ASR1 – TD8 : Un vrai RISC dans mon FPGA !

{ Jeremie.Detrey, Patrick.Loiseau, Nicolas.Veyrat-Charvillon } @ens-lyon.fr
http://perso.ens-lyon.fr/jeremie.detrey/06_asr1/

11 et 18 décembre 2006, et 8 janvier 2007

Le but de ce triple TP est de mettre en application un bon paquet des choses vues en cours et en TD, et pour cela, rien de tel que de mettre les mains dans le cambouis !

Nous allons donc réaliser ici un vrai processeur RISC, plus précisément celui que nous sommes parvenus (non sans mal !) à spécifier lors des derniers TD. Pour ce faire, nous allons le décrire intégralement en VHDL, puis nous l'implémenterons sur un FPGA (*Field-Programmable Gate Array*), un circuit intégré complètement programmable capable d'implémenter n'importe quel circuit.

Mais avant toute chose, pas mal de choses préalables à ingurgiter pour vous. N'hésitez pas à appeler vos TD-men à la rescousse si les explications suivantes ne vous sont pas limpides.

1 Rapide mise à niveau en VHDL

Mis à part pour les plus téméraires d'entre vous qui ont osé jeter un coup d'œil au *VHDL Cookbook* lors du DM, vous ne vous êtes peut-être pas vraiment rendu compte que ce que vous aviez vu de VHDL dans ce DM n'était qu'une infime partie de ce langage. Voici donc quelques compléments qui vont vous être nécessaires pour poursuivre.

1.1 Des types en plus

1.1.1 Le type `std_logic`

Dans le DM, tous les signaux étaient de type `std_logic`, et pouvaient prendre pour valeur `'0'`, `'1'` ou `'U'` uniquement.

En fait, nous vous avions un peu arnaqués à l'époque, car un signal de type `std_logic` peut prendre neuf valeurs différentes (dont `'0'`, `'1'` et `'U'` heureusement) ! La plupart de ces valeurs sont inutiles, et nous continuerons en pratique de n'utiliser que `'0'` et `'1'`. Pour information, voilà la liste complète :

Valeur	Nom	Signification
`'U'`	Non initialisé	Aucune valeur n'a été affectée au signal
`'X'`	Inconnu	Impossible de déterminer la valeur du signal
`'0'`	0 logique	
`'1'`	1 logique	
`'Z'`	Haute impédance	Le signal est isolé grâce à un transistor (*cf.* le bus I^2C)
`'W'`	Faible	Le signal est faible
`'L'`	0 faible	Le signal est faible, mais proche de 0
`'H'`	1 faible	Le signal est faible, mais proche de 1
`'-'`	Sans importance	La valeur du signal n'est pas importante

Vous rencontrerez assurément `'X'` lors de vos simulations, mais ne vous inquiétez pas, il est assez inoffensif.

1.1.2 Le type `std_logic_vector`

Pour vous simplifier la tâche, nous avions aussi volontairement omis de vous parler du type `std_logic_vector`, qui représente simplement un vecteur de signaux de type `std_logic`.

La taille d'un tel vecteur est définie de la manière suivante :

`std_logic_vector(`n `downto` m`)`

où m et n ($m \leq n$) représentent les indices minimal et maximal du vecteur.

Par exemple la déclaration :

```
signal x : std_logic_vector(9 downto 0);
```

définit un signal `x` composé de 10 sous-signaux de type `std_logic`, indexés de 0 à 9 inclus.

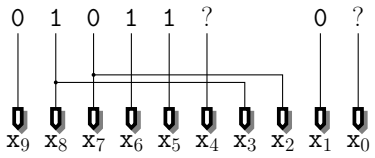
Remarque : Pourquoi une notation en (``$n$ `downto` $m$``) et non pas (`` $m$ `to` $n$ ``) ? Car la notation `downto` nous permet de représenter les bits de poids forts à gauche, ce qui est beaucoup plus lisible dès qu'on veut manipuler des signaux représentant des nombres.

Les sous-signaux d'un vecteur peuvent être accédés individuellement ou par "tranches" :

```
x(1) <= '0';                -- x(1) est de type std_logic
x(9 downto 5) <= "01011";   -- une tranche est aussi un std_logic_vector
x(3 downto 2) <= x(8 downto 7);
```

Remarque : Notez bien l'utilisation des guillemets `" "` pour désigner une constante de type `std_logic_vector`.

Les affectations sur le signal `x` sont représentées figure suivante :



1.1.3 D'autres types

Il existe d'autres types pour les signaux, mais la plupart d'entre eux sont à éviter car ils n'offrent généralement pas de garantie quand à l'implémentation réelle de ces signaux. On peut tout de même noter le type `boolean`, dont les seules valeurs possibles sont les constantes `true` et `false`, le type `integer`, ainsi que les types énumérés.

Le type `boolean` sert essentiellement dans les expressions conditionnelles (pratiques pour décrire un multiplexeur !) du type $expr_1$ `when` $cond$ `else` $expr_2$. Ainsi, la condition d'égalité `a = '0'` est de type `boolean` dans l'exemple suivant, qui décrit un multiplexeur affectant au signal `y` le signal `x0` ou `x1` selon que `a` vaut `'0'` ou `'1'` respectivement :

```
y <= x0 when a = '0' else x1;
```

Quant au type `integer`, il permet de représenter des nombres entiers signés sur 32 bits (de $-2\,147\,483\,647$ à $2\,147\,483\,647$), comme par exemple les indices d'un signal `std_logic_vector`.

Enfin, les types énumérés se déclarent de la manière suivante :

`type` t `is (`cst_1`,` cst_2`, ...);`

Un signal déclaré du type t peut donc prendre comme valeur uniquement une des constantes cst_1, cst_2, ... Cela est très pratique pour définir par exemple le signal représentant l'état courant d'un automate. Le choix de l'encodage de ce signal est alors laissé au synthétiseur VHDL, qui fera ce boulot généralement mieux que nous de toute façon.

Remarque : En fait, les types `std_logic` et `boolean` sont aussi des types énumérés :

```
type std_logic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
type boolean is (true, false);
```

1.2 Bibliothèques et packages

De la même manière qu'en C (ou en OCaml) on peut utiliser des bibliothèques de fonctions déjà écrites grâce à l'instruction `#include` (ou `open` en OCaml), cela est aussi possible en VHDL.

1.2.1 Syntaxe

En VHDL chaque bibliothèque (*library*) comprend plusieurs *packages*. L'inclusion de ces packages s'effectue avant la déclaration d'une entité de la manière suivante :

`library` lib`;`
`use` $lib.pkg_1$`.all;`
`use` $lib.pkg_2$`.all;`

Ces trois lignes chargent donc les packages pkg_1 et pkg_2 de la bibliothèque lib, qui vont ensuite pouvoir être utilisés par l'entité et son architecture.

1.2.2 Package `std_logic_1164`

Contrairement à ce que je vous avais laissé croire dans le DM, le type `std_logic` n'est pas défini par défaut dans VHDL. Il faut pour cela inclure le package `std_logic_1164` de la bibliothèque `ieee` :

```
library ieee;
use ieee.std_logic_1164.all;
```

Ce package a le bon goût de définir aussi le type `std_logic_vector` dans la foulée, ainsi que tous les opérateurs classiques sur ces types (`and`, `or`, ...).

1.2.3 Package `std_logic_arith`

Toujours dans la bibliothèque `ieee`, le package `std_logic_arith` définit plusieurs composants et fonctions, parmi lesquelle la très utile `conv_std_logic_vector`. Cette fonction s'appelle de la manière suivante :

`conv_std_logic_vector(`i`,` n`)`

où i et n sont des entiers, et elle renvoie un objet de type `std_logic_vector(`n`-1 downto 0)`, représentant la valeur de i codée en binaire sur n bits. Nous éviterons cependant d'utiliser cette fonction ailleurs qu'en simulation.

1.2.4 Package `std_logic_unsigned`

Un autre package important est `ieee.std_logic_unsigned`, qui définit quelques opérations arithmétiques de plus haut niveau (essentiellement `+`, `-` et décalages à gauche `shl` et à droite `shr`) sur le type `std_logic_vector` (considéré alors comme codage binaire pour des entiers non signés).

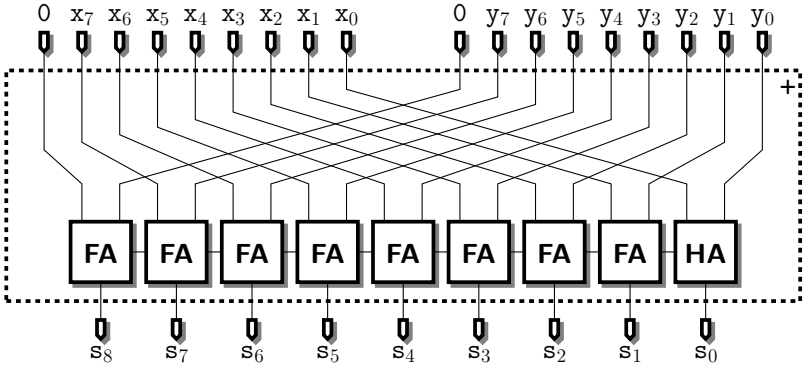
Ainsi, l'addition de deux signaux de 8 bits `x` et `y` de type `std_logic_vector(7 downto 0)` peut s'effectuer de la manière suivante :

```
s <= x + y;
```

Remarque : Il est à noter ici que le signal `s` fait lui aussi 8 bits, c'est-à-dire que la retenue sortante de l'addition est perdue. Pour la conserver, on peut manuellement étendre l'addition sur 9 bits, en déclarant `s` de type `std_logic_vector(8 downto 0)` et en effectuant :

```
s <= ("0" & x) + ("0" & y);   -- l'opérateur & désigne la concaténation
```

Cela est représenté sur la figure suivante :



1.3 Processes

Lors du DM, nous vous avions aussi caché tout ce qui concerne les *processes*, qui sont le "côté obscur" de VHDL (c'est-à-dire le VHDL *comportemental*, par opposition au *structurel* que vous avez vu dans le DM). Mais ce coup-ci, vous ne pourrez pas y couper.

1.3.1 Principe

Les processes servent à décrire des comportements *synchrones* dans une architecture. Ils se présentent de la manière suivante :

```
process(signal_1, signal_2, ...)
begin
  instr_1;
  instr_2;
  ...
end process;
```

Les signaux indiqués en début de process sont appelés liste de sensibilité (*sensitivity list*) du process. En fait, un process peut être vu comme une sorte de sous-fonction, qui est appelée à chaque fois qu'un événement (changement de valeur) se produit sur un des signaux de sa liste de sensibilité. Cette liste peut bien entendu être vide, auquel cas le process est appelé en boucle.

La grande caractéristique du process concerne le mode d'exécution de ses instructions. En effet, même s'il se trouve au milieu d'une architecture (dont les instructions sont donc exécutées toutes en parallèle), le process assure la *séquentialité* de l'exécution de ses instructions, comme pour n'importe quel langage impératif classique.

Les instructions composant le cœur du process peuvent être principalement de trois types (en fait il y en a d'autres, mais ces trois-là sont largement suffisants) :

Affectations Même principe que les affectations dans une architecture :

```
signal <= expr;
```

Constructions de plus haut niveau Ce sont typiquement des conditionnelles ou des boucles. Par exemple :

```
if cond then
  instr_1;
  instr_2;
  ...
elsif cond' then
  ...
else
  ...
end if;
```

```
while cond loop
  instr_1;
  instr_2;
  ...
end loop;
```

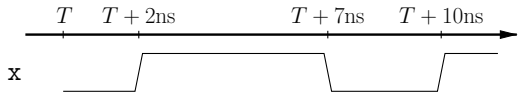
Temporisations Ces instructions permettent de donner des indications de temps concernant l'enchaînement séquentiel des instructions. L'instruction de temporisation la plus courante est le `wait` :

```
wait for n ns;
```

Cette instruction permet d'attendre n nanosecondes (on peut bien évidemment changer l'unité de temps) avant d'exécuter les instructions suivantes. Par exemple, dans un process, les instructions :

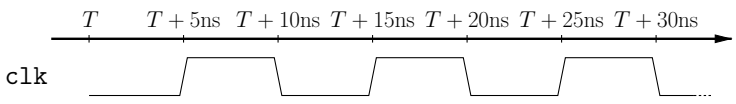
```
x <= '0';
wait for 2 ns;
x <= '1';
wait for 5 ns;
x <= '0';
wait for 3 ns;
x <= '1';
```

génèreront le chronogramme suivant (où le temps T correspond au temps au début de l'exécution de ces instructions) :



Un process sans liste de sensibilité étant exécuté en boucle, on peut ainsi aisément générer un signal d'horloge, comme le suivant, cadencé à 100MHz (période de 10ns) :

```
process
begin
  clk <= '0';
  wait for 5 ns;
  clk <= '1';
  wait for 5 ns;
end process;
```



1.3.2 Restrictions d'utilisation

Généralement, à ce moment-là, l'étudiant qui découvre VHDL et les processes s'écrie :

"Whouah mais c'est trop bien les processes ! Je vais arrêter de m'embêter à écrire du VHDL structurel, et écrire du comportemental comme si j'écrivais du C ![1]*"*

Ce à quoi les TD-men avertis s'empressent de répondre :

"Houla non, surtout pas malheureux ! ![2]*"*

En effet, sauf cas rares (que l'on verra dans la suite), les processes ne sont pas synthétisables. C'est-à-dire qu'ils vont très bien fonctionner pour tout ce qui est simulation, mais en ce qui concerne la réalisation effective du circuit (sous forme de fils et de transistors), le syntétiseur se retrouve très vite dans les choux face à des processes.

L'utilisation de ces processes est donc **strictement** limitée aux fichiers de simulation, car ils nous permettront dans ce cadre de générer facilement signaux d'horloge et stimuli.

1.3.3 Application aux registres

En fait, la seule utilisation des processes autorisée dans une architecture VHDL destinée à la synthèse est pour décrire des registres. Comme vous l'avez remarqué dans le DM, les

[1] En fait, dans la vraie vie, l'étudiant a plutôt tendance à s'écrier "*Bouh, c'est nul VHDL, j'vais plutôt aller me boire une mousse au foyer !*", mais bon, c'était pour l'exemple.

[2] Ou bien "*OK, mais ramène-nous en une aussi s'il-te-plaît !*".

éléments de mémoire décrits directement vous obligent à utiliser des boucles combinatoires, dont la stabilisation n'est jamais très claire.

Les processes offrent une alternative à cela, puisqu'ils sont par définition des éléments synchrones du circuit. Jetons par exemple un œil au process suivant :

```
process(clk)
begin
  if clk'event and clk = '1' then
    q <= d;
  end if;
end process;
```

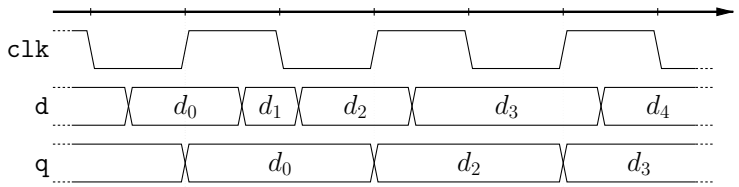
Ce process a le signal `clk` dans sa liste de sensibilité. Cela signifie qu'il est exécuté à chaque fois qu'un événement (changement de valeur) se produit sur `clk`. Il sera donc appelé sur chaque front (montant comme descendant) de notre horloge.

L'instruction `if` nous permet de vérifier les conditions suivantes :

– qu'il s'agisse bien d'un front d'horloge (`clk'event`),

– et que ce front soit montant, c'est-à-dire que l'horloge soit désormais à 1 (`clk = '1'`).

Une fois ces deux conditions vérifiées, le signal `q` reçoit la valeur de `d`. Si l'on trace le chronogramme correspondant, on voit bien que le comportement de ce process est le même que celui d'une bascule D (*flip-flop*) :

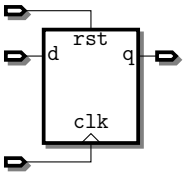


C'est là que le synthétiseur réalise une belle prouesse face à un tel process : en observant le chronogramme du process, il est capable de reconnaître un registre qu'il synthétisera alors proprement en utilisant sa propre bibliothèque de cellules standards.

Mais c'est aussi là que se situe sa limite : mis à part ce process et quelques variations, point de salut. Parmi les variations acceptables, on peut trouver :

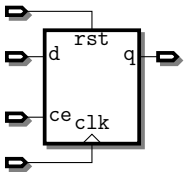
– la bascule D avec port de *reset* (`rst`) :

```
process(rst, clk)
begin
  if rst = '1' then
    q <= '0';
  elsif clk'event and clk = '1' then
    q <= d;
  end if;
end process;
```



– la bascule D avec *clock-enable* (`ce`) :

```
process(rst, clk)
begin
  if rst = '1' then
    q <= '0';
  elsif clk'event and clk = '1' and ce = '1' then
    q <= d;
  end if;
end process;
```



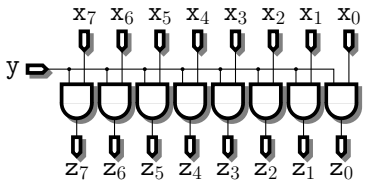
1.4 Génération de circuit

1.4.1 Boucles de génération

Parfois, lorsqu'un circuit (ou tout du moins un bout de celui-ci) est très régulier, il est plus simple de le générer à l'aide d'une boucle plutôt que d'écrire plusieurs fois la même chose en changeant juste les indices dans les vecteurs de signaux.

Prenons par exemple le bout d'architecture suivant :

```
z(0) <= x(0) and y;
z(1) <= x(1) and y;
z(2) <= x(2) and y;
z(3) <= x(3) and y;
z(4) <= x(4) and y;
z(5) <= x(5) and y;
z(6) <= x(6) and y;
z(7) <= x(7) and y;
```



Cela reste encore lisible pour huit signaux auxquels on applique juste une porte `and`, mais pour un nombre plus important de signaux ou des fonctions plus complexes, ça devient rapidement incompréhensible, source potentielle de plein d'erreurs et bien entendu indébuggable.

VHDL permet d'éviter cela en adoptant une construction à base de boucle :

$inst$: `for` i `in` m `to` n `generate`
  $instr_1$;
  $instr_2$;
  ...
`end generate;`

où i représente l'indice de boucle entier (qui n'a pas besoin d'être déclaré au préalable) qui va parcourir l'intervalle de m à n (inclus), instanciant à chaque fois le groupe d'instructions indiqué dans le corps de la boucle. L'identifiant $inst$ va servir à identifier de manière unique chacune des instances du groupe d'instructions, en les nommant de $inst_m$ à $inst_n$.

Pour revenir à l'exemple précédent, on peut donc écrire plus simplement les lignes suivantes, qui généreront exactement le même circuit :

```
inst_and : for i in 0 to 7 generate
  z(i) <= x(i) and y;
end generate;
```

1.4.2 Conditionnelle

On peut aussi trouver la version conditionnelle du `generate` qui fonctionne de la même manière :

inst `: if` *cond* `generate`
  $instr_1$`;`
  $instr_2$`;`
  ...
`end generate;`

2 Symphony EDA : un simulateur qu'il est gratuit[3]

Le simulateur VHDL que vous allez utiliser pour simuler vos composants s'appelle Symphony EDA. Ce n'est carrément pas un simulateur très puissant, mais il a déjà le mérite d'être disponible en version d'évaluation gratuite (mais un peu bridée) et qui plus est sous Linux.

2.1 Premier contact

Pour le lancer, il faut dans un premier temps d'appeler un script qui va définir certaines variables d'environnement. Si vous utilisez `bash`, lancez :

```
. /home/jdetrey/simili30.sh
```

ou bien, si vous utilisez `tcsh` :

```
source /home/jdetrey/simili30.csh
```

Ensuite, pour accéder à Sonata, l'interface graphique de Symphony EDA, lancez la commande suivante :

```
sonata &
```

S'ouvre alors une jolie fenêtre divisée en quatre parties : l'éditeur de fichiers sur la droite, et la gestion de la structure du projet dans la colonne sur la gauche ; en bas, sur la droite une petite console et sur la gauche, la liste des signaux accessibles (lors d'une simulation).

2.2 Débuggons z'ensemble un décodeur pour afficheur 7 segments

2.2.1 Ouverture du projet

Allez sur la page web des TD pour récupérer le fichier `demo1_decoder.tar.gz`, et décompressez-le. Ensuite, dans Sonata, allez dans le menu File puis Open Workspace... et sélectionnez le fichier `decoder.sws` fraîchement décompressé.

Admirez sur la gauche la hiérarchie des fichiers. Ce projet est donc composé de deux fichiers :

- `decoder.vhd`, qui contient véritablement le décodeur à simuler, et
- `decoder_test.vhd`, qui contient le composant de test qui va nous permettre de vérifier le bon fonctionnement du décodeur.

Si vous allez voir dans l'onglet Modules et que vous déroulez le contenu de la bibliothèque `decoder`, vous voyez apparaître les deux entités et les deux architectures définies dans le projet, qui correspondent aux deux fichiers vus précédemment.

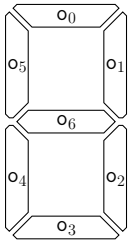
2.2.2 Le composant principal

Double-cliquez sur l'entité `decoder` pour jeter un œil au VHDL correspondant[4]. Le décodeur a donc deux ports, le premier `i` en entrée, représentant en binaire le chiffre hexadécimal à afficher (donc sur 4 bits), et le second `o` en sortie, contenant les 7 signaux (numérotés comme

[3] À défaut d'être puissant.

[4] Si vous y repérez une faute, bravo ! Mais par contre, ne la corrigez pas tout de suite, on va faire joujou avec le simulateur avant.

indiqué figure suivante) à destination de l'afficheur 7 segments. Quant à l'architecture, pas de surprise, on utilise la construction `when` pour se simplifier la vie. Profitez-en d'ailleurs pour observer l'utilisation des `std_logic_vector`.



2.2.3 Le composant de test

Allez maintenant voir du côté du composant de test `decoder_test`. Ce coup-ci, aucun port d'entrée/sortie : tous les signaux vont être générés par ce composant. Dans l'en-tête de l'architecture sont d'abord définis les trois signaux `done`, `passed` et `ok` de type `boolean` sur lesquels nous reviendrons plus tard. Ensuite se trouve la déclaration du composant `decoder` pour qu'il puisse être instancié depuis `decoder_test`. Enfin, les définitions des signaux `i`, qui va être appliqué en tant que stimulus d'entrée au décodeur, et `o`, qui va recevoir la réponse du décodeur.

Le fonctionnement de ce composant de test est relativement classique : en envoyant un ensemble de stimuli au décodeur, puis en comparant la réponse obtenue avec la réponse attendue, il va pouvoir vérifier s'il fonctionne bien (tout du moins sur cet ensemble de stimuli). D'où le comportement des trois signaux booléens :

- Le signal `ok` nous indiquera pour chacune des réponses si celle-ci coïncide avec la réponse attendue ou pas, nous permettant donc de reprérer sur quels stimuli notre composant a échoué.
- Le signal `passed`, initialisé à `true`, restera vrai tant que le composant n'aura pas fait d'erreur. Ainsi, si à la fin du test, `passed` est toujours vrai, c'est que le composant a réussi le test. Par contre, s'il est faux, cela signifie qu'au moins une erreur a été détectée durant la simulation.
- Enfin, le signal `done` passe à `true` une fois le test achevé (c'est-à-dire que l'ensemble des stimuli a bien été soumis au composant à tester).

Dans l'architecture, on y trouve donc d'abord l'instanciation du décodeur, suivi de deux processes :

- le premier est chargé de générer les stimuli, au rythme d'un toutes les 10 nanosecondes,
- et le second, décalé d'une nanoseconde par rapport au premier (pour laisser le temps au décodeur de réagir aux stimuli), vérifie les réponses données par le décodeur, en mettant le signal `ok` à jour selon la réponse.

2.2.4 Compilation et simulation

Pour compiler les deux fichiers VHDL, allez dans le menu Compile[5] et sélectionnez-y l'option Build (smart compile). Le compilateur vous raconte alors des trucs dans la console en bas.

[5] *"Thank you, Captain Obvious!"*

Avant de lancer la simulation, il nous faut d'abord spécifier quelle entité nous voulons simuler. Pour cela, dans l'onglet Modules de la colonne de gauche, cliquez avec le bouton droit sur l'entité `decoder_test`, puis sélectionnez l'option Set `'decoder_test'` as top level. Ensuite, vous pouvez aller dans le menu Simulate puis sélectionner Go pour démarrer le simulateur.

Un chronogramme s'ouvre alors dans l'éditeur de fichiers, ainsi qu'un nouvel onglet Hierarchy dans la colonne de gauche, où se trouve représentée la hiérarchie des instances de composants. Enfin, en bas à gauche apparaissent les signaux définis dans `decoder_test`.

Pour ajouter ces signaux au chronogramme, cliquez dans cette zone avec le bouton droit, et sélectionnez Add all signals to waveform. Vous pouvez aussi si vous le souhaitez afficher les signaux de l'instance `decoder_0` du décodeur, en double-cliquant sur cette instance dans la colonne de gauche (ou bien bouton droit et option Set scope to `':decoder_test:decoder_0:'`).

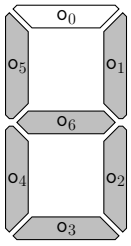
Pour lancer la simulation proprement dite, vous n'avez plus qu'à aller dans le menu Simulate et sélectionner Run, qui va simuler le composant sur la durée indiquée dans la barre d'outils (100ns par défaut), ou bien Run All, qui va simuler le composant jusqu'à ce que tous les signaux arrêtent de changer de valeur (c'est-à-dire qu'il n'y ait plus un seul événement dans la file d'attente du simulateur).

2.2.5 Argh, un bug !

En simulant un peu notre composant de test, vous vous apercevez que la valeur du signal `done` passe à `true` après 161ns de simulation. Et qu'à cet instant-là, la valeur de `passed` est à... `false`. Mince, il y a donc un bug dans le circuit[6].

La première étape est de repérer à quel moment le signal `passed` devient faux. En remontant rapidement le chronogramme, on trouve qu'il s'agit de l'instant 111ns. D'ailleurs, en regardant la valeur de `ok`, on se rend compte qu'il n'y a qu'à cet instant que ce signal est à faux. Nous n'avons donc qu'une erreur dans le test, mais c'en est déjà une de trop.

Regardons donc les valeurs de `i` et de `o` à ce moment-là : `i` vaut `"1011"` (soit le chiffre hexadécimal B) et `o` vaut `"1111110"`, ce qui nous donne l'affichage suivant :



Il y a donc bien un problème ! Le segment correspondant au fil `o(1)` devrait être éteint, pour que l'on puisse y lire un `b`. Heureusement, en allant farfouiller dans l'architecture du décodeur, on trouve bien vite la constante erronée dans la branche correspondante du `when`, qu'il suffit alors de corriger.

En relançant la compilation puis la simulation, ce coup-ci, tout roule comme sur des marchettes. Ouf !

2.2.6 Oui, mais, heu...

Et là, normalement, vous êtes censés me demander :

[6] Quelle surprise !

"Oui, mais, heu... Comment faire si y'a un bug dans le fichier de simulation ?"

Et vous n'auriez pas tort du tout, car effectivement, c'est une éventualité à ne pas écarter.

En général (même si ce n'est pas le cas du tout dans cet exemple), le fichier de simulation (ou bien le programme C qui aura servi à le générer automatiquement) est beaucoup plus court que le code VHDL à tester, et donc la probabilité d'y faire des erreurs est bien plus faible.

Cependant, soyez toujours vigilants lors de vos simulations, et n'hésitez pas à mettre en doute le composant de test autant que le composant testé.

2.3 Un multiplieur 8×8 bits pipeliné

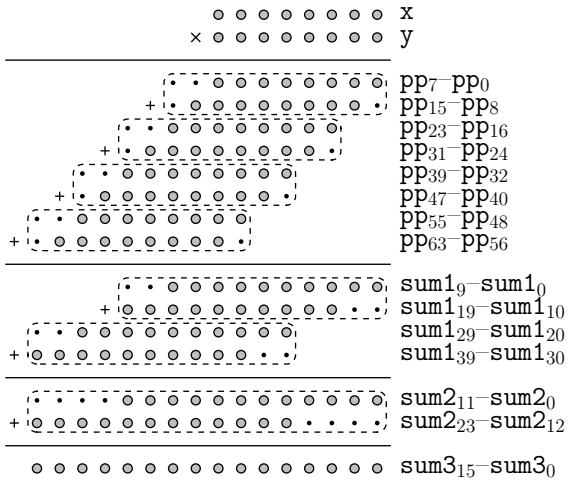
2.3.1 Ouverture du projet

Retournez sur la page web, et téléchargez-y ce coup-ci le fichier `demo2_mult.tar.gz`. Décompressez-le, puis ouvrez le projet `mult.sws` dans Sonata.

Le projet est encore une fois composé de deux fichiers et de deux composants, `mult` qui définit le multiplieur pipeliné, et `mult_test` qui définit le composant de test pour ce multiplieur.

2.3.2 Le composant principal

Ce multiplieur est basé sur une génération en parallèle de tous les produits partiels, suivie d'un arbre d'additions. Des registres sont insérés après la génération des produits partiels, puis après chaque étage d'additions. Remarquez comme les suffixes `_`i des noms des signaux correspondent à l'étage du pipeline dans lequel interviennent ces signaux.



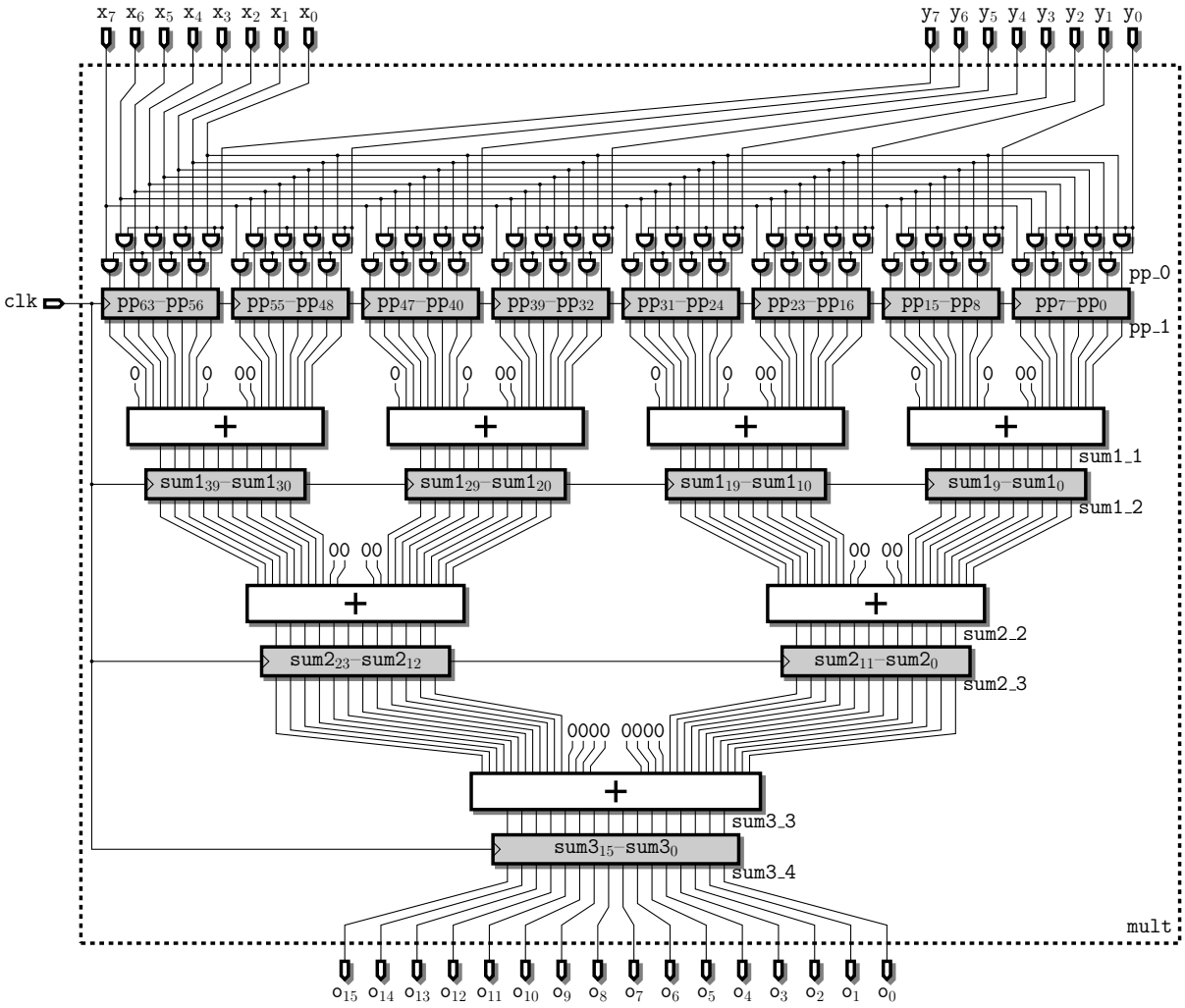
Les produits partiels sont donc générés par une double boucle `for ... generate` de sorte que le signal `pp_0`$_{8j+i}$ reçoive le produit partiel `x`$_i \cdot$ `y`$_j$. Les 64 bits du signal `pp_0` sont ensuite stockés dans un registre. Le signal en sortie du registre est noté `pp_1`.

Les produits partiels sont ensuite répartis en huit groupes de 8 bits chacun : `pp_1`$_{8j+7}$–`pp_1`$_{8j}$ correspondant au produit `x`$\cdot$`y`$_j$. Les premières additions s'effectuent donc sur les quatre paires consécutives de ces groupes, avec les décalages adéquats. Remarquez au passage que pour ces premières additions, une retenue est succeptible de se propager, il faut donc l'absorber en rajoutant artificiellement des `'0'` en poids forts. Ces additions nous donnent donc quatre nombres de 10 bits, sous forme du signal `sum1_1`, qui est à son tour stocké dans un registre, pour donner `sum1_2` à l'étage suivant du pipeline.

Les quatre sommes partielles `sum1_2`$_{10j+9}$–`sum1_2`$_{10j}$ sont ensuite aussi additionnées deux par deux, en tenant compte bien-sûr des décalages. Ce coup-ci, il est facile de prouver qu'aucune retenue supplémentaire ne peut se propager hors de l'additionneur, donc pas besoin de ruser. On obtient donc le signal `sum2_2`, composé des deux sommes de 12 bits, qui devient `sum2_3` après un étage de registres.

Enfin, la dernière somme s'effectue de la même manière, et ne provoque pas non plus de retenue sortante. On obtient donc le nombre de 16 bits `sum3_3`, qui passe par un dernier étage de pipeline avant d'être placé sur le port de sortie du composant.

On obtient donc le circuit suivant, effectivement pipeliné sur quatre étages :



2.3.3 Le composant de test

Ce coup-ci, le composant de test est bien plus simple que pour le décodeur 7 segments. On y retrouve bien les signaux usuels `done`, `passed` et `ok`, ainsi que les signaux de stimulus et de réponse pour le composant testé.

Le signal d'horloge à 100MHz est donc généré par le premier process, comme vu précédemment.

Le principal stimulus soumis au composant est généré par le second process, qui énumère tout simplement toutes les valeurs possibles pour les entrées `x` et `y` (on parle alors de test exhaustif[7]).

Enfin, le dernier process, après avoir attendu les quatre cycles d'horloge nécessaires à la propagation des valeurs au travers du pipeline du multiplieur, vérifie que le résultat donné par le multiplieur correspond bien au produit de ses deux opérandes.

2.3.4 Argh, un bug ! (bis)

Compilez et simulez le tout sur quelques cycles. Ça semble marcher correctement. Poursuivez la simulation sur une petite dizaine de microsecondes. Et là, constatez avec stupéfaction[8] que le signal `passed` est à `false`.

Allons donc voir ce qu'il se passe. En sélectionnant le signal `passed` dans le chronogramme, puis en cliquant sur le bouton Go to next transition (le dernier sur la droite de la barre d'outils), vous pouvez accéder directement à l'instant où `passed` est passé à faux (pour les flemmards, c'est à $2,641\mu$s). Effectivement, le résultat obtenu est 1 alors que les opérandes, qui sont donc décalés de quatre cycles en arrière, sont respectivement 1 et 4.

Malheureusement, ici, la correction est moins simple que pour le décodeur, car le circuit est bien plus complexe. Il faut donc effectuer la simulation avec le détail des signaux internes de l'opérateur.

Pour cela, commencez par supprimer tous les signaux affichés dans le chronogramme[9], puis sélectionnez l'option Restart dans le menu Simulate. De là, ajoutez au chronogramme les signaux `ok` et `clk` du composant `mult_test`, ainsi que `x`, `y`, `pp_1`, `sum1_2`, `sum2_3` et `sum3_4` du composant `mult`. Vous pourrez ainsi voir les données contenues dans chacun des registres de l'opérateur, ce qui suffit généralement à localiser le bug.

Simulez donc le circuit jusqu'à l'instant fatidique. Si vous regardez plus précisément ce qu'il se passe, vous pouvez constater que les produits partiels (`pp_1`) sont corrects, ainsi que les premières sommes (`sum1_2`). Par contre, la somme $\texttt{sum2\_3}_{11}$–$\texttt{sum2\_3}_0$ (celle des poids faibles) est fausse.

Et effectivement, l'erreur se trouve bien sur la ligne 66 de `mult.vhd`. Vous pouvez la corriger puis relancer la simulation exhaustive pour vérifier que tout est bien correct.

[7] Question : en général, est-ce que ce genre de test exhaustif est suffisant ?

[8] Ou pas.

[9] La version gratuite de Symphony EDA ne permet d'afficher que 10 signaux à la fois dans un chronogramme. Vous allez voir que ça peut devenir extrêmement vexant, mais on n'a pas le choix sur ce coup.

3 FPGAは何(なん)ですか。

Cette section n'a pas vocation à être un cours complet sur les FPGA, mais juste une présentation rapide pour que vous ayiez une idée de la manière dont votre processeur RISC va être implanté.

3.1 Vue générale

Le paradigme qui se cache derrière les FPGA est le parallélisme poussé à l'extrême, jusqu'au niveau le plus fin (on parle de *fine-grain parallelism*) : il s'agit d'avoir un grand nombre de toutes petites unités de calcul programmables, reliés entre eux par suffisamment de fils pour ne pas risquer de congestion.

Ces unités de calcul (ou PE, pour *processing elements*) sont toutes identiques et généralement regroupées en îlots de 4 ou 8 PE en général. Ces îlots sont ensuite disposés sur une grille bidimensionnelle : un FPGA classique en compte plusieurs milliers[10].

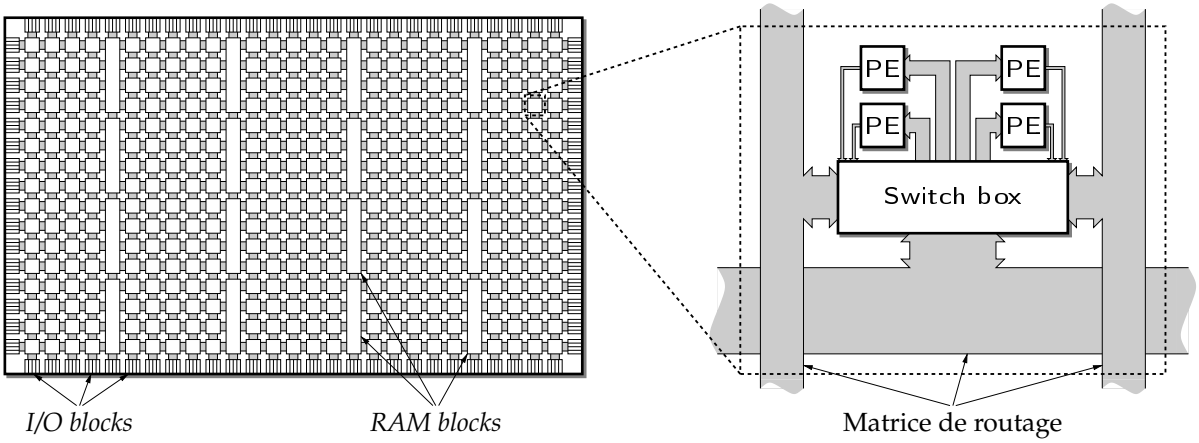
Les îlots sont ensuite reliés entre eux grâce à une matrice de routage programmable, permettant ainsi d'effectuer toutes les connections nécessaires entre les îlots d'un circuit, et ce où qu'ils soient situés sur le FPGA.

On trouve aussi sur tout le pourtour du FPGA des ports d'entrée/sortie (*I/O blocks*) eux aussi programmables et accessibles par la matrice de routage, permettant ainsi de lire ou d'envoyer des signaux sur les plots du FPGA, et donc de l'interfacer avec le monde extérieur.

Généralement, un FPGA contient aussi de quelques dizaines à plusieurs centaines de petits blocs de RAM (*RAM blocks*) de 16 ou 18kbits chacun, disposés en colonnes régulièrement réparties sur la surface de la puce.

Sur les modèles les plus récents, on y trouve aussi des petits multiplieurs 18×18 bits, voire carrément des cœurs de processeurs classiques (PowerPC ou ARM).

Voici de manière très simplifiée l'architecture globale d'un FPGA, ainsi que le détail d'un îlot :

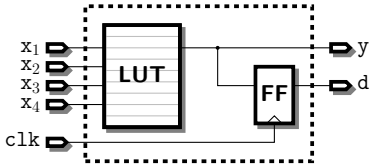


[10] Celui que nous allons utiliser en a $13\,696$, mais c'est un gros modèle.

3.2 Anatomie d'un PE

En simplifiant là aussi grandement, un PE peut être réduit à une petite mémoire programmable de $2^4 \times 1$ bit (ou LUT, pour *look-up table*), à laquelle est adjoint un registre de type flip-flop (FF).

La table permet donc d'implanter n'importe quelle fonction booléenne à 4 variables en stockant dans la LUT la table de vérité de cette fonction. Le bit du résultat de la table peut alors soit être stocké dans le registre, soit être routé directement vers un autre PE pour la suite du calcul.



Ainsi, grâce à la matrice de routage, on peut cascader les PE pour calculer n'importe quelle fonction booléenne, et les FF nous permettent de créer des registres pour nos signaux. A priori, on dispose là de tout ce qu'il nous faut pour y implanter notre processeur RISC !

4 Notre plateforme

4.1 Vue d'ensemble

Le FPGA sur lequel nous allons travailler se trouve sur une carte de développement munie de plein de ports d'entrée/sortie tels que USB, RS232 (port série), PS/2, audio, SATA, VGA,... ainsi que d'un emplacement pour une barrette de mémoire DDR. Tous ces ports et périphériques sont bien entendu directement interfacés avec le FPGA grâce aux I/O blocks de celui-ci. La carte dispose en plus d'une horloge cadencée à 100MHz, reliée elle aussi à un port d'entrée du FPGA.

Histoire d'avoir un système à peu près interactif, vos TD-men vénérés ont écrit un contrôleur pour la sortie VGA ainsi qu'un contrôleur pour port PS/2, tous deux implantés sur le FPGA. Nous pourrons donc utiliser un clavier pour interagir avec le processeur, et visualiser la sortie sur écran.

4.2 Contrôleur vidéo

Le contrôleur vidéo affiche à l'écran une résolution de 1280×1024 pixels en 24 bits / pixel (16 millions de couleurs). Cependant, comme il faut stocker l'image complète en mémoire avant de pouvoir l'afficher, la résolution effective sera uniquement de 320×256 pixels en 8 bits / pixel (256 couleurs), ce qui sera largement suffisant pour s'amuser déjà un bon peu.

Les 8 bits d'un pixel codent la couleur de ce pixel de la manière suivante :
- les 3 bits de poids forts (de 7 à 5) codent la composante rouge,
- les 2 bits suivants (de 4 à 3) codent la composante verte,
- et les 3 derniers bits (de 2 à 0) codent la composante bleue.

On a donc le codage : `RRRGGBBB`.

Les 320×256 pixels représentent 80ko de mémoire réservés pour l'affichage, auxquels le contrôleur accèdera directement (on parle de DMA, pour *Direct Memory Access*).

4.3 Contrôleur PS/2

Le protocole PS/2 est un protocole série sur deux fils, à la I^2C, le maître étant le périphérique. Ce bus est bidirectionnel : le clavier (ou la souris) peut écrire sur le bus, mais peut aussi recevoir des données, comme par exemple l'état allumée/éteinte des LED sur le clavier. Le contrôleur écrit pour ce TD étant très simpliste, il se contente d'écouter sur le bus, mais n'y écrit jamais rien.

Les messages envoyés par le périphérique font toujours 11 bits de long : 1 bit de début de transmission, 8 bits de données, 1 bit de parité impaire et enfin 1 bit de fin de transmission.

De manière à ne pas avoir à gérer les phénomènes d'interruptions dans notre processeur, le contrôleur PS/2 fonctionne lui aussi en DMA : une file d'attente cyclique longue de 254 éléments est réservée en mémoire. Dans cette file d'attente ne seront stockés que les 8 bits de données de chaque transmission du périphérique.

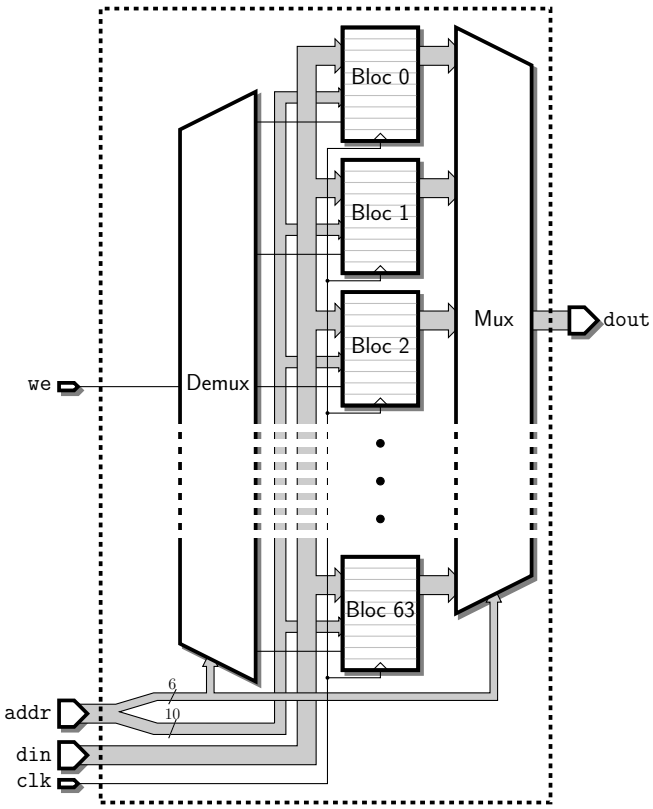
4.4 Mémoire

La mémoire DDR étant réellement une horreur à gérer, nous allons nous contenter d'utiliser la mémoire disponible sur le FPGA sous forme de RAM blocks de 16kbits. En effet, ces blocs

ont le mérite d'être rapides d'accès, mais aussi et surtout de disposer de deux paires indépendantes de ports lecture/écriture. Ainsi, nous n'aurons pas à nous soucier des accès simultanés à la mémoire entre le processeur, le contrôleur vidéo et le contrôleur PS/2. De plus, les RAM blocks proposent plusieurs modes d'adressage, de 16k $\times$ 1 bit à 512 $\times$ 32 bits, ce qui va nous permettre de gérer différentes tailles de données de manière transparente.

Comme le processeur RISC que nous allons implanter est un processeur 16 bits, les mots mémoire font 16 bits et des adresses de 16 bits aussi. L'espace mémoire maximal est donc $2^{16} \times 16$ bits, soit 64k $\times$ 16 bits $=$ 128ko.

64 RAM blocks de 1k $\times$ 16 bits vont donc être nécessaires à la réalisation de cette mémoire. L'interconnexion des blocs se fait par des multiplexeurs et démultiplexeurs, qui sélectionnent le bon bloc suivant les 6 bits de poids forts de l'adresse, les 10 bits restants adressant effectivement le mot à l'intérieur du bloc, comme indiqué sur la figure suivante. La latence de cette mémoire est d'un cycle d'horloge : il faut attendre un cycle d'horloge entre le moment où l'adresse est déposée sur le bus `addr` et le moment où la donnée est disponible sur le bus `dout`.



Les 64k mots correspondent aux adresses de `0x0000` à `0xFFFF` du point de vue du processeur.

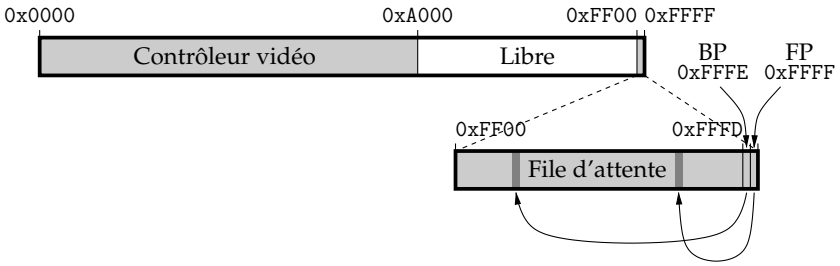
Du point de vue du contrôleur vidéo, les 80ko (soit 40k mots) de mémoire réservés pour l'écran sont mappés de `0x0000` à `0x9FFF`. En fait, pour simplifier le contrôleur vidéo, les blocs de mémoire correspondant à ces adresses-là sont utilisés en mode 2k $\times$ 8 bits (du côté du contrôleur uniquement). Cela n'est pas bien important, car complètement transparent pour le processeur.

Ainsi, la couleur d'un pixel ayant pour coordonnées sur l'écran (x, y) sera stockée dans le

mot d'adresse $\frac{1}{2}(320y + x)$, dans les 8 bits de poids faibles si x est pair, dans les 8 bits de poids forts si x est impair.

Quant à la file d'attente pour le contrôleur PS/2, elle est mappée sur les adresses de `0xFF00` à `0xFFFD`. Même si les données reçues sur le bus PS/2 ne font que 8 bits, elles sont ici stockées sur un mot complet (16 bits donc). Les pointeurs de tête de file (FP, pour *front pointer*, là où sont insérés les nouveaux événements) et de queue (BP, pour *back pointer*, là où le processeur dépile les événements en attente) sont eux mappés aux adresses `0xFFFF` et `0xFFFE` respectivement.

On a donc le mapping suivant, avec la zone de mémoire libre de `0xA000` à `0xFEFF` :



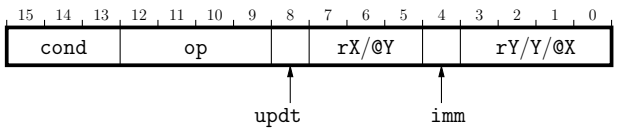
4.5 Processeur RISC 16 bits

Voici une spécification complète du processeur tel qu'il a été défini lors des TD précédents. Il s'agit donc d'une machine à deux adresses, muni de 64 registres avec un système de fenêtrage.

4.5.1 Mot d'instruction

Le mot d'instruction est découpé de la manière suivante :

- bits 15 à 13, `cond` : prédicat conditionnant l'exécution de l'instruction en fonction des bits de statut ;
- bits 12 à 9, `op` : opcode de l'instruction ;
- bit 8, `updt` : drapeau déterminant si les bits de statut doivent être mis à jour par cette instruction ;
- bits 7 à 5, `rX/@Y` : numéro de registre `rX` ou adresse `@Y` de la fenêtre des registres `rY`.
- bit 4, `imm` : drapeau déterminant si le champ suivant correspond à un numéro de registre ou à un immédiat ;
- bits 3 à 0, `rY/Y/@X` : numéro de registre `rY` si `imm` est à 0, ou valeur immédiate `Y` si `imm` est à 1, ou encore adresse `@X` de la fenêtre des registres `rX`.



4.5.2 Jeu d'instructions

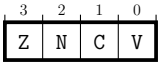
Le jeu d'instructions est très simple, puisqu'il n'y a que 16 instructions possibles.

Opcode	Codage	Action
AND	0000	rX ← rX AND rY/Y
OR	0001	rX ← rX OR rY/Y
XOR	0010	rX ← rX XOR rY/Y
NOT	0011	rX ← NOT rY/Y
ADD	0100	rX ← rX + rY/Y
SUB	0101	rX ← rX − rY/Y
LSL	0110	rX ← rX ≪ rY/Y
LSR	0111	rX ← rX ≫ rY/Y

Opcode	Codage	Action
LDR	1000	rX ← [rY/Y]
STR	1001	[rY/Y] ← rX
MOV	1010	rX ← rY/Y
SWP	1011	wX ← @X wY ← @Y
JRP	1100	PC ← PC + rY/Y
JRN	1101	PC ← PC − rY/Y
JMP	1110	PC ← rY/Y
CAL	1111	rX ← PC + 1 PC ← rY/Y

4.5.3 Statut et conditions

On se donne les 4 drapeaux de statut suivants :
- Z, *Zero* : 1 si le résultat d'une opération est nul, et 0 sinon ;
- N, *Negative* : 1 si le résultat est strictement négatif, et 0 sinon ;
- C, *Carry* : 1 s'il y a dépassement de capacité sur 16 bits, et 0 sinon.
- V, *oVerflow* : 1 s'il y a dépassement de capacité sur 15 bits (dépassement signé), et 0 sinon.



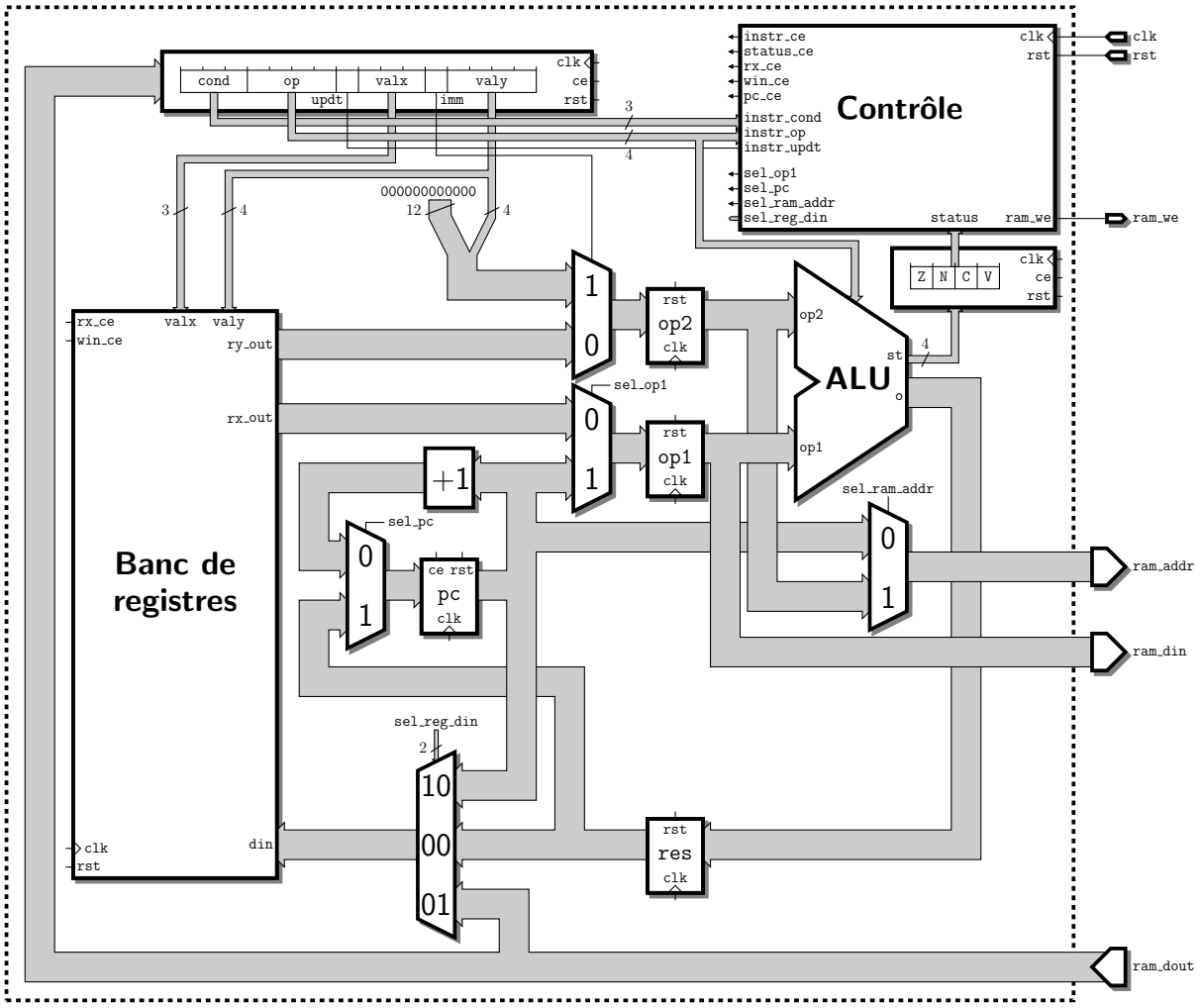
Les prédicats conditionnant les instructions sont alors codés de la manière suivante :

Condition	Codage	Expression
F	000	0
T	001	1
C	010	C
V	011	V
Z	100	Z
NZ	101	¬Z
P	110	(¬Z) ∧ (¬N)
NP	111	Z ∨ N

4.5.4 Cycle d'exécution

Le cycle d'exécution d'une instruction sur ce processeur se déroule en 5 étapes, chacune d'entre elles durant un cycle d'horloge :
- `fetch1` : la valeur de `PC` est mise sur le bus d'adresse vers la mémoire.
- `fetch2` : le mot d'instruction lu en mémoire est stocké dans le registre d'instruction.
- `decode` : les valeurs des opérandes de l'instruction sont stockées dans les registres `op1` et `op2` ; le `PC` est incrémenté ; si la condition d'éxécution de l'instruction est fausse ou bien s'il s'agissait d'un `SWP`, on passe directement à l'instruction suivante.
- `exec` : l'UAL calcule le résultat de l'opération ; l'accès mémoire (en lecture ou en écriture) est aussi initié.
- `store` : le résultat de l'ALU ou de la lecture en mémoire est mémorisé dans le registre adéquat ou dans le PC, selon l'instruction.

Remarque : On se garde bien de pipeliner le tout pour exécuter une instruction par cycle ! C'est bien trop compliqué pour le moment, car il faudrait gérer les conflits qui pourraient alors survenir.



5 À vous de jouer ![11]

Pour réaliser ce processeur, vous allez tous travailler ensemble, et non pas chacun dans votre coin. Répartissez vous donc en quatre groupes à peu près équilibrés :
- le groupe A sera chargé de réaliser les registres (instruction, statut et banc de registres),
- le groupe B s'occupera de l'ALU,
- le groupe C réalisera la partie contrôle,
- et le groupe D décrira l'architecture globale du processeur sur laquelle viendront se greffer les composants précédents.

Avant de vous lancer tête baissée, allez chercher l'archive correspondant à votre groupe sur la page web. Chacune contient un projet pour Symphony EDA, avec une ébauche de VHDL ainsi que des composants de test.

5.1 Registres

Si vous le souhaitez, vous pouvez répartir encore un peu les tâches au sein de ce groupe, en vous répartissant en 2 équipes : l'une pour les registres d'instruction, de statut et le `PC`, et l'autre pour le banc de registres.

5.1.1 Registre d'instruction

Il s'agit d'un registre 16 bits, qui reçoit les données de la mémoire par le port `instr`. Le mot d'instruction stocké est renvoyé sous forme découpée `cond`, `op`, `updt`, `valX` (qui correspond à `rX/@Y`), `imm` et `valY` (qui correspond à `rY/Y/@X`).

Ce registre dispose d'un port de *clock enable*, noté `ce`, ainsi que d'un port de reset `rst`.

5.1.2 Registre de statut

Même topo que pour le registre d'instruction, mis à part que celui-ci ne fait que 4 bits. Les drapeaux de statut arrivent de l'ALU par le port `i`, puis sont stockées dans le registre lorsque le signal de *clock enable* `ce` est à 1.

5.1.3 Banc de registres

Le banc de registre contient donc 64 registres de 16 bits et comme cela avait été décidé en TD, l'accès aux registres est fenêtré.

Le banc de registre contient donc deux petits registres nommés `wX` et `wY` de 4 et 3 bits respectivement, qui contiennent l'adresse de début des fenêtres pour `rX` et `rY`. Le mapping entre registres virtuels (les `rX` et `rY` du mot d'instruction) et registres physiques (notés `pX` et `pY`) est le suivant :

$$\begin{cases} \texttt{rX} \equiv \texttt{p}(\texttt{wX} \times 4 + \texttt{X}), \text{ et} \\ \texttt{rY} \equiv \texttt{p}(\texttt{wY} \times 8 + \texttt{Y}). \end{cases}$$

En fonctionnement normal, `valx` et `valy` désignent respectivement `rX` et `rY`. Par contre, dans le cas de l'instruction `SWP`, les registres `wX` et `wY` prennent les valeurs de `valy` et `valx` respectivement.

[11] C'est pas trop tôt !

Il y a deux signaux *clock enable* pour le banc de registre : l'un désigne l'écriture dans le registre `rX`, et l'autre la mise à jour des registres de fenêtre.

5.2 Unité arithmétique et logique

Vous pouvez ici aussi vous répartir les tâches entre les différentes opérations si vous le souhaitez, mais une approche unifiée pour minimiser la logique serait préférable.

Il faut donc que votre ALU réalise les opérations suivantes :

- AND, OR, XOR et NOT bit à bit : vous pouvez utiliser pour cela les versions vectorielles des instructions `and`, `or`, `xor` et `not`.
- Addition et soustraction : de même, pour vous simplifier la vie, utilisez les opérateurs `+` et `-`. Attention aux dépassements de capacité !
- Décalages à gauche et à droite : ce coup-ci, il va vous falloir vous creuser la tête pour faire un *barrel-shifter*, sachant en plus que les valeurs des décalages peuvent être négatives. Encore une fois, attention aux dépassements de capacité !
- Recopies (pour les instructions comme `MOV` ou `JMP`) : notez que les déplacements de registre à registre passent par l'ALU pour pouvoir mettre les drapeaux de statut à jour.

Pensez aussi à mettre correctement à jour les drapeaux de statut en fonction de la valeur du résultat.

5.3 Unité de contrôle

L'unité de contrôle est composée d'un automate tout simple qui enchaîne de manière cyclique les états `fetch1`, `fetch2`, `decode`, `exec` et `store`. Dans certains cas (instruction `SWP` ou condition fausse) on peut interrompre le cycle plus tôt.

En fonction de l'état courant stocké dans le registre `state_r` ainsi que d'autres paramètres tels que l'instruction ou le statut, cette unité doit générer les signaux de commande appropriés pour contrôler les écritures dans les registres (`*_ce`) ou en mémoire (`ram_we`) ainsi que les multiplexeurs (`sel_*`).

5.4 Architecture globale du processeur

Il s'agit ici de relier tous les composants (registres, ALU, contrôle) entre eux, en insérant multiplexeurs et registres correctement contrôlés aux endroits nécessaires. Il suffit en gros de suivre le schéma global du processeur donné précédemment.

Attention : lors d'un *reset*, le `PC` doit être initialisé à l'adresse du début du programme, soit `0xA000` sinon on va tomber en pleine mémoire vidéo !