École thématique — Archi 2023
Introduction à `Chisel`

Bruno FERRES
Inria Lyon

Préambule

Le but de ce TP est de présenter rapidement l'utilisation du langage `Chisel` pour la conception de circuits numériques. En particulier, ces exercices devraient vous permettre d'acquérir les bases de syntaxe `scala`, ainsi que les constructions de bases du langage.

Dépendances

Les dépendances nécessaires sont les suivantes :
- `scala` : à installer avec votre gestionnaire de paquet préféré
- `java` : il est recommandé d'installer le JDK 8 ou plus récent
- `sbt` : l'outil de *build* le plus utilisé par la communauté `scala` (voir les consignes d'installation)
- `gtkwave` : un outil de visualisation des résultats d'une simulation RTL (sous forme de chronogrammes) — normalement disponible via votre gestionnaire de paquet. C'est un outil qui peut s'avérer très utile pour débugger votre design (voir Annexe B).

Structures du projet

La structure classique d'un projet en `Chisel` est basée sur la structure de projet de `scala`[1]. En particulier, les sources sont dans le répertoire `src` (étonnant, me direz vous), qui est constitués de deux sous dossiers :
- les descriptions des circuits, dans le sous répertoire `src/main/scala`.
- les tests associés, dans le sous répertoire `src/test/scala`.

Vous serez également intéressés par le contenu des répertoires `test_run_dir`, qui contiendra les fichiers intermédiaires générés par les différentes simulations qui permettront de valider le fonctionnement de vos circuits, et `verilog`, qui contiendra les fichiers `verilog` générés à partir de vos descriptions en `Chisel`.
Les autres sous répertoires fournis sont nécessaires au bon fonctionnement du projet, mais il est inutile de comprendre leur structure.

1. En réalité issue du *template* `Chisel` de base, un squelette de projet pour débuter facilement.

Utilisation de `sbt`

`sbt` est un outil qui permet à la fois de gérer automatique la gestion des dépendances du projet, et sert d'interprète de commande. Pour le lancer, rien de plus simple : lancez la commande `sbt` à la racine du projet (le répertoire parent du répertoire `src`).
Les différentes commandes qui vous seront utiles sont présentées succintement ici (et seront également précisée au fil du sujet) :

- `compile` : permet de compiler les sources dans le sous répertoire `src/main/scala` (et uniquement celles ci !)
- `test:compile` : permet de compiler les sources dans les sous répertoire `src/main/scala` et `src/test/scala`
- `runMain <package>.<main>` : permet de lancer l'exécutable `main` du sous projet `scala package`
- `test` : permet de lancer tous les tests disponibles dans le sous répertoire `src/test/scala`
- `testOnly <package>.<test>` : permet de lancer le test `test` du sous projet `scala package` en particulier

Pour l'intégralité des exercices proposés, on fournit les tests nécessaires à valider vos designs, et il ne vous sera donc pas nécessaire de les modifier. Cependant, je ne peux que vous recommander de regarder les différents tests fournis, afin d'avoir une idée de leurs fonctionnements.

Exercice préliminaire : le PGCD

Afin de mieux prendre en main l'outil, j'ai décidé de laisser l'exemple type fourni dans le *template* `Chisel`. Il s'agit de deux implémentations naïves de l'algorithme du PGCD, disponibles dans le sous répertoire `src/main/scala/gcd`[2]. Le but de cet exercice est uniquement de prendre en main la syntaxe `Chisel` et les outils proposés pour le TP.

Exercice 0.1

Le fichier `GCD.scala` présente une implémentation très simple de l'algorithme, qui aurait très bien pu être réalisée dans un autre langage de description — VHDL ou (System)Verilog — à la syntaxe près.

Question 0.1.1 : Élements de syntaxe

On se propose de présenter rapidement les éléments de syntaxes de base sur cet exemple.

Utiliser `Chisel`: Tout d'abord, il est important de noter que `Chisel` n'est en réalité rien de plus qu'une **bibliothèque** de `scala`. Il faut donc penser à l'importer au début de votre fichier avec `import chisel3._` (le _ ayant pour but d'importer toutes les méthodes et valeurs du paquet `chisel3`, un peu comme le $*$ en `python`). Cela implique en outre que tout code `Chisel` correcte respecte la syntaxe de `scala` !

Module: Chaque module que vous allez développer est en réalité une classe[3], qui doit nécessairement étendre la classe `Module` de base pour que le mécanisme de génération du code se passe bien. Ce simple fichier implémente donc un module `GCD`, dont le corps est situé dans la déclaration suivante :

```
class GCD extends Module {
    ...
}
```

Entrées/Sorties: `Chisel` fournit un certain nombre de fonctionnalité pour décrire les entrées/sorties disponibles pour un module. Dans un premier temps, nous nous intéresserons uniquement à un mécanisme simple pour la gestion des entrées/sorties : les `Bundles`.
Un `Bundle` est un simple type aggrégé, permettant de définir différents signaux qui seront exposés à l'interface du module en question. Ici, 3 entrées et 2 sorties sont exposées :

```
    val io = IO(new Bundle {
        val value1        = Input(UInt(16.W))
        val value2        = Input(UInt(16.W))
        val loadingValues = Input(Bool())
        val outputGCD     = Output(UInt(16.W))
        val outputValid   = Output(Bool())
    })
```

2. Les tests correspondants sont évidemment disponibles dans `src/test/scala/gcd`.
3. Au sens de la programmation orientée-objet. Cependant, pas de soucis si vous n'en avez jamais fait, ce n'est pas nécessaire pour comprendre ce TP.

On a ici un premier aperçu de la gestion des types en `Chisel` : le type `UInt(w)` correspond à un type entier non signé sur w bits, et le type `Bool`... est un type booléen.
Les entrées/sorties sont accessibles (en écriture pour les sorties, en lecture pour les entrées) en utilisant `io.value1` par exemple.

Déclaration des signaux internes: En plus des entrées/sorties, il est également possible de déclarer des signaux internes à votre circuit. Ici intervient une première distinction avec les HDLs classiques : s'il est possible de déclarer des signaux combinatoires simplement avec `val my_wire = Wire(UInt(16.W))` par exemple, il est également possible de déclarer des **registres**, dont le comportement est fixé à la génération. De ce fait, au lieu de devoir déclarer un signal ET un processus synchrone pour générer un registre — comme c'est le cas en VHDL ou en Verilog — une simple déclaration `val x = Reg(UInt())` suffit pour déclarer un registre `x`[4].

Description RTL: Après la définition des signaux externes et internes disponibles, il ne reste plus qu'à décrire le comportement de notre circuit. Pour ce faire, la syntaxe est relativement semblable à celles des HDL classiques :

— il est possible d'exprimer des structures conditionnelles de type multiplexeur avec la construction suivante (qui correspond au `if`/`else` `verilog`) :

```
when(cond){
    ...
}.otherwise{
    ...
}
```

— l'affectation se fait grâce à l'opérateur `:=`. Cet opérateur prend en compte le type sous-jacent des données : si `x` est un registre, alors `x := y` modifiera la valeur de `x` au prochain cycle, alors que si `x` est un simple fil, l'affectation se fera dans le même cycle.

— les opérations sur les types de bases (par exemple, les entiers) sont définis en surchargeant les opérateurs `scala`. Cela signifie que, pour deux signaux `a` et `b` de type `UInt`, l'opération `a + b` correspondra bien à une addition matérielle de deux signaux non signés.

— il est également possible de réaliser des tests simples sur les signaux manipulés, comme par exemple `x > y`. Cependant, pour l'égalité, il faut penser à utiliser l'opérateur `===`, puisque l'opérateur `==` de `scala` correspond à l'égalité physique du langage.

— On peut utiliser des constantes (soit pour l'affectation, soit pour la comparaison) grâce à des constructions du langage pour les types de base. Par exemple, on peut comparer un signal `y` de type `UInt` à la valeur 0 avec `y === 0.U`. La valeur `0.U` représente le 0 du type *unsigned*.

4. A noter également qu'on peut ne pas spécifier la largeur du type `UInt` dans ce cas là — le processus de génération va inférer une *bitwidth* minimale de façon statique pour les registres `x` et `y`.

<u>Flot de contrôle:</u> Dans ce circuit, le flot de contrôle est très simple, et se base sur les constructions `when` et `otherwise`. L'entrée `io.loadingValues` permet de préciser que nous sommes dans un cycle de chargement des valeurs d'entrée (donc, un cycle où l'on doit recopier les valeurs de `io.value1` et `io.value2` sur les registres `x` et `y`). La sortie `io.outputValid` précise quand le calcul est fini, et que la valeur que l'on peut lire sur `io.outputGCD` correspond bien à la sortie attendue.

<u>Question 0.1.2 : Simulation</u>

Maintenant que vous avez un peu pris en main la syntaxe de `Chisel`, on se propose de vérifier que le circuit en question respecte bien sa spécification — c'est à dire qu'il calcule bien le PGCD des entrées `io.value1` et `io.value2` lorsque le signal de contrôle `io.loadingValues` est correctement positionné.
Pour vérifier le comportement de notre circuit, on fournit le fichier `GcdSpec.scala`, commenté pour l'occasion pour préciser le déroulement du test.
Après avoir analysé le contenu du test, lancez la simulation depuis `sbt` :

```
sbt > testOnly gcd.GcdSpec
```

Outre le fait que tout doit bien se passer (sortie en vert dans `sbt`), une **trace de simulation** est générée au format VCD (*Value Change Dump*) dans le répertoire `test_run_dir/GCD_should_respect_its_spec`. Vous pouvez visualiser son contenu avec `gtkwave` (voir Annexe B), afin de dérouler l'algorithme en suivant la valeur des différents signaux du circuit, comme illustré en Figure 1.

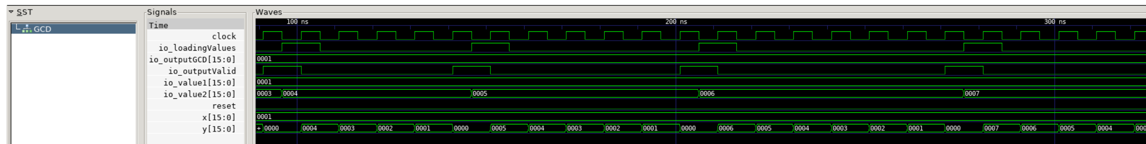

Figure 1 – Trace de la simulation du module `GCD` avec `gtkwave`

Pour utiliser `gtkwave`, il est nécessaire de sélectionner l'ensemble des signaux que l'on veut observer dans les panneaux de gauche [5] — par défaut, l'outil ne suit aucun signal. N'oubliez pas également de dézoomer (avec la molette de la souris, par exemple) afin d'adapter l'échelle de visualisation à quelque chose semblable à celle présentée en Figure 1. Cette phase de simulation vous sera sûrement utile par la suite, afin de débugger vos circuits — n'hésitez pas à m'appeler si vous n'arrivez pas à prendre l'outil en main.

5. Le panneau supérieur correspond à la hiérarchie du circuit (ici, il n'y a qu'un module), et le panneau inférieur correspond, pour un module donné, aux signaux de simulation qui ont été tracés — en double cliquant sur chaque signal, celui ci sera ajouté à la visualisation en cours.

Question 0.1.3 : Génération depuis Chisel

Afin d'avoir une meilleure idée du fonctionnement du processus de génération de `Chisel`, on vous propose de générer le code `verilog` correspondant au module `GCD`.
Pour ce faire, dans `sbt`, lancez la commande `runMain gcd.Emitter`, qui va générer un dossier `verilog/GCD`.

```
sbt > runMain gcd.Emitter
```

Dans ce dossier, regardez le fichier `GCD.v`, qui correspond donc au résultat de la génération basée sur le module `GCD` précédemment décrit. Vérifier que les différentes étapes de l'algorithme sont bien générées.

Remarque: La génération de `verilog` depuis `Chisel` ayant pour but d'être portable pour différents usages (simulation, synthèse, ...) et ce sur plusieurs *backend*, un certain nombre de directives sont insérées à la génération, dans des blocs `‘ifdef...‘endif`. Vous pouvez ne pas tenir compte de ces blocs pour ce TP [6].

Question 0.1.4 : Synthèse logique

On peut finalement s'intéresser à l'étape de synthèse logique, en s'appuyant sur l'outil `vivado`, installé sur le serveur.
Pour ce faire, rien de plus simple si vous avez correctement mis en place l'accès au serveur (voir Annexe A). Lancez la commande `make synthesis VERILOG=verilog/GCD/GCD.v`.
L'outil vous fournira un rapport succinct des ressources nécessaires pour implémenter le circuit, en terme de primitives de calculs. Ces primitives sont les suivantes :
- **LUTs** (Look-Up Tables) : structure de base pour l'implémentation générique des calculs. Permet d'implémenter n'importe quelle fonction booléenne.
- **Regs** (Registers, ou Flip Flops) : structure de base pour la mémorisation. Chaque registre permet de mémoriser **un bit**.
- **BRAMs** (Block RAMs) : unité de mémorisation un peu plus grande que les registres. C'est un équivalent de la RAM de vos machine, directement embarquée sur la FPGA.
- **DSPs** (Digital Signal Processors) : unités de calculs dédiés pour certaines opérations. Cela permet notamment d'implémenter de façon efficaces des multiplications, voir des MAC (*Multiply-and-Accumulate*).

En outre, le rapport de synthèse vous fournit également une première estimation de la fréquence opérative de votre circuit sur la FPGA. La fréquence estimée correspond à la fréquence maximale à laquelle le circuit peut fonctionner sans compromettre la validité des calculs — en fonctionnant plus rapidement, un nouveau cycle d'horloge pourrait commencer alors que des signaux sont encore en cours de calcul (notion de **chemin critique**), et ces résultats intermédiaires pourraient alors être perdus ! Les performances de votre accélérateur sont grandement liées à cette fréquence, il est donc essentiel d'avoir accès à cette métrique afin de pouvoir estimer la qualité d'un accélérateur, notamment pour comparer différentes implémentations entre elles.

6. Ces blocs ont surtout pour intérêt de générer des valeurs aléatoires sur les signaux internes lors de la simulation, afin de détecter des bugs qui seraient par exemple liés à des signaux mal/non initialisés.

1 Implémentation d'un produit scalaire

Dans cette partie, on s'intéresse maintenant à l'implémentation de l'algorithme de calcul du produit scalaire en `Chisel`, afin de démontrer les possibilités du langage.
Pour ce faire, on va considérer trois spécifications de l'algorithme. On partira d'une spécification très simple, qui pourrait facilement être implémentée en VHDL/Verilog, et on aboutira à une implémentation bien plus paramétrique, mettant en avant la nécessité d'avoir des fonctionalités de génération avancées.

Remarque générale La prise en main de l'écosystème `Chisel` peut être compliquée au début. Le but de ce TP étant de vous permettre de découvrir l'outil, n'hésitez pas à m'appeler si vous êtes bloqués (notamment, si vous avez des erreurs de compilation dont le message ne vous évoque rien).

Architecture du TP Afin de vous faciliter la prise en main de `Chisel`, on vous fourni un squelette pré-rempli pour chaque module à implémenter dans cet exercice. En particulier, ces squelettes définissent les interfaces des modules, et **vous ne devez pas modifier comment sont déclarées ces interfaces** (variable `io`). Cela nous permet, entre autre, de vous fournir des bancs de tests pour valider le comportement de vos implémentations en simulation.

Exercice 1.1 Produit scalaire combinatoire et non paramétré

Pour cet exercice, on s'intéressera au fichier `FixedDotProduct.scala`, dans le sous répertoire `src/main/scala/dot` (ainsi qu'au fichier de test associé `FixedDotProductSpec.scala`).
La spécification du circuit est simple : on fournit deux vecteurs `io.vec1` et `io.vec2` de 4 entiers non signés sur 8 bits, et le circuit calcule **en un cycle** le résultat `io.out` du calcul suivant :

$$
\begin{aligned}
io.out &= (io.vec1_0 * io.vec2_0) + (io.vec1_1 * io.vec2_1) + (io.vec1_2 * io.vec2_2) + (io.vec1_3 * io.vec2_3) \\
&= \sum_{i=0}^{3} io.vec1_i * io.vec2_i
\end{aligned}
$$

Aucun signal de synchronisation n'est donc nécessaire ici.
On fournit le squelette du module `FixedDotProduct`, ainsi que les entrées/sorties et signaux internes nécessaires.

Question 1.1.1 : Précision sur les types de donnée

Que pouvez vous remarquer sur les types des signaux `io.out`, `mult`, `tmp1` et `tmp2` ? Que pouvez vous en déduire sur le processus de génération de ce module, par rapport à ces signaux ?
Quel serait l'intérêt de spécifier la taille de ces signaux, plutôt que de laisser l'outil de génération l'inférer ?

Remarque Bien que cela ne soit pas nécessaire, on affecte la valeur `DontCare` (une valeur par défault) aux différents signaux, afin de préciser où l'on s'attend à ce que vous modifiez le code dans ce premier exemple. En réalité, si vous tentez d'élaborer ce circuit avec les `DontCare`, le processus d'élaboration levera une exception, car il n'arrive pas à inférer la taille des signaux en question.

Question 1.1.2 : Implémentation

Il s'agit maintenant d'implémenter, dans le squelette fourni, le comportement attendu. Pour ce premier module, vous devriez pouvoir vous en sortir en décrivant séquentiellement les différentes affectations nécessaires à implémenter les calculs requis.

Multiplication: Tout d'abord, on fournit un premier vecteur [7] `mult` permettant de stocker les valeurs intermédiaires des calculs de produit ($io.vec1_i * io.vec2_i$). Pour remplir ce vecteur, on va pouvoir utiliser la construction `for` de `Chisel` [8] pour générer les différentes affectations. Pour chaque indice `i`, vous pouvez accéder à la `i`ème valeur d'un vecteur `vect` grâce à `vect(i)`.

Réduction binaire: Maintenant que le vecteur `mult` contient bien les résultats intermédiaires de type $io.vec1_i * io.vec2_i$, on peut calculer le produit final. Pour ce faire, utilisez les signaux fournis (`tmp1` et `tmp2`, mais également `io.out`) pour implémenter les additions nécessaires pour respecter la spécification du circuit.

Question 1.1.3 : Validation comportementale

Valider le fonctionnement de votre premier circuit grâce aux tests fournis. Vous pourrez pour ce faire regarder rapidement le contenu du fichier de test `FixedDotProductSpec.scala`, avant de lancer :

```
sbt > testOnly dot.FixedDotProductSpec
```

Remarque: Si les tests ne passent pas, `gtkwave` vous sera peu utile pour débugger : en effet, ce circuit est purement combinatoire, et analyser les chronogrammes générés est donc peu intéressant. L'outil sera beaucoup plus pertinent pour les prochains exercices.

Question 1.1.4 : Génération du circuit

Générez la description `verilog` correspondante :

```
sbt > runMain dot.FixedEmitter
```

Analysez le contenu généré dans `verilog/fixed/FixedDotProduct.v`. Vous devriez retrouver le comportement attendu

7. Vous pouvez voir cela comme un `array verilog`, ou n'importe quelle structure de tableau.
8. En réalité, c'est la construction `for` de `scala` qui est utilisée ici — lors de la génération du code `Chisel`, le code va être exécuté, et une affectation `verilog` sera générée pour chaque tour de boucle.

Question 1.1.5 : Synthèse logique

Il est maintenant nécessaire de réaliser la synthèse logique de votre circuit, afin de pouvoir caractériser à la fois ses performances, et les ressources nécessaires à son implémentation. Encore une fois, on va s'appuyer sur le `Makefile` fourni (voir Annexe A).
Lancez la commande `make synthesis VERILOG=verilog/fixed/FixedDotProduct.v`, et attendez que la synthèse termine.
Analysez le rapport de synthèse. Que pouvez vous dire sur la fréquence de fonctionnement estimée ? En réalité, le circuit est trop simple pour que l'outil de synthèse ne fournisse une estimation pertinente de la fréquence de fonctionnement : il s'agit d'un circuit purement combinatoire, et le chemin critique connecte donc un port *in* à un port *out*. Les estimations seront bien plus pertinentes dans le prochain exercice !
Vous pouvez également analyser les ressources nécessaires à l'implémentation. En particulier, conservez les fichiers de rapport de synthèse générés, pour les comparer avec les prochaines implémentations.

Exercice 1.2 Produit scalaire entier paramétré avec synchronisation *ad-hoc*

Dans cet exercice, on s'intéresse maintenant à optimiser notre implémentation de l'algorithme du produit scalaire, si possible de façon générique, afin de pouvoir réutiliser le code dans différents cas d'usages. On s'intéressera au fichier `ParametrizedDotProduct.scala`, ainsi qu'au fichier de test associé `ParametrizedDotProductSpec.scala`.

Question 1.2.1 : Paramètres de génération

Cette fois-ci, notre module `ParametrizedDotProduct` est paramétré par 5 arguments, définis directement dans le constructeur du module.
Essayez de comprendre le rôle de chaque argument. Que remarquez vous sur le type des paramètres `mulFunc` et `addFunc` ? Pourquoi le paramètre `latency` est il nécessaire ?

Remarque sur l'architecture proposée Afin de ne pas être encore confrontés aux problèmes liés à la synthèse de circuits purement combinatoires (comme dans l'exercice précédent), l'implémentation en question utilise des *buffers* sur les entrées et sorties des circuits générés. Il va donc falloir utiliser comme entrées les vecteurs `vec1Buffered` et `vec2Buffered`, afin de définir le contenu d'un vecteur `outBuffer`, qui sera recopié sur le port `io.out` sur front montant de l'horloge. Vous êtes libres de modifier la déclaration de `outBuffer`, si cela vous semble plus pratique.

Définition des paramètres Pour implémenter notre générateur d'accélérateurs de calcul de produit scalaire, vous allez devoir utiliser les paramètres suivants [9] :

- `width` : nombre de bits dans les entiers non signés en entrée
- `nElem` : nombre d'élements dans chaque vecteur d'entrée
- `mulFunc` : fonction pour réaliser les multiplications
- `addFunc` : fonction pour réaliser les additions
- `latency` : latence induite par les fonctions `mulFunc` et `addFunc` dans les calculs. Ce paramètre est utilisé pour calculer la latence entre le cycle où les vecteurs sont positionnés en entrée du circuit, et le cycle où le signal `io.out` contient le résultat attendu [10]. Ce paramètre est nécessaire, car on ne peut inférer la latence des circuits générés de façon automatique, les fonctions `mulFunc` et `addFunc` pouvant introduire des cycles de latence dans leurs définitions respectives.

Vous remarquerez que l'on peut utiliser la fonction `require` pour définir des propriétés qui doivent être respectées avant de générer un circuit. Cela signifie, par exemple, qu'une exception sera levée si l'on essaye de générer une architecture opérant sur des entiers de taille nulle ou négative, ou avec une latence négative.

Question 1.2.2 : Implémentation fonctionnelle

Cet exercice a pour but de montrer l'expressivité de `Chisel` pour définir de façon générique des calculs complexes. Par exemple, ici, il s'agit de définir un pattern *map-reduce* classique en programmation fonctionnelle. En effet, si l'on souhaitait implémenter l'algorithme en `scala`, on obtiendrait le code suivant :

```
def dot(vec1: Seq[Int], vec2: Seq[Int]) = (vec1 zip vec2)
                                          .map{ case (a, b) => a * b }
                                          .reduce (_ + _)
```

Listing 1 – Implémentation fonctionnelle du produit scalaire (version logicielle)

On voit que cette fonction opére sur 2 vecteurs d'entrée, et réalise 3 fonctions de façon successive (en supposant que *size* est la taille des 2 vecteurs) :

1. les vecteurs `vec1` et `vec2` sont **combinés** avec la fonction `zip`, afin de donner un seul vecteur qui contient les couples d'entier $(vec1_i, vec2_i), \forall i \in [\![0, size]\!]$.
2. ce nouveau vecteur est **parcouru** avec `map` : pour chaque couple (à l'index i), on calcul le produit $vec1_i * vec2_i$. La fonction *map* renvoie un **nouveau vecteur**, qui respecte la séquence initiale des indices i : $(vec1_i * vec2_i), \forall i \in [\![0, size]\!]$
3. ce nouveau vecteur est parcouru avec `reduce`, pour calculer la **somme** de tous les éléments : $\sum_{i=0}^{size-1} vec1_i * vec2_i$.

En se basant sur cet syntaxe, ainsi que sur les paramètres de génération du circuit, implémentez un générateur paramétrique pour l'algorithme cible.

9. Oui, c'est complétement la réponse à la question précédente, vous ne rêvez pas.
10. En particulier, on définit `totalLatency = latency + 2` afin de prendre en compte la synchronisation des I/O du circuit. En effet, comme les entrées et les sorties sont bufferisées (voir Remarque précédente), la latence minimale de l'accélérateur est de 2, sans compter les potentielles opérations.

En particulier, vous aurez besoin des informations suivantes :

- les endroits où insérer votre code dans le module fourni sont désignés par un commentaire `// INSTRUCTION:`.
- appuyez vous sur l'API `Chisel` en ligne pour avoir une idée des fonctions existantes dans le langage.
- implémentez les trois étapes présentées dans l'Algorithme 1.
  Vous pouvez déclarer des variables intermédiaires pour chaque étape. Les fonctions `mulFunc` et `addFunc` devront être utilisées aux étapes adéquates.
- le paramètre `width` est déjà utilisé pour définir le type des données en entrée, dans la variable `tpe`. Vous n'en aurez pas besoin ailleurs, en principe.
- de même, le paramètre `nElem` est utile pour définir les vecteurs `io.vec1` et `io.vec2`. Selon votre implémentation du générateur, vous pourriez en avoir besoin.
  Gardez cependant à l'esprit qu'en programmation fonctionnelle, on a souvent moyen de se passer de ce genre de borne lorsqu'on interagit avec un vecteur. Par exemple, au lieu de faire une boucle `for` pour itérer sur un vecteur `vec`, on peut utiliser `vec.map(func)`, sans avoir à préciser la condition d'arrêt. Ce faisant, on peut également préciser quelle fonction `func` appliquer à chaque élément de `vec`.
- vous pouvez vous inspirez de la déclaration des vecteurs `vec1Buffered` et `vec2Buffered` : la primitive `RegNext(x)` permet de créer un registre dont la valeur vaut toujours la valeur du signal `x` au cycle précédent.
- vous pourrez avoir besoin de la primitive `ShiftRegister(x, n)` pour créer un registre à décalage. Ce genre de registre est très utile pour introduire une latence fixe dans un signal : à un cycle t, le signal en sortie du registre aura la valeur qu'avait le signal `x` au cycle $t - n$. Cela peut notamment vous être utile pour définir le signal `io.outValid`.
- sur un vecteur `vec` donné contenant des signaux (par exemple, le résultat des multiplications dans le produit scalaire), vous pouvez utiliser la fonction `vec.reduceTree(func)`, qui va créer un arbre équilibré de réduction binaire appelant la fonction `func` à chaque étape. C'est un équivalent du `reduce` de l'exemple `scala` précédent, mais plus optimal pour une implémentation matérielle.
- en `scala`, il n'est pas nécessaire de déclarer le type des variables, celui-ci est inféré statiquement à la compilation. Pour déclarer un signal, il vous suffit donc d'utiliser le mot clef `val` et l'opérateur de définition `=` de la façon suivante :
  `val monSignal = a + b`.
  En revanche, si un signal (ou un registre) a déjà été déclaré (par exemple, avec `mySignal = Wire(UInt())` ou `myRegister = Reg(UInt())`), il faudra utiliser l'opérateur d'affectation `:=` pour en modifier la valeur [11] :
  `mySignal := a + b` ou `myRegister := a + b`

11. On remarque ici que les primitives `Wire` et `Reg` ont en réalité le même type : celui du signal qu'elles contiennent. La différence est la sémantique d'accès à ces valeurs : si vous utilisez `mySignal` comme opérande dans un calcul ou une affectation (par exemple, `io.out := mySignal`), vous accéderez à la valeur actuelle de `a + b` (c'est un fil, après tout), alors que si vous utilisez `myRegister`, vous accéderez à la valeur qu'avait `a + b` au cycle précédent !

Remarque Cette étape est particulièrement difficile, car il est nécessaire de prendre en main les fonctionnalités de `Chisel`, tout en découvrant sa syntaxe. N'hésitez pas à poser des questions, plutôt que de rester bloqué(e)s — notamment si vous avez des erreurs de compilation/d'exécution que vous avez du mal à interpréter.

Question 1.2.3 : Validation comportementale : version non pipelinée

Vous avez maintenant décrit votre premier générateur en `Chisel` (félicitations !). Comme son nom l'indique, il s'agit d'un générateur : il est difficile (voir impossible) de valider son comportement en simulation de façon formelle, puisque l'on peut générer une infinité de circuits différents avec ce même générateur.
Dans cette question, nous allons donc plutôt chercher à valider le comportement de certains des circuits générés, en s'appuyant sur les bancs de tests fournis.
Vous pouvez analyser le contenu du fichier `ParametrizedDotProductSpec.scala` (dans le répertoire `src/test/scala/dot`), qui contient une méthode assez générique pour tester les circuits générés. En particulier, on utilise ici une classe `DotProductConfig` (définie dans `src/main/scala/dot/Config.scala`) afin de stocker les différents arguments nécessaires pour instancier un circuit avec le générateur décrit en question précédente.
Cette fois-ci, on simule un circuit non combinatoire (puisque l'on a utilisé des registres dans le générateur). Pour ce faire, on utilise une bibliothèque de tests de scala — `chiseltest` — qui permet de créer des processus parallèles pour la simulation. En particulier, ici, on va utiliser un processus pour injecter des valeurs dans les entrées du circuit (avec la méthode `poke`), et un autre processus qui va attendre que la sortie soit valide, et qui va vérifier que la valeur est la même que celle calculée avec une implémentation logicielle. Vous n'avez pas besoin de comprendre les détails d'implémentation, mais vous pouvez lire le test et les commentaires afin de mieux comprendre comment fonctionne la simulation.
Une fois que vous avez compris comment fonctionne la simulation, vous pouvez vérifier le comportement des circuits générés. Lancez la commande suivante [12] :

```
sbt > testOnly dot.ParametrizedDotProductSpec -- -z "no pipeline"
```

Cette commande permet de valider le comportement d'un circuit non pipeliné (sans registre ajouté sur les chemins de calcul), généré avec les paramètres suivants (configuration `Config1` dans le fichier `Config.scala`) :

- `width = 8`
- `nElem = 4`
- `mulFunc(a, b) = a * b` (multiplication de base entre deux entiers non signés `a` et `b`)
- `addFunc(a, b) = a + b` (addition de base entre deux entiers non signés `a` et `b`)
- `latency = 0` (aucune latence n'est introduite par l'addition ou la multiplication)

12. La syntaxe pour spécifier quels tests lancer dans un fichier en particulier est un peu verbeuse. Ici, `-- -z "no pipeline"` permet de spécifier qu'on ne veut lancer que les tests donc la description contient la chaine "no pipeline", dans le test `ParametrizedDotProductSpec` du paquet `dot`.

Si tout se passe bien, les tests devraient passer. Sinon, vous pouvez utiliser l'outil `gtkwave` (voir Annexe B) pour analyser la valeur des différents signaux lors de la simulation, afin de débugger votre implémentation :

```
> cd test_run_dir/Parametrized_Dot_Product_with_no_pipeline
> gtkwave ParametrizedDotProduct.vcd
```

Remarque utiliser de `gtkwave` peut être compliqué. N'hésitez pas à poser des questions.

Question 1.2.4 : Synthèse logique : version non pipelinée

Une fois le comportement validé en simulation, on peut s'intéresser aux performances de notre circuit, notamment l'utilisation des ressources et la fréquence opérative. Pour ce faire, on va encore une fois faire appel à l'étape de synthèse logique.
Tout d'abord, générez une description `verilog` de votre accélérateur (toujours avec les paramètres d'instanciation définis dans la `Config1`) :

```
sbt > runMain dot.ParametrizedEmitterConfig1
```

Cela devrait générer un fichier `ParametrizedDotProduct.v` dans le répertoire `verilog/paramConfig1/`.
Pour lancer la synthèse, utilisez ensuite :

```
> make synthesis VERILOG=verilog/paramConfig1/ParametrizedDotProduct.v
```

Après quelques minutes, vous devriez avoir accès à un rapport de synthèse (plus pertinent que celui de l'exercice précédent, notamment au niveau de l'estimation de la fréquence d'opération). Que pouvez vous dire sur cette fréquence ? Sur l'utilisation des ressources disponibles ?
Un des intérêts principaux d'utiliser des générateurs de circuit est de pouvoir comparer différentes implémentations générées avec le même code. Conservez ce rapport quelque part (une simple capture d'écran suffira), et passez à la suite.

Question 1.2.5 : Validation comportementale : version pipelinée

Une fois cette première synthèse réalisée, on va s'intéresser à optimiser notre accélérateur, afin d'obtenir de meilleurs performances. Pour ce faire, si on avait décrit un accélérateur non paramétré (comme dans l'exercice précédent), il faudrait modifier le code afin d'ajouter les optimisations souhaitées. Ici, on va utiliser la notion de générateur pour intégrer des optimisations directement dans les paramètres d'instanciation de notre générateur, plutôt que dans son code. En l'occurence, dans cette question, on va s'intéresser à une nouvelle configuration (`Config2`, toujours dans le fichier `src/main/scala/dot/Config.scala`) :

- `width = 8`
- `nElem = 4`
- `mulFunc(a, b) = RegNext(a * b)`
- `addFunc(a, b) = RegNext(a + b)`
- `latency = 3` (on ajoute un cycle pour la multiplication, puis 2 cycles pour les deux additions successives — réduction binaire en $log_2(nElem)$)

On remarque que les fonctions `mulFunc` et `addFunc` ont été modifiée, et un registre a été ajouté en sortie des opérations, afin de casser le chemin critique. Cela va permettre de générer un circuit pouvant fonctionner à une plus haut fréquence (fréquence $= \frac{1}{\text{chemin critique}}$), et ayant donc potentiellement de meilleurs performances — mais cela demande également des ressources supplémentaires (les registres ajoutés), et introduit une latence dans le fonctionnement (3 cycles).
Validez le comportement du circuit généré par votre générateur avec cette configuration, grâce à la commande suivante :

```
sbt > testOnly dot.ParametrizedDotProductSpec -- -z "with pipeline"
```

Si votre implémentation est encore correcte (notamment, si vous avez bien définis le signal `io.outValid`), le test devrait encore passer. Sinon, il ne vous reste plus qu'à débugger, à l'aide de `gtkwave` :

```
> cd test_run_dir/Parametrized_Dot_Product_with_pipeline
> gtkwave ParametrizedDotProduct.vcd
```

Question 1.2.6 : Synthèse logique : version pipelinée

On va maintenant comparer les performances des deux versions que nous avons générées, après synthèse logique.
Générez la description `verilog` correspondant à cette nouvelle configuration `Config2` à l'aide de la commande suivante :

```
sbt > runMain dot.ParametrizedEmitterConfig2
```

Lancez la synthèse logique sur le serveur avec :

```
> make synthesis VERILOG=verilog/paramConfig2/ParametrizedDotProduct.v
```

Que remarquez vous sur les résultats de synthèse ? Comparez les résultats obtenus avec ceux de l'exercice précédent. Quelle version vous semble la meilleure ?

Remarque sur les paramètres Vous remarquerez que dans les deux configurations proposées (`Config1` et `Config2`), le nombre d'élément dans les vecteurs en entrée est une puissance de 2. C'est très pratique, car la fonction `reduceTree` a besoin d'un arbre équilibré pour être utilisée comme on l'a fait. Si ce n'est pas le cas, on peut rencontrer un problème de synchronisation. Par exemple, si on a un vecteur comprenant 3 éléments à un cycle t $v_t = \{v_{(t,0)}, v_{(t,1)}, v_{(t,2)}\}$, on va additionner les deux premiers éléments $v_{(t,0)} + v_{(t,1)}$, et le dernier ensuite. Cela pose problème si l'addition se fait en plusieurs cycles de façon pipelinée : les éléments ne sont plus synchronisés, et le résultat de l'addition des deux premiers éléments $v_{(t,0)} + v_{(t,1)}$ sera ensuite additionnée au troisième élément d'un autre vecteur $v_{(t+1,2)}$, car un cycle de latence aura été introduit.
En réalité, une utilisation plus "propre" aurait été de fournir deux fonctions à `reduceTree` : une fonction pour réaliser la réduction binaire (ici, l'addition de deux éléments), et une fonction à réaliser dans le cas où il n'y pas d'autre élément avec lequel effectuer la réduction.

Par défaut, cette fonction est l'identité, mais il est tout a fait possible d'utiliser un registre (voir un registre à décalage) pour resynchroniser les éléments en cas d'arbre non équilibré.

Question 1.2.7 : Synthèse logique sur des instances réalistes

Les deux premières configurations permettent de réaliser des produits scalaires sur des vecteurs de 4 éléments sur 8 bits chacun. Il est peu probable, dans la vraie vie, qu'on ait besoin d'une implémentation matérielle pour ce genre de calcul.
Pour vous donner une idée de ce à quoi peuvent ressemble des implémentations plus réalistes, je propose d'analyser des résultats de synthèse (réalisées de mon côté, pour des soucis de temps et de ressources de calcul), sur trois autres configurations.
Ces résultats sont détaillés dans la Table 1. La Table 2 détaille quand à elle les paramètres des différentes configurations (définies dans le fichier `src/main/scala/dot/Config.scala`).

Configuration	**Fréquence**	**LUTs**	**Registres**	**BRAMs**	**DSPs**
`Config3`	171MHz	1712 (3.22%)	1570 (1.48%)	0	64 (29.09%)
`Config3DSP`	250MHz	960 (1.80%)	1842 (1.73%)	0	64 (29.09%)
`Config4`	81MHz	113310 (212.99%)	23554 (22.14%)	0	220 (100.00%)

Table 1 – Résultats des différentes synthèses logiques

Configuration	`width`	`nElem`	`mulFunc`	`addFunc`	`latency`
`Config1`	8	4	`a * b`	`a + b`	0
`Config2`	8	4	`RegNext(a * b)`	`RegNext(a + b)`	3
`Config3`	32	16	`RegNext(a * b)`	`RegNext(a + b)`	5
`Config3DSP`	32	16	`ShiftRegister(a * b, 3)`	`RegNext(a + b)`	7
`Config4`	32	128	`RegNext(a * b)`	`RegNext(a + b)`	8

Table 2 – Détails des configurations utilisées en Table 1

Utilisation des DSPs On remarque que, contrairement à vos implémentations, les implémentations plus réalistes utilisent les DSPs disponibles sur la carte cible (pour l'implémentation des multiplications). C'est lié à une heuristique de l'outil de synthèse : pour les multiplications sur moins de 16 bits, il utilisera des LUTs, pour des multiplications plus large, des DSPs. Cela a un impact non négligeable sur la fréquence.
On remarque également que la seule différence entre les configurations `Config3` et `Config3DSP` est l'ajout de registres supplémentaires sur le résultat de la multiplication. Ce n'est encore une fois pas un hasard : la spécification de la carte cible précise qu'il est de bon ton d'ajouter 3 registres en sortie des multiplications. L'outil de synthèse peut ensuite "absorber" ces registres dans les blocs DSPs, afin d'implémenter des multiplications pipelinées plus performantes.

Et non, ce n'est pas de la magie, même si parfois ça y ressemble.

Questions d'ouverture Analysez les différentes implémentations proposées. Quel est l'impact d'ajouter des registres sur les chemins critiques ? Que remarquez vous quand les blocs DSPs sont saturés (`Config4`) ?

Exercice 1.3 Produit scalaire avec type générique

On s'intéresse maintenant à implémenter un générateur encore plus générique, avec la possibilité de définir le type de données manipulées à la génération.
Pour ce faire, on vous fourni encore une fois un squelette de module pour l'implémentation, dans le fichier `src/main/scala/dot/GenericDotProduct.scala` (et, évidemment, le fichier de test associé dans `src/test/scala/dot/GenericDotProductSpec.scala`).

Question 1.3.1 : Types de données génériques

Regardez le squelette fourni, et comparez le au module développé dans l'exercice précédent. Quelles sont les principales différences ? Vous parait-il compliqué de proposer des générateurs avec un type paramétrique, plutôt que fixe ?

Point de syntaxe Le constructeur `GenericDotProduct` utilise maintenant un **paramètre de type** : `[T <: Data with Num[T]]`. Cette syntaxe (un peu déconcertante, il est vrai), signifie que le module va utiliser un type `T` générique (à la façon des *templates* `C++`, par exemple). De plus, ce type `T` est contraint de par sa déclaration. Tout d'abord, il doit forcément hériter du type `Data` (le type de donnée de base de `Chisel`). En outre, il doit également implémenter une interface (ou `trait`, en `scala`) particulière : l'interface `Num`, qui permet de garantir que les opérations numériques de base (par exemple, addition et multiplication) sont bien définies sur `T`. C'est ce qui nous permet d'utiliser les opérations de base de `Chisel` dans le code, sans avoir à se soucier du type sous-jacent.

Question 1.3.2 : Implémentation et validation comportementale

Vous devriez pouvoir implémenter ce nouveau module sans trop de soucis (en réalité, en recopiant ce que vous avez fait à l'exercice précédent).
Si tout se passe bien, vous pourrez valider le comportement à l'aide du banc de test fourni, que ce soit sur des **entiers non signés** ou sur des nombres en **virgule fixe** (*fixed point*) :

```
sbt > testOnly dot.GenericDotProductSpec
```

A priori, si votre code était fonctionnel à l'exercice précédent, les tests devraient passer sans soucis

Question 1.3.3 : Comparaison des performances

Si vous le souhaitez, vous pouvez maintenant réaliser des synthèses des circuits générés avec ce nouveau générateur. Vous pouvez générer deux versions (signée et *fixed point*) basée sur la configuration `Config3` définie à l'exercice précédent :

```
sbt > runMain dot.GenericEmitterConfig3
sbt > runMain dot.GenericEmitterConfig3FP
```

Il est intéressant de regarder le code `verilog` généré, notamment pour l'implémentation avec *fixed point* : analysez le code produit dans `verilog/paramConfig3FP/GenericDotProduct.v`. Comment sont gérées les implémentations en virgule flottante ?
Pour vous éviter d'avoir à relancer des synthèses, une comparaison des résultats de synthèse pour ces deux configurations est présentée en Table 3).

Configuration	Fréquence	LUTs	Registres	BRAMs	DSPs
`Config3`	171MHz	720 (1.35%)	786 (0.74%)	0	48 (21.82%)
`Config3FP`	171MHz	1216 (2.29%)	1282 (1.20%)	0	64 (29.09%)

Table 3 – Résultats des différentes synthèses logiques pour la version générique

Consommation des ressources Vous remarquerez que les ressources pour la configuration `Config3` sont moindres que celles utilisées à l'exercice précédent. C'est du à un détail d'implémentation : dans l'exercice précédent, la taille du signal `io.out` était laissé non définie, ce qui amenait `Chisel` à l'inférer (résultant en un calcul sur 64 bits pour cette configuration). Dans cette nouvelle version, on a fixé le type de `io.out` à `UInt(32.W)`, ce qui permet à `Chisel` d'éliminer une partie de la logique, qui correspond à du code mort (nécessaire pour calculer les bits 33 à 64 du résultat, désormais inutiles. Vous pouvez retrouver cette différence dans les fichiers `verilog` générés (`verilog/paramConfig3`) : le port `io_out` n'a pas la même taille entre le fichier `ParametrizedDotProduct.v` et le fichier `GenericDotProduct.v`.

Comparaison entre virgules fixes et entiers non signés Comparez les résultats de synthèse présentés dans la Table 3. Que remarquez vous sur la fréquence ? Sur la consommation de ressource ?

2 Pour aller plus loin

Le but de ce TP était de vous présenter très rapidement des usages avancés (notamment, la programmation fonctionnelle pour la génération) de `Chisel`, tout en prenant en main la syntaxe, et l'écosystème. On aurait pu, avec plus de temps, implémenter une unité de calcul matriciel plus poussée — même si l'une des limites principales sur ce genre d'architecture est en fait de gérer les protocoles de communication, et non pas d'implémenter les calculs. Si l'utilisation de ce langage vous a plu, et que vous souhaitez en apprendre un peu plus, je vous encourage à jeter un oeil aux ressources complémentaires (Section 3), et à me contacter au besoin.

3 Ressources complémentaires

Voici différentes ressources complémentaires sur `Chisel`, qui ont été utilisées dans la conception de ce TP, ainsi que celle du cours associé :

- un *bootcamp* permettant d'acquérir les bases de `scala` et de `Chisel` nécessaires — notamment basé sur un **jupyter notebook** permettant de tout faire en ligne !
- le *template* de base `Chisel`, pour démarrer un nouveau projet.
- le site web de la communauté `Chisel`, et la *mailing list* associée.
- l'API `Chisel` en ligne
- un ouvrage dédié à l'apprentissage de la conception numérique avec `Chisel`.
  Martin Schoeberl. *Digital design with chisel*. Kindle Direct Publishing, 2019
- le manuscrit de thèse de Jean Bruant (OVHcloud & TIMA).
  Jean Bruant. "Abstracting Hardware Architectures for Agile Design of High-performance Applications on FPGA". Thèse de doct. Université Grenoble Alpes, 2023
- mon manuscrit de thèse — notamment la première annexe.
  Bruno Ferres. "Leveraging Hardware Construction Languages for Flexible Design Space Exploration on FPGA". Thèse de doct. Université Grenoble Alpes, 2022

Contact

Vous pouvez me contacter par mail à l'adresse bruno.ferres@inria.fr.

A Utilisation de vivado pour la synthèse logique

Afin de vous éviter de devoir installer vivado sur votre machine (≃ 70Go...), nous vous proposons de réaliser les synthèses logiques (nécessaires notamment pour analyser la qualité de vos circuits) sur les serveurs. Pour ce faire, vous pouvez directement utiliser le Makefile fourni à la racine du TP, afin de réaliser vos synthèses [13].
Pour lancer une synthèse sur le serveur, vous devez d'abord modifier les premières lignes du Makefile pour renseigner les informations de connexions qui vous ont été fournies pour les serveurs. Vous devez donc renseigner votre nom d'utilisateur, le nom du serveur et le nom de votre répertoire personnel, respectivement dans les variables USER, SERVER et USER_PATH. Vous pouvez également mettre en place l'authentification par clef RSA, pour être plus efficace (voir Section A.1).
Une fois fait, vous pouvez utiliser le Makefile pour synthétiser une description verilog **générée précédemment**. Pour ce faire, vous pouvez lancer simplement la commande :

```
make synthesis VERILOG=<pathToFile>
```

Les rapports de synthèse sont générés dans $HOME/$USER_PATH/chisel/results/<top>_<date>.
Chaque répertoire généré est donc suffixé avec l'heure de sa création (au format h_min_sec), afin de garder une trace de vos différentes synthèse.
Le processus de synthèse peut prendre du temps (≃ 5 minutes maximum, dans le cadre de ce TP). Si votre synthèse dépasse ce délai, n'hésitez pas à m'appeler. En outre, si vous le souhaitez, vous pouvez activer l'affiche des logs de vivado en ajoutant V=1 lorsque vous lancez la commande make.

A.1 Authentification SSH par clef RSA

Pour éviter d'avoir à renseigner le mot de passe à chaque connexion (plusieurs fois par synthèse, en réalité), je vous encourage fortement à mettre en place l'authentication par clef RSA [14]. Pour ce faire :

1. générez une clef RSA dans le fichier .ssh/id_rsa.pub (et .ssh/id_rsa) avec ssh-keygen
2. ajoutez le contenu du fichier id_rsa.pub sur le serveur distant, dans le fichier .ssh/authorized_keys

13. Pour information, la cible technologique est une FPGA de la gamme Virtex 7 de chez **Xilinx**, plus précisément la FPGA que l'on retrouve dans des SoCs comme les cartes Zybo ou ZedBoard.
14. Pour plus d'informations, vous pouvez vous renseigner en ligne ici, ou m'appeler directement.

B Utilisation de `gtkwave` pour la simulation

Le logiciel `gtkwave` peut être utilisé pour visualiser le résultat de simulations au niveau RTL. Cette annexe présente succintement l'utilisation de ce logiciel.

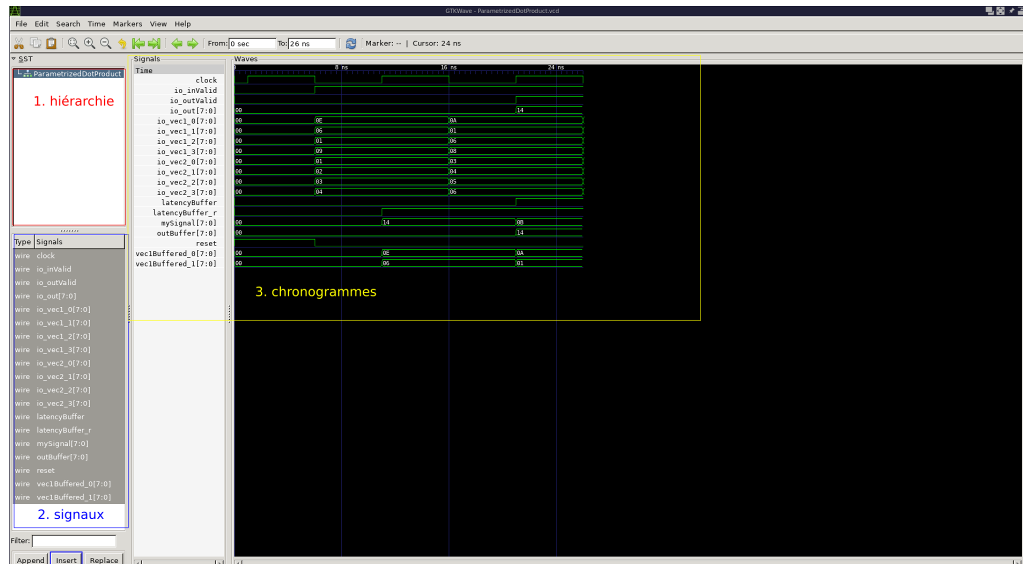

FIGURE 2 – Exemple d'utilisation de `gtkwave`

Afin d'utiliser `gtkwave`, vous devez en premier lieu générer un fichier `.vcd`, contentant les résultats d'une simulation RTL (durant ce TP, les appels à `sbt testOnly` devraient vous permettre de générer ces fichiers). Ensuite, vous il faut ouvrir ces fichiers avec `gtkwave` :

```
> gtkwave <pathToFile>/<top.vcd>
```

Une fois fait, il faut sélectionner les signaux à tracer, via trois étapes (Figure 2) :

1. sélectionner le module dans lequel les signaux sont définis (étape 1, dans le cadre rouge en haut à gauche)
2. une fois un module sélectionné, le cadre bleu (en bas à droite) contient tous les signaux définis dans ce module. Il faut donc sélectionner les signaux que l'on souhaite tracer (par exemple, en utilisant *Ctrl + Click* ou *Shift + Click*), et appuyer sur le bouton *Insert* pour les ajouter aux signaux tracés.
3. enfin, dans le cadre jaune, on peut naviguer dans le temps de la simulation, zoomer, etc, pour analyser le comportement de notre circuit.

Représentation des signaux Par défaut, les signaux affichés dans `gtkwave` sont affichés au format **hexadécimal**. Si vous souhaitez utiliser un autre format (par exemple, décimal, si on fait des calculs sur des entiers), vous pouvez faire un clic droit sur le nom des signaux dans l'encadré jaune, et changer le format.