

- **Objectif** : Vérifier que `start_acquisition()` et `stop_acquisition()` effectuent les bonnes écritures sur le FPGA.
- **Méthode** : Vérifie que les registres corrects sont écrits (`reg 1` à `1` pour démarrer et `0` pour arrêter).

`TestSpikeDetector.TestAcquisitionInProgress`

- **Objectif** : Vérifier si `is_acquisition_in_progress()` détecte correctement l'état de l'acquisition.
- **Méthode** : Modifie la valeur du registre de statut et vérifie que la méthode retourne les bonnes valeurs en fonction de la valeur du registre.

`TestSpikeDetector.TestDataReady`

- **Objectif** : Vérifier si `is_data_ready()` détecte correctement si une fenêtre de données est disponible.
- **Méthode** : Change la valeur du registre et vérifie que la méthode retourne `true` ou `false` correctement.

`TestSpikeDetector.TestStatus`

- **Objectif** : Vérifier que `get_status()` retourne correctement la valeur du registre de statut.
- **Méthode** : Compare les valeurs retournées avec celles du registre.

`TestSpikeDetector.TestReadWindow`

- **Objectif** : Vérifier que `read_window()` lit correctement une fenêtre de données depuis la mémoire du FPGA.
- **Méthode** :
  - Initialise une mémoire avec des données.
  - Vérifie que chaque valeur lue correspond bien à la valeur attendue.
  - Vérifie que les bonnes lectures ont été effectuées.

`TestSpikeDetector.SetFile`

- **Objectif** : Vérifier que `set_simulation_file()` enregistre correctement le chemin du fichier.
- **Méthode** : Compare le chemin défini avec celui attendu.

4.2. FpgaAccessRemote

La classe `FpgaAccessRemote` est un peu plus complexe, comme elle ouvre un serveur TCP, il faut se connecter dessus et vérifier ce qu'il se passe lorsqu'on envoie des données.

4.2.1 Explication du fonctionnement de la classe

L'implémentation initiale de la classe `FpgaAccessRemote` ouvrait un serveur TCP et attendait la connexion d’un client avant de retourner de la fonction `setup`. Cela était pratique pour l’application finale, évitant toute tentative de communication avec le FPGA avant qu'un client ne soit connecté.

Cependant, pour les tests unitaires, cette approche posait un problème : impossible de se connecter tant que le serveur n’était pas ouvert, mais la fonction `setup` ne retournait pas tant qu’une connexion n’était pas établie.

Plutôt que d’utiliser des threads complexes, la solution adoptée a été de modifier l'implémentation :

- **Le serveur est d’abord ouvert**, puis un thread est lancé pour attendre les connexions entrantes.

```
void FpgaAccessRemote::start_server(uint16_t port)
{
  // Code pour ouverture du serveur
  ...
  // Démarre un thread pour attendre la connexion du client
  listener_thread =
    std::thread(&FpgaAccessRemote::accept_connection, this, sockfd);
}
```

- **Une option de configuration** permet de choisir si l'on doit attendre la connexion d'un client avant de poursuivre.

```
void FpgaAccessRemote::setup()
{
    start_server(opts.port);
    rx_thread = std::thread(&FpgaAccessRemote::receiver, this);

    if (opts.wait_for_connection) {
        wait_connection();
    }
}
```

Cette modification facilite les tests unitaires en permettant d'exécuter `setup()` sans bloquer l'exécution, tout en conservant le comportement initial pour l'application finale.

4.2.2. Tests unitaires de FpgaAccessRemote

Voici les tests unitaires que nous avons implémenté pour valider le bon fonctionnement de la classe `FpgaAccessRemote`.

`TestFpgaAccessRemote.SetupStartServer`

Objectif : Vérifier que `FpgaAccessRemote` ouvre bien un serveur TCP et accepte une connexion après l'appel à `setup()`.
Méthode :

- Avant `setup()`, une tentative de connexion doit échouer.
- Après `setup()`, la connexion doit être acceptée avec succès.

`TestFpgaAccessRemote.WriteRegister`

Objectif : Vérifier que `write_register()` envoie correctement une commande d'écriture sur le socket TCP.
Méthode :

- `write_register(1, 2)` doit envoyer la commande `"wr 1 2\n"`.
- Vérifie que la commande envoyée est correctement reçue sur le socket.

`TestFpgaAccessRemote.ReadRegister`

Objectif : Vérifier que `read_register()` envoie bien la commande et lit correctement la réponse du serveur.
Méthode :

- Envoie la commande `"rd 1\n"`.
- Écrit manuellement `"1 10\n"` sur le socket pour simuler une réponse.
- Vérifie que `read_register(1)` retourne bien `10`.

`TestFpgaAccessRemote.HandlerIsCalledOnIrq`

Objectif : Vérifier que lorsqu'un message d'interruption (`irq`) est reçu, le gestionnaire est bien déclenché.
Méthode :

- Définit un gestionnaire `nullptr` et vérifie qu'il ne cause pas de crash.
- Envoie un message `"irq my fancy message\n"`.
- Utilise une variable de condition pour attendre l'exécution du gestionnaire et vérifie qu'il a bien reçu le message.

`TestFpgaAccessRemote.EndTestMessageIsSent`

Objectif : Vérifier que lors de la destruction d'un objet `FpgaAccessRemote`, un message `"end_test\n"` est bien envoyé.
Méthode :

- Ferme proprement la connexion et vérifie que le message `"end_test\n"` est reçu sur la socket.

5. Tests d'intégration

Pour tester l'intégration du **Spike Detector**, il est nécessaire de le valider avec différentes séries de données. Cela permet de s'assurer que le système fonctionne correctement dans divers scénarios. Pour ce faire, une commande spéciale `set_file <file>` a été ajoutée, permettant d'envoyer un fichier de simulation au testbench avant le début de la simulation.

5.1 Définition du fichier de simulation

Dans `SystemVerilog`, un événement est déclenché lorsqu'un fichier est défini via cette commande. Cette action est obligatoire et ne permet pas de modifier le fichier en cours d'exécution, car cela rendrait la vérification trop complexe, voire impossible.

```
// Déclaration de l'événement
event input_file_set;

// Gestion de la réception de la commande
...
else if (command == "set_file") begin
  ret = $sscanf(recv_msg, "set_file %s", input_file);
  ->input_file_set;
end
...

// Attente de la définition du fichier avant de commencer l'acquisition
...
wait (input_file_set.triggered);
...
```

5.1.2 Génération des fichiers de simulation

Afin de disposer de jeux de données variés, un script **Python** a été développé pour générer plusieurs fichiers de simulation contenant des valeurs aléatoires ou prédéfinies.

```python
content = "\n".join([str(i) for i in range(900)])

with open("linear.txt", "w") as f:
    f.write(content)

content = "0\n" * 900

with open("zeros.txt", "w") as f:
    f.write(content)

content = ("0\n" * 70 + "1000\n" + "0\n" * 100) * 17 + ("0\n" * 200)

with open("constant_spikes_16_windows.txt", "w") as f:
    f.write(content)
```

Les fenêtres générées sont de taille réduite afin de permettre des simulations rapides et de faciliter le débogage. Le fichier **constant_spikes_16_windows.txt** est particulièrement intéressant, car il contient exactement le même nombre de fenêtres que celles stockées en mémoire par le **DUV** (Device Under Verification).

5.2. Finalisation de la simulation

Une amélioration importante a été apportée à l'implémentation originale : l'envoi d'un message **Simulation → Software** pour indiquer que toutes les données du fichier ont été traitées. Cette optimisation accélère considérablement les tests, car auparavant, l'arrêt du système reposait sur un `timeout` de cinq minutes après le dernier `irq`.

Dans `SystemVerilog`, la tâche qui lit les données du fichier attend quelques cycles d'horloge après avoir traité la dernière donnée, puis envoie un message de fin au logiciel.

```
// Attente après la dernière donnée pour vérifier s'il y a d'autres interruptions
##150;
client.send_mbox.put("irq end\n");
$display("Simulation is over");
```

Côté `C++`, le logiciel détecte ce message et interrompt la boucle d'attente des interruptions.

```
// main
while (irqCondVar.wait_for(lk, std::chrono::seconds(600),
        [] { return !irqFifo.empty(); })) {
  std::string value = irqFifo.back();
  irqFifo.pop();
  if (strstr(value.c_str(), "end")) {
    break;
  }
  ...
}
```

5.3. Gestion du port TCP pour l'exécution parallèle

Même avec l'optimisation de la fin de simulation, les tests utilisant `input_values.txt` restent longs (~5 minutes). Pour accélérer le processus, plusieurs simulations peuvent être exécutées en parallèle. Chaque instance de simulation et son logiciel de test doivent alors utiliser un port `TCP` différent.

Dans `C++`, le port est défini dans la structure `SetupOptions` et utilisé lors de l'ouverture du serveur.

```
struct SetupOptions {
  bool wait_for_connection;
  uint16_t port;
};
```

Dans `SystemVerilog`, ce paramètre est passé au programme en tant que paramètre générique.

Afin d'adapter l'environnement de test, il est nécessaire de modifier le **Makefile** et le script `arun.sh` pour prendre en compte ce port dynamique.

Dans `arun.sh`, le script crée un répertoire distinct pour chaque instance, basé sur le numéro de port :

```
run_with_questa() {
    if [ -d work${SERVER_PORT} ]; then
        rm -rf work${SERVER_PORT}
    fi
    mkdir work${SERVER_PORT}

    cd work${SERVER_PORT}
    vlib work${SERVER_PORT}
    ...
}
```

Le **Makefile** est ajusté pour transmettre le port au simulateur avec l'option `-GPORT=$(SERVER_PORT)`.

```
run_questa: build_questa
    cd work$(SERVER_PORT); vsim -64 amiq_top -do vsim_cmds.do -lib work -GPORT=$(SERVER_P
```

Enfin, dans `SystemVerilog`, la valeur du port est récupérée et utilisée lors de l'initialisation de la connexion :

```
module amiq_top #(
    int ERRNO = 0,
    int PORT  = 8888
);
...
  amiq_server_connector #(
      .hostname("127.0.0.1"),
      .port(PORT),
      .delim("\n")
  ) client = new();
...
```

Avec ces modifications, plusieurs simulations peuvent être exécutées simultanément, réduisant ainsi le temps total de validation du **Spike Detector**.

5.4 Généralisation des tests d'intégration

Dans le logiciel, les différents tests à exécuter suivent une structure similaire. Afin d'éviter de dupliquer du code, une fonction générique a été créée pour prendre en charge les tests en paramétrant les aspects spécifiques.

Cette fonction reçoit plusieurs paramètres, notamment deux fonctions de callback :

- Une pour gérer la réaction à un `irq`.
- Une autre pour traiter les données lorsqu'une fenêtre est lue.

```
typedef bool (*on_irq_trigger_t)(const std::queue<std::string> &,
                 SpikeDetector &);
typedef void (*on_window_read_t)(SpikeDetector &);

void test_file(const char *simulation_file, uint16_t port,
           size_t expected_spike_nb, on_irq_trigger_t on_irq,
           on_window_read_t on_window_read);
```

5.5 Exécution des tests d'intégration

Grâce à cette préparation, plusieurs fichiers de simulation peuvent être testés en parallèle.

Voici la liste des test et leur explication:

`Integration.LinearNoSpikes`

Objectif : Vérifier que le DUV ne détecte aucun spike dans un signal linéaire.
Méthode : Exécute le test avec `linear.txt`, un fichier contenant une séquence linéaire de valeurs, et s'assure qu'aucun spike n'est détecté (`expected_spike_nb = 0`).

`Integration.Zeros`

Objectif : Vérifier que le DUV ne détecte aucun spike lorsque le signal est constant à zéro.
Méthode : Utilise `zeros.txt`, un fichier ne contenant que des valeurs nulles, et s'assure qu'aucun spike n'est détecté (`expected_spike_nb = 0`).

`Integration.StopAcquisitionsWhileReading`

Objectif : Tester la robustesse du système lorsqu'on interrompt et redémarre l'acquisition en cours de test.
Méthode :

- Utilise `constant_spikes_16_windows.txt`, un fichier contenant des spikes réguliers.
- Lorsqu'un `irq` est reçu, l'acquisition est arrêtée (`on_irq_stop_acquisition`).
- Lorsqu'une fenêtre est lue, l'acquisition est relancée (`on_window_read_restart_acquisition`).

`Integration.AccumulateAndReadAtTheEnd`

Objectif : Vérifier que les spikes détectés restent accessibles même si aucune fenêtre n'est lue immédiatement.

Méthode :

- Utilise `constant_spikes_16_windows.txt` pour générer des spikes.
- L'acquisition est arrêtée dès qu'un `irq` est reçu (`on_irq_stop_acquisition`).
- Les fenêtres ne sont pas lues immédiatement mais accumulées et récupérées en fin de test.

`Integration.RandomSpikes`

Objectif : Vérifier la détection correcte des spikes avec des valeurs réalistes.
Méthode :

- Utilise `input_values.txt`, un fichier simulant un signal proche de la réalité.
- Lorsqu'un `irq` est reçu, une lecture de fenêtre est déclenchée (`on_irq_read_window`).
- L'acquisition continue normalement sans modification (`on_window_read_do_nothing`).
- Vérifie que 52 spikes sont bien détectés (`expected_spike_nb = 52`).

Ces tests couvrent plusieurs scénarios : **absence de détection erronée, robustesse aux interruptions d'acquisition et validité des résultats sur des signaux réalistes**.

5.6 Exécution des tests

5.6.1 Tests unitaires

Les tests unitaires sont situés dans `embeded_soft/test/unit`. Pour les exécuter :

```
cmake -S . -B build
cmake --build build
./build/test_spike_detector
./build/test_fpga_access_remote
```

5.6.2 Tests d'intégration

1. **Génération des fichiers de simulation**
   - Dans le répertoire `simulation_files` :

   ```
   python3 generate.py
   ```

2. **Compilation des tests d'intégration**
   - Dans `embeded_soft/test/integration` :

   ```
   cmake -S . -B build
   cmake --build build
   ```

3. **Lister les tests disponibles**

```
./build/test_integration --gtest_list_tests
```

4. **Lancer un test spécifique**
   - Exemple avec `Integration.RandomSpikes` :

```
export SERVER_PORT=8888
./build/test_integration --gtest_filter=Integration.RandomSpikes
```

5. **Exécuter la simulation dans `fpga_sim`**

```
export PROJ_HOME=${PWD}
export SERVER_PORT=8888
./arun.sh -tool questa
```

   Il est possible de lancer plusieurs simulations en parallèle en modifiant la valeur de `SERVER_PORT`.

5.6.3 Script pour exécuter tous les tests

Un script est fourni pour exécuter tous les tests simultanément. Celui-ci ouvre plusieurs instances de `questasim`, ce qui peut entraîner une charge CPU importante.

Attention, il est donc judicieux de vérifier la température de la pièce, en effet la charge CPU extrême peut entrainer une surchauffe de la pièce. Dans ce cas, il est recommandé de poser sa machine à l'exterieur.

Ce script a été testé sous **Fedora 41** et dans **VM Reds**, mais il est fourni sans garantie. Assurez-vous que votre système est connecté au **VPN** avant d'exécuter `questasim`.

Dans le répertoire `code`, exécutez :

```
python run_tests.py
```

Ces étapes permettent une validation rapide et robuste du **Spike Detector** sur différentes configurations et fichiers de test.

6. Conclusion

Ce laboratoire a permis de refactoriser et d'améliorer la modularité du code en séparant les responsabilités des différentes classes (FpgaAccess, FpgaAccessRemote, SpikeDetector). Grâce à cette nouvelle architecture, l'ajout de fonctionnalités, comme l'arrêt et la reprise de l'acquisition, a été facilité.

Un effort particulier a été porté sur la validation du système à travers des tests unitaires et tests d'intégration rigoureux, couvrant divers scénarios. L'utilisation de fichiers de simulation et la parallélisation des tests ont permis d'assurer la fiabilité du Spike Detector.

Enfin, des optimisations ont été mises en place, notamment pour accélérer la fin des simulations et gérer dynamiquement le port TCP, garantissant ainsi un environnement de test performant et évolutif.