EFREI Paris

Concéption de circuit numérique 2

Rapport Projet

Microcontrôleur en VHDL

Jacques Soghomonyan

Table des matières

> Les images de simulation sont parfois assez petite et un zoom peut être nécessaire pour les comprendre.

1 — Introduction

Ce projet a pour but de nous introduire au monde des cartes programmables FPGA.
Pour ce faire nous avons programmé, en VHDL, une architecture simple de microcontrôleur. Ici, nous allons détailler les aspects techniques de toutes les composantes.

L'implémentation des composants est traité dans ce rapport mais, le code étant assez explicite, nous ne nous attarderons pas plus que ça sur le code.

J'ai décidé de donner des noms très verboses aux signaux pour mieux comprendre et identifier les problèmes. C'est pourquoi ils diffèrent de ceux donné dans le sujet.

2 — Plan d'attaque.

2.1 - Avant-gout.

Nous devons diviser le problème pour qu'il convienne au modèle entité de VHDL.
L'énoncé est exhaustif quant à la manière de faire et les éléments à prendre en compte pour construire le système.

Dans le sujet, un schéma nous est donné. J'ai décidé d'en diverger un peu[a]. En effet, certaines connexions semblaient redondantes p. ex. connexion de l'horloge à l'interconnexion. De plus, d'autres étaient implicites p. ex. les *enable* des *buffer*.

[a] Le schéma simplifié, montrant les nouvelles connexions, peut être trouvé dans le fichier **project-schema.pdf**.
Le schéma n'est pas légendé, mais les entrées, sorties, dimensions des ports, nom des ports sont identifiables. Même s'il n'est pas à jour, il permet de visualiser l'agencement du projet.
Il est fait pour être lu en tandem avec le sujet original.

2.2 - Outils auxiliaires.

Pour ce projet, j'ai décidé d'utiliser des outils auxiliaires pour faciliter le flux de travail.

GNU Make Un outil d'automatisation de construction de projet. Toutes les commandes d'élaboration et des tests unitaires sont déléguées à ce système.

cocotb Un outil aidant l'écriture de tests unitaires[a] avec des scripts python.

[a] J'ai originalement rédigé des tests en vhdl, mais j'ai ultimement décidé de les refaire en python. Les originales sont encore là.

2.3 - Les composants.

Voici la liste des composants, traduis par des entités en VHDL, de notre système.

- Le buffer
- L'unité arithmétique logique
- Le bloque d'instruction
- L'interconnexion

Nous élaborerons sur l'implémentation technique dans la partie qui suit.

3 — L'implémentation.

3.1 - Arborescence du projet.

Les composants sont nommés en anglais, parfois mal, comme avec le **dbus**, représentant l'interconnexion. Mais par soucis de complexité de tout changer, j'ai laissé les noms donnés au début du projet.
Tous les composants nommés au-dessus sont stockés dans des dossiers éponymes[a].

Dans les dossiers, on peut retrouver, en général :

- Un fichier VHDL principal contenant l'entité en question et son architecture.
- Un fichier VHDL générant le signal visualisable.
- Un fichier VHDL contenant les tests unitaires pour l'entité.
- Un fichier python contenant également des tests unitaires pour l'entité.
- Un dossier contenant les signaux visualisables, nommé **waves/**.
- Et enfin un *makefile*, utilisé pour lancer les tests et générer les signaux pour l'entité.

Dans la racine du répertoire, un *makefile* lance tous les tests de tous les composants du système et synthétise le composant nommé **microcontroler**, la synthèse de toutes les entités.

[a] en anglais

3.2 - nbuffer

Le buffer est l'élément le plus utilisé du projet.
En effet, c'est l'entité derrière :

- out_sel_buf
- fn_sel_buf
- carries_buf
- a_buf
- b_buf
- cache1_buf
- cache2_buf

C'est un buffer assez simple. Le composant renvoie l'entrée sur front montant d'horloge.
Il possède aussi deux signaux de contrôle.
Le signal **enable**, qui permet une mémoire, la sortie ne change que si le signal est actif.
Le signal **reset**, qui remet la sortie à zéro asynchroniquement.

> Les signaux originalement nommés **SR_IN_L** et **SR_IN_R**, sont devenus un seul buffer nommé **carries_buf**.
> De plus, les signaux nommés **SR_OUT_L** et **SR_OUT_R**, sont juste un vecteur de taille 2 en sortie du microcontrôleur.

3.3 - alu

L'alu agit asynchroniquement. Il est responsable de la partie logique, il est contrôlé par **fn_sel_buf**.
Étant donné que l'on jongle avec des entrées à 4bit et une sortie à 8bit, on a défini des variables représentant les entrées sur 8bit. Cela est fait principalement pour conserver la cohérence du signe dans les opérations arithmétiques.

3.4 - dbus

Le dbus représente l'interconnexion. Il agit un peu comme un bus de données, il lie tout le monde entre eux et permet aux données d'être transmissent où on veut[a].
Il est dirigé par **out_sel_buf** et **route_selection**[b].
Il agit asynchroniquement.

[a] J'essaye de justifier mes mauvais choix de nommage, je sais.
[b] Une sortie du bloc d'instruction.

3.5 - instructions

Le bloc d'instruction est un bloc synchrone qui émet ses signaux de sorties sur front descendant d'horloge.

Il stocke les trois programmes demandés dans le sujet en mémoire. Les programmes sont des matrices de taille 128 de *std_logic_vector*.

Le choix du programme est fait par un signal en entrée.
On a créé un signal interne qui définit le pointeur vers l'instruction actuelle. Il est mit à zéro quand le programme change. Et incrémenté de un chaque front montant.

3.6 - microcontroler

Ce composant est la synthèse de tous les autres composants avec des *port map*.

4 — Les résultats

Ici, nous allons visualiser et analyser les résultats.

Pour lancer les tests, il suffit d'être dans le répertoire racine du projet et lancer **Make** avec la commande `make`.
Par defaut les tests seront ceux que j'ai écrit en vhdl. Pour lancer les tests python, il faut installer la librairie **cocotb**[a] et lancer la commande `VHDL_PYTEST=1 make`.

[a] `pip install cocotb`

4.1 - alu

Pour l'ALU j'ai dû être exhaustif quant au couvrage des tests.
C'est ici que la polyvalence de python brille. En effet, rédiger les tests unitaires est bien plus simple en python.

```python
@cocotb.test()
async def a_left_shift_with_carry(dut):
    dut.function_selection.value = LogicArray("0010")
    dut.carries_received.value = LogicArray("11")

    for test_value in range(2**4):
        dut.a.value = LogicArray(test_value, Range(3, 'downto', 0))
        await wait(dut) # 1ns
        expected_value = ((test_value << 1) & 0b00001111) | 0b0001
        expected_carry = (test_value & 0b1000) >> 2

        assert LogicArray(dut.alu_output.value) == LogicArray(expected_value, Range(7,'downto',0))\
        and LogicArray(dut.carries_emitted.value)==LogicArray(expected_carry, Range(1,'downto',0))
```

Listing 1: Partie du fichier *alu/alu_test.py* concernant le décalage gauche avec retenu de a.

On voit qu'on a une boucle dont l'indice va prendre toutes[a] les valeurs possibles de a, puis on assigne à l'entrée **a** du composant la valeur de l'indice.
On va ensuite, faire la transformation que l'on attent sur la valeur de l'indice. Ici, un décalage à gauche (`(test_value << 1)`), puis on force la sortie sur 4 bit (`& 0b00001111)`) et on ajoute la retenue attendue (`| 0b0001`).
Ensuite on fait la même chose pour la retenue attendue. Ici, on isole le bit sortant (`test_value & 0b1000`), puis on le décale pour qu'il corresponde avec le modèle de la sortie de retenu(un *std_logic_vector(1 downto 0)*).
Finalement, on fait un test d'assertion pour savoir si les sorties de l'**ALU** correspond avec nos valeurs attendues.

```
**************************************************************************************
** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
**************************************************************************************
** alu_test.nop                   PASS           1.00           0.00       1086.61  **
** alu_test.a_right_shift_no_carry PASS          16.00           0.00      17216.23  **
** alu_test.a_right_shift_with_carry PASS        16.00           0.00      20997.77  **
** alu_test.a_left_shift_no_carry PASS           16.00           0.00      13819.78  **
** alu_test.a_left_shift_with_carry PASS         16.00           0.00      16677.15  **
** alu_test.b_right_shift_no_carry PASS          16.00           0.00      11366.68  **
** alu_test.b_right_shift_with_carry PASS        16.00           0.00       8715.44  **
** alu_test.b_left_shift_no_carry PASS           16.00           0.00      11224.10  **
** alu_test.b_left_shift_with_carry PASS         16.00           0.00      12452.94  **
** alu_test.a_identity            PASS           16.00           0.00      10059.79  **
** alu_test.b_identity            PASS           16.00           0.00      12811.93  **
** alu_test.not_a                 PASS           16.00           0.00      19367.64  **
** alu_test.not_b                 PASS           16.00           0.00      20062.44  **
** alu_test.a_and_b               PASS          256.00           0.02      14513.16  **
** alu_test.a_or_b                PASS          256.00           0.02      15941.06  **
** alu_test.a_xor_b               PASS          256.00           0.02      16613.68  **
** alu_test.a_plus_b_with_carry   PASS          256.00           0.02      13556.32  **
** alu_test.a_plus_b_no_carry     PASS          256.00           0.02      14820.66  **
** alu_test.a_minus_b             PASS          256.00           0.02      14339.69  **
** alu_test.a_times_b             PASS          256.00           0.02      11367.40  **
**************************************************************************************
** TESTS=20 PASS=20 FAIL=0 SKIP=0               1985.00           0.75       2659.24  **
**************************************************************************************
```

Figure 1: Confirmation du logiciel que tous les tests sont passés.

[a] 4bit $\Longrightarrow$ 2^4 valeurs ($[0; 2^4[$)

4.2 - dbus

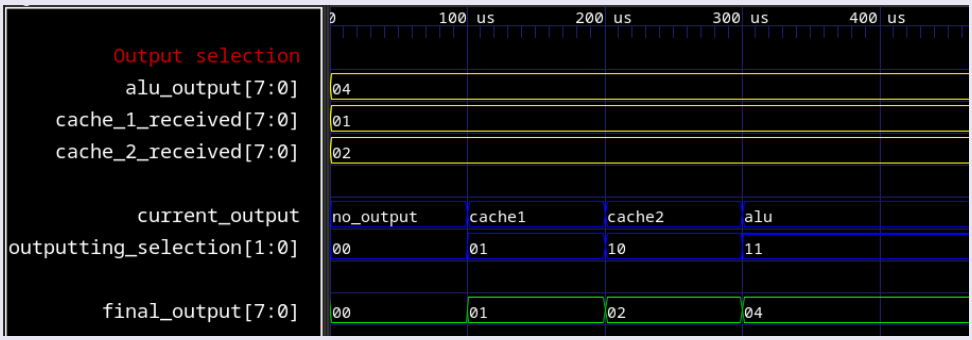

Figure 2: Résultats visuels du dbus pour la partie de sélection de sortie.

On a défini un signal auxiliaire pour une meilleure lisibilité. Il indique de quel composant la sortie va hériter. On voit que la sortie de l'interconnexion est bien contrôlée asynchroniquement par le signal **outputting_selection**.

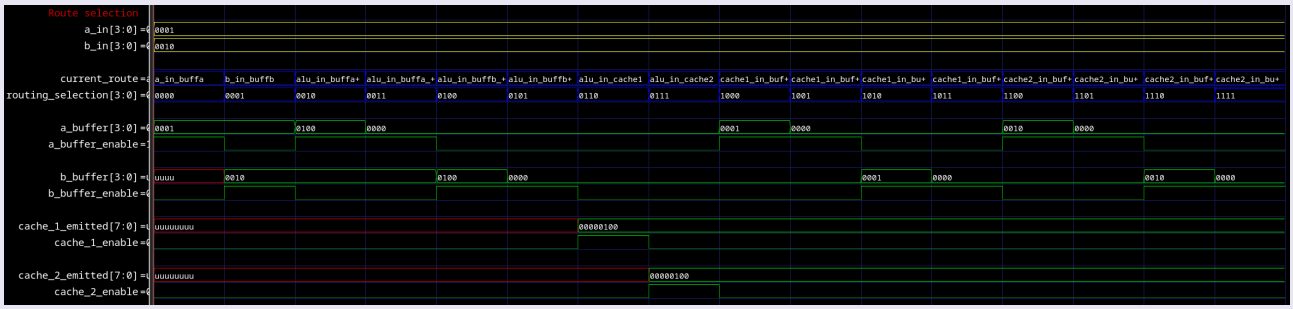

Figure 3: Résultats visuels du dbus pour la partie de sélection de route des données.

On voit (assez peu précisément) que l'interconnexion contrôle bien la route des données. Par exemple les enable sont bien mise à un quand on a envie de modifier le buffer en question.
Pour être plus exhaustif, je me remets aux tests unitaires. Ceux-ci sont rédigés comme suit.
On a fixé les valeurs des entrées (**a_in**, **b_in**, **cache_1_received**, **cache_2_received**, **alu_output**).

Puis on observe les sorties (**b_buffer**, **a_buffer_enable**, **b_buffer_enable**, **cache_1_enable**, **cache_2_enable**) pour détecter si le comportement est celui que l'on attend.

```
**********************************************************************************************
** TEST                                  STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
**********************************************************************************************
** dbus_test.a_in_buf_a                  PASS          1.00          0.00       1428.58  **
** dbus_test.b_in_buf_b                  PASS          1.00          0.00       4447.83  **
** dbus_test.alu_lsb_in_buf_a            PASS          1.00          0.00       6864.66  **
** dbus_test.alu_msb_in_buf_a            PASS          1.00          0.00       7825.20  **
** dbus_test.alu_lsb_in_buf_b            PASS          1.00          0.00       6842.26  **
** dbus_test.alu_msb_in_buf_b            PASS          1.00          0.00       7061.12  **
** dbus_test.alu_in_cache_1              PASS          1.00          0.00       6017.66  **
** dbus_test.alu_in_cache_2              PASS          1.00          0.00       6177.18  **
** dbus_test.cache_1_lsb_in_buf_a        PASS          1.00          0.00       7157.52  **
** dbus_test.cache_1_msb_in_buf_a        PASS          1.00          0.00       4529.49  **
** dbus_test.cache_1_lsb_in_buf_b        PASS          1.00          0.00       5405.04  **
** dbus_test.cache_1_msb_in_buf_b        PASS          1.00          0.00       5440.09  **
** dbus_test.cache_2_lsb_in_buf_a        PASS          1.00          0.00       7219.12  **
** dbus_test.cache_2_msb_in_buf_a        PASS          1.00          0.00       7002.18  **
** dbus_test.cache_2_lsb_in_buf_b        PASS          1.00          0.00       6944.22  **
** dbus_test.cache_2_msb_in_buf_b        PASS          1.00          0.00       7206.72  **
** dbus_test.final_output_is_none        PASS          1.00          0.00       6423.14  **
** dbus_test.final_output_is_cache_1     PASS          1.00          0.00       7810.63  **
** dbus_test.final_output_is_cache_2     PASS          1.00          0.00       8081.52  **
** dbus_test.final_output_is_alu_output  PASS          1.00          0.00       8338.58  **
**********************************************************************************************
** TESTS=20 PASS=20 FAIL=0 SKIP=0                    20.00          0.65         30.92  **
**********************************************************************************************
```

Figure 4: Confirmation du logiciel que tous les tests sont passés.

4.3 - instructions

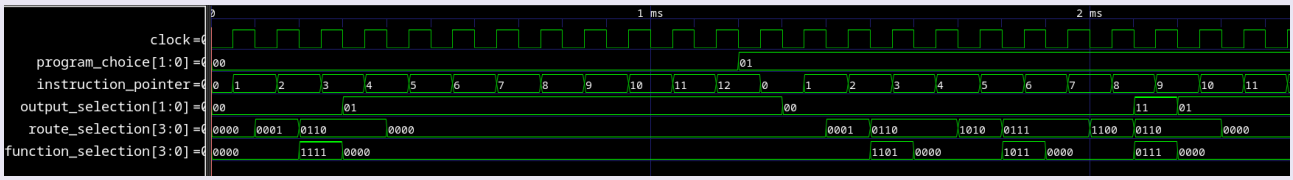

Figure 5: Sortie visuelle du bloc d'instruction des programmes res_out_1 et res_out_1 à 1200μs.

```
constant a_times_b : t_matrix := (
    ------------------------------------a * b-------------------------------------------
    to_instr("0000", "0000", "00"), -- nop, a -> buf a, nop
    to_instr("0000", "0001", "00"), -- nop, b -> buf b, nop
    to_instr("1111", "0110", "00"), -- a*b, s -> cache1, nop
    to_instr("0000", "0110", "01"), -- nop, s -> cache1, resout -> cache1,
    ------------------------------------------------------------------------------------
    others => "0000000001"
);
```

Listing 2: Instructions pour res_out_1, pour comparer.

On voit bien que le bloc d'instruction envoie bien les commandes sur front descendant d'horloge. De plus, le pointeur d'instruction est bien remise à zéro quand on change de programme.

4.4 - nbuffer

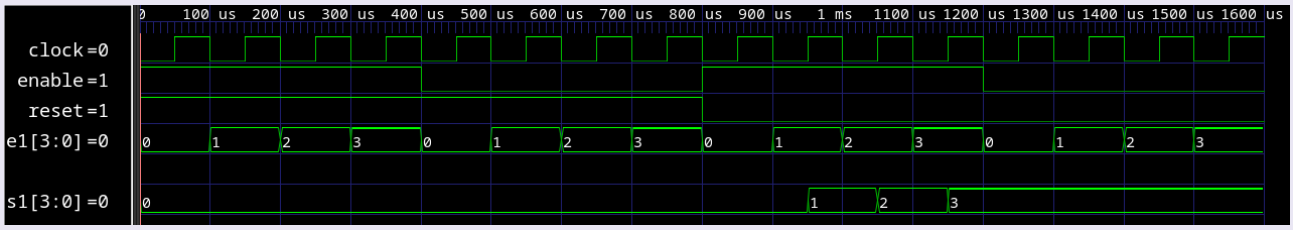

Figure 6: Résultats visuels du buffer sur 2bit.

On voit que tout fonctionne comme entendu.
Le signal **reset** force l'entrée à zéro.
La sortie est bien sur front montant.
Le signal **enable** permet la garde en mémoire, l'entrée précédente.

4.5 - LE RÉSULTAT FINAL

On arrive enfin à la pièce maitresse du projet, le microcontrôleur.
Dans le sujet, il nous est demandé de réaliser trois programmes pour le microcontrôleur. Comme indiqué précédemment, ils sont stockés dans le bloc d'instructions.
J'ai suivi le même processus que pour l'ALU, et j'ai rédigé trois tests[a] exhaustifs avec un couvrage maximal.

```
dut.program_choice.value = LogicArray("00")
for a in range(-8, 8):
    for b in range(-8, 8):
        reset(dut)
        dut.a_in.value = LogicArray(a, Range(3, 'downto', 0))
        dut.b_in.value = LogicArray(b, Range(3, 'downto', 0))

        await n_cycles(dut, 10)

        assert LogicArray(dut.final_output.value) == LogicArray(a * b, Range(7,'downto',0))

        dut.program_choice.value = LogicArray("01")
        await n_cycles(dut, 1)
        dut.program_choice.value = LogicArray("00")
```

Listing 3: Test pour le programme res_out_1.

Les trois ont la même forme.
On va d'abord choisir le programme.
Puis, on définit deux boucles qui vont être les valeurs des entrées du microcontrôleur.
Ensuite, on les assigne et on attend le temps de calcul du résultat.
Par la suite, nous appliquons le test d'assertion avec la valeur attendue calculée avec python.
Enfin, on change de programme pour remettre à zéro le pointeur d'instruction.
On remarque que les bornes sont signées, car le programme res_out_1 consiste en la multiplication, une instruction arithmétique.

Pour le test du deuxième programme, on remarque que les bornes sont maintenant non signées, car le programme res_out_2 consiste en des transformations logiques.

Nous avons la même chose pour le troisième programme, à l'exception que celui si à des bornes allant de 0 à 3 inclus pour les entrées. En effet, le programme ne concerne que les 2 premiers bit de chaque entrée.

```
**************************************************************************************
** TEST                            STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
**************************************************************************************
** microcontroler_test.program_1   PASS       56320.00           0.19     301971.44  **
** microcontroler_test.program_2   PASS       81920.00           0.30     270201.05  **
** microcontroler_test.program_3   PASS       19520.00           0.09     222645.64  **
**************************************************************************************
** TESTS=3 PASS=3 FAIL=0 SKIP=0              157760.00           1.12     141038.65  **
**************************************************************************************
```

Figure 7: Confirmation du logiciel que tous les tests du microcontrôleur sont passés.

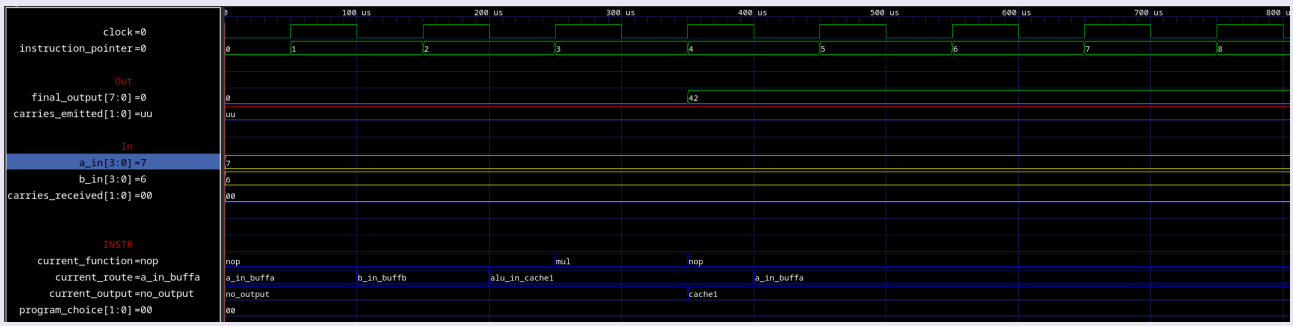

Figure 8: Rendu visuel montrant le programme 1 avec 6 et 7 en entrées.
On peut voir que le résultat reste sur la sortie jusqu'au bout.

[a] Les plus durs à mettre en place.

5 — Conclusion

Je peux avec confiance, affirmé que le projet fonctionne.
De plus, nous avons eu l'opportunité d'expérimenter avec une carte FPGA[a]. J'y ai testé l'ALU du projet, qui fonctionnait.

Tous les tests sont reproduisibles. Il suffit, comme indiqué en introduction, d'utiliser *make*.
Je conseillerais d'utiliser les tests pythons qui sont beaucoup plus exhaustifs et ayant un couvrage de toutes les possibilités.
Vous trouverez tous les résultats de rendu et des tests dans les dossiers respectif de tous les composants. Les fichiers d'extension `.gtkw` sont déjà formatés par simplicité, il suffit de les ouvrir pour voir les signaux déjà organisés pour la visualisation.

J'ai pu explorer en approfondissant mes nouvelles connaissances en VHDL et en programmation de FPGA.

[a] Xilinx Arty 35T