VHDL par l'Exemple

Jean-Christophe Le Lann

9 mars 2018

Table des matières

Chapitre 1

Introduction

Ce cours est une introduction au langage VHDL : ce langage permet de décrire, simuler et synthétiser des systèmes numériques. VHDL est un langage important car il accompagne le développement de l'ensemble de la société numérique : tous les circuits fabriqués depuis les 20 dernières années ont été décrits en VHDL (ou Verilog), puis simulés et synthétisés grâce à des outils qui connaissent ces langages. Il est donc incontournable. Il existe certes des tentatives pour s'affranchir des HDLs classiques : on peut noter par exemple MyHDL, basé sur Python. Mais force est de constater que ces tentatives ne font que répliquer les concepts clés de ces HDLs. Ces langages et ces concepts sont relativement complexes au premier abord, mais le jeu en vaut la chandelle, car ils vous permettront d'être extrêmment créatif en matière de systèmes numériques.

1.1 Que peut-on faire avec VHDL ?

Comme nous venons de le dire, VHDL est un langage de *description matérielle* (hardware description). Il est donc très différent de C, Java, Ruby ou Python. Alors que ces derniers *utilisent* un circuit pour s'exécuter [1], VHDL permet de *créer* un tel circuit, ou tout autre circuit. Il ne faut donc pas s'attendre à ce que VHDL rende les mêmes services que les langages que nous venons de citer.

Parmi les circuits que l'on peut créer avec VHDL, on trouve des circuits très classiques comme des microprocesseurs, des microcontrôleurs ou des processeurs de traitement du signal (DSP). A l'inverse, on peut créer des circuits très spécialisés, qui répondent à des besoins précis en calcul et communication : codecs vidéo, appareils de mesure médical, radar et sonar, etc. Ces circuits sont appelés ASIC (application specific integrated circuits) ; ils exhibent des performances surpassant la performance de processeurs généralistes.Il existe enfin un dernier type de circuits, qui peut être vu comme un compromis entre ces deux types de circuits : il s'agit des FPGA. Ces FPGA (Field Programmable Gate Array) sont des circuits reconfigurables, c'est-à-dire des circuits qui peuvent être reprogrammés à volonté. Ils cumulent ainsi les avantages des deux types de circuits précédent, aliant vitesse et reprogrammabilité. Les FPGA sont aujourd'hui utilisés massivement et ils constituent une cible de choix lors de l'apprentissage du langage.

Ce cours doit vous permettre à terme de développer de tels circuits en VHDL et, au final, de les exécuter sur un circuit de type FPGA. On ne devra toutefois pas associer uniquement VHDL à la notion de FPGA et se souvenir qu'un circuit, par exemple prototypé sur un FPGA, peut devenir un ASIC spécialisé, aux performances encore plus redoutables.

1.2 Comment est-il né ?

Le langage VHDL a émergé dans les années 80 et est depuis devenu incontournable dans l'Industrie des Semi-conducteurs. C'est le département à la défense américain (DARPA) qui a fait la commande de VHDL. A cette époque concevoir un système à l'aide de composants venant de différentes sociétés était très délicat et l'industrie de la défense ne faisait pas exception. Alors que toute la Silicon Valley

1. Même la machine *virtuelle* Java a besoin d'un processeur pour s'exécuter !

fourmillait d'inventivité, le *manque de format d'échange* devenait criant. Inspiré d'ADA, le langage informatique phare de l'époque, une première version de VHDL est apparu en 1983, puis 1987. Mais c'est la version VHDL 93 qui s'est imposé et qui continue de représenter l'essentiel des codes disponibles. C'est également cette norme qu'on utilisera ici. Plusieurs évolutions ont été adoptées par le comité de normalisation et il y a fort à parier que VHDL continuera d'être utilisé durant de nombreuses années encore.

1.3 VHDL et Verilog

On ne peut pas parler de VHDL, sans parler de son *alter ego* : Verilog. Ils partagent un grand nombre de points communs, à l'exception de leur syntaxe : on peut reconnaître une lointaine parenté avec le C côté Verilog. En terme d'utilisateurs, Verilog est globalement plus utilisé que VHDL : Verilog est notamment plus utilisé aux Etats-Unis et au Japon, alors que VHDL est fortement ancré en Europe. Mais cette "géopolitique" est approximative, étant donné la mondialisation des développements actuels. Par ailleurs et à titre personnel, je souligne que les systèmes que l'on conçoit aujourd'hui nécessitent le recours aux deux langages à la fois : certains composants (on parle plutôt de blocs de propriété intellectuelle ou *IP*) sont disponibles dans l'un ou l'autre des langages. Les outils commerciaux savent désormais simuler et synthétiser des systèmes décrits dans les deux langages. De même, à certaines étapes d'un développement industriel grandeur nature, il est de mise de recourir à l'un plutôt qu'à l'autre : ainsi, Verilog excelle dans les simulations de portes logiques (c'est-à-dire à bas niveau), alors que VHDL semble permettre de meilleures abstractions à haut niveau. Il est donc très fréquent de devoir passer de l'un à l'autre... Verilog a donc connu une évolution similaire à celle de VHDL, jusqu'à l'avénement du langage SystemVerilog, qui le modernise fortement (notamment concernant les tests automatiques). Dans tous les cas, l'industrie et les besoins en la matière permettent de s'accommoder de la présence de tels langages, aux buts similaires...

1.4 Description de systèmes parallèles

VHDL n'est pas un langage simple. Sa syntaxe verbeuse rebute au premier abord. Pourtant, VHDL et les HDL (hardware description languages) en général présentent des caractéritiques très intéressantes susceptibles de piquer la curiosité de l'informaticien. Parmi ces traits marquants, on trouve la notion de parallélisme : VHDL permet de décrire des systèmes hautement parallèles. On sait que cette gestion du parallélisme est généralement pénible et sujette à erreur dans la plupart des langages de programmation traditionnels. Ici, le parallélisme est immédiat. Par exemple, on peut réaliser plusieurs affectations sans avoir à se soucier de l'ordre dans lequel ces affectations sont décrites dans le langage !

1.5 L'EDA : Electonic Design Automation

Ce cours vous permettra de mettre un pied dans une industrie fascinante : celle de l'EDA. L'EDA (Electronic design Automation) est l'industrie qui développe les outils permettant d'automatiser la conception de circuits. Les simulateurs VHDL en font partie. C'est une industrie vaste et aux pieds solides, qui continue de s'étendre. Pour les lecteurs intéressés, ils consulteront avec intérêt les analyses des experts et notamment de Gary Smith sur les différents axes de l'EDA : de très nombreuses sociétés (Mentor Graphics, Synopsys, Cadence) et start-ups y sont citées (environ 400). L'EDA est souvent présentée comme exemple d'industrie : elle a su accompagner les évolutions rapides de la microélectronique, sur des sujets variés et complexes. L'EDA est notamment consommatrice de développeurs logiciels et d'algorithmiciens (la théorie des graphes est omniprésente).

1.6 Communauté et Open-source

Autres faits marquants dans l'histoire de ces langages : très rapidement, des simulateurs open-source ont vu le jour. Ils ont ainsi permis de démocratiser la pratique de ces langages. Dans ce cours d'introduction, nous utiliserons le simulateur GHDL. GHDL a été développé par une seule personne : le français Tristan Gingold. GHDL est aujourd'hui un simulateur stable, mature et parfaitement utilisable sous Linux, Windows ou Mac. Une petite équipe de volontaire épaule désormais Tristan Gingold et propose sans cesse de nouvelles évolutions.

Comment poser des questions à la communauté ? Il existe plusieurs sites Internet permettant d'échanger autour de questions VHDL.

- On peut par exemple signaler le tag VHDL sur Stackoverflow. Ce site met également à disposition un tutoriel, co-construit par les internautes, toujours selon le principe de gratification par des points.
- Forum Xilinx et Altera

1.7 Ouvrages à consulter

Un grand nombre de livres ont été écrits sur VHDL et continuent de l'être. Parmi les références qui me paraissent les plus proches de ce cours, je n'hésite pas à citer les livres de Pong.P CHU [**?**].

Première partie

Modélisation au niveau RTL

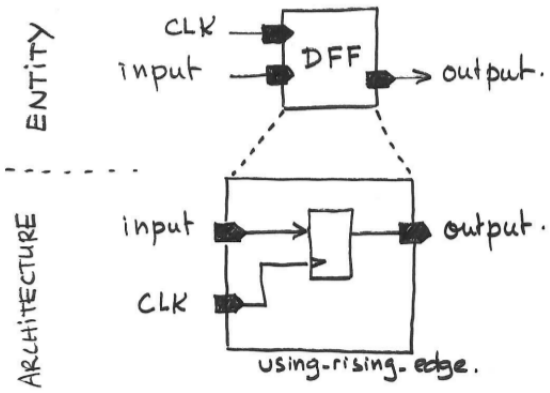
Chapitre 2

Les bases du code RTL

2.1 Registres

Registre simple d'un bit Le code suivant présente la description d'un composant (*entity*) qui contient un registre de 1 bit, le plus simple qui soit. Nous parlerons ici de registre, de bascule ou flip-flop de manière indifférenciée : il s'agit de l'élément séquentiel par excellence : la bascule D. La première partie du code décrit l'entité : c'est l'interface du composant. Le registre à proprement parlé est contenu dans l'ıarchitecture associée à cette entité. La seule manière d'*inférer* un registre est de recourir à un *processus*. Ce processus, éventuellement précédé d'un label, commence par déclarer une liste de sensibilité : comme notre processus n'a vocation qu'à être évalué qu'au front montant de l'horloge (tand en simulation qu'en synthèse matérielle), seule l'horloge apparaît dans cette liste de sensibilité. Le corps du processus contient ici une instruction *if* qui indique que toutes les assignations de signaux qui suivront (il n'y en a qu'une ici) seront *clockées*. Le synthétiseur RTL comprend cette règle simple et n'inférera qu'un simple registre ici. Dans le cas de plusieurs assignations, plusieurs registres, associés aux signaux mentionnés, seront inférés.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity dff is
5    port(
6      clk      : in  std_logic;
7      input    : in  std_logic;
8      output   : out std_logic
9    );
10 end entity;
11
12 architecture using_rising_edge of dff is
13 begin
14
15   -- only means to describe registers : use
         processes !
16   process(clk)  -- only clock is necessary
       here
17   begin
18     if rising_edge(clk) then  --or : clk'
         event and clk='1'
19       output <= input;
20     end if;
21   end process;
22
23 end using_rising_edge;
```



Registre générique de plusieurs bits et reset asynchrone Le second exemple met en avant la capacité de VHDL à décrire des structures *génériques*. Nous y reviendrons plus tard, mais l'exemple reste intéressant : le synthétiseur possède ici toutes les informations pour inférer un registre codé sur plusieurs bits. Afin de complexifier l'exemple, nous avons également adjoint la notion de *reset asynchrone* au code du processus. Asynchrone signifie que les assignations afférentes sont *prioritaires* sur l'horloge et indépendante d'elle. Ceci est parfaitement traduit, de manière intuitive, dans VHDL,

par le recours à la construction *if...elsif...*.

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

entity dff is
  generic (NB_BITS : natural := 8);
  port(
    reset_n : in  std_logic;
    clk     : in  std_logic;
    sreset  : in  std_logic;
    input   : in  std_logic_vector(NB_BITS-1 downto 0);
    output  : out std_logic_vector(NB_BITS-1 downto 0)
  );
end entity;

architecture using_rising_edge of dff is
begin

  -- only means to describe registers : use processes !
  process(reset_n, clk)  -- only async reset & clock are necessary here
  begin
    if reset_n = '0' then                --active low
      output <= (others => '0');         -- all bits to '0'
    elsif rising_edge(clk) then          --or : clk'event and clk='1'
      output <= input;
    end if;
  end process;

end using_rising_edge;
```

Registre générique de plusieurs bits, resets synchrone et asynchrone Il est fréquent que la condition de reset ne soit pas asynchrone, mais provienne d'une autre partie (synchrone) du design. Dans ce cas, la possibilité de réinitialiser la bascule est une *feature* du système à concevoir.

```
architecture with_synchronous_reset of dff is
begin
  process(reset_n, clk)
  begin
    if reset_n = '0' then                --active low
      output <= (others => '0');
    elsif rising_edge(clk) then
      if sreset = '1' then
        output <= input;
      end if;
    end if;
  end process;
end with_synchronous_reset;
```

Utilisation des *wait until* Il est également possible d'inférer une bascule D à l'aide de la construction VHDL *wait until*, mais nous préférons l'éviter dans ce cours. Un exemple est donné dans le listing suivant.

```
architecture using_wait_until of dff is
begin
  process  -- No sensitivity list
  begin
    wait until rising_edge(clk);
    output <= input;
  end process;
end using_wait_until;
```

2.2 Compteurs

Inférence à parti d'un seul *process* séquentiel Armé de la bascule D, nous souhaitons désormais rapidement concevoir des circuits plus significatifs en terme de fonctionnalité. Sans brûler les étapes, nous présentons un compteur qui s'incrémente et se décrémente. Si nous ne disposions pas de VHDL,

mais simplement d'un papier et d'un crayon, nous décriverions ce compteur par un registre de plusieurs bits, précédé de multiplexeurs qui, selon la condition *up* ou *down* font *recirculer* la valeur *précédente* du compteur incrémentée ou décrémentée de 1. C'est déjà, sur papier un réseau de fonctions logiques interconnectées remarquablement complexe.

La chose intéressante à observer est que le codage VHDL, typique du niveau RTL, va rendre cette *capture* bien plus simple que la description explicite d'un tel réseau : il s'agit de l'exemple le plus illustratif de la notion d'*inférence*, essentielle à la bonne compréhension de la synthèse VHDL (ou Verilog). A titre d'anecdote, on estime que l'Industrie de l'Electronique au Japon a réalisé une erreur colossale en ne faisant pas le choix initial des langages HDL, au profit d'outils de capture graphique : recourir à de tels outils de saisie graphique est une fausse bonne idée : la portabilité des design est rendue très dépendente des outils.

```
architecture rtl of counter is
  signal value : signed(NB_BITS-1 downto 0);
begin

  process(reset_n, clk)
  begin
    if reset_n = '0' then
      value <= to_signed(0, NB_BITS);
    elsif rising_edge(clk) then
      if cnt_up = '1' then
        value <= value + 1;
      elsif cnt_down = '1' then
        value <= value - 1;
      end if;
    end if;
  end process;

end rtl;
```

Notons enfin que la valeur du compteur *value* ne peut être directement émise vers la sortie, ce qui peut paraître contre-intuitif. La raison en est simple : on ne peut relire une sortie ; or le compteur doit effectivement faire re-circuler cette valeur.

Décomposition séquentiel-combinatoire Pour des cas plus complexes que les compteurs, le fait de mélanger, dans un seul processus *clocké*, des descriptions qui conduiront à des éléments séquentiels **et** combinatoires peut être délicat. Parfois, les concepteurs préfèrent séparer explicitement les parties purement séquentielles et combinatoires : dans le cas du compteur, cela conduit à deux processus, comme exposé dans le code suivant.

```

architecture rtl of counter is
  signal value_r,value_c : signed(NB_BITS-1 downto 0);
begin

  sequential : process(reset_n, clk)
  begin
    if reset_n = '0' then
      value_r <= to_signed(0, NB_BITS);
    elsif rising_edge(clk) then
      value_r <= value_c;
    end if;
  end process;

  combinatorial : process(cnt_up,cnt_down,value_r)
  begin
    if cnt_up = '1' then
      value_c <= value_r + 1;
    elsif cnt_down = '1' then
      value_c <= value_r - 1;
    else
      value_c <= value_r;
    end if;
  end process;

end rtl;
```

Pour procéder à la transformation, nous avons soigneusement appelé *value_r* la sortie du registre, tandis que l'entrée du registre s'appelle désormais *value_c*, issue de la logique combinatoire. Un second

processus a été ajouté : à première vue, il ressemble beaucoup au code initial. Toutefois, on a ajouté une dernière clause *else* à l'ensemble des *if*; la raison est importante : dans le cas où l'on aurait omis cette partie du code, le synthétiseur aurait automatiquement inféré un élément étranger : un latch transparent. Par défaut, le synthétiseur tente de refaire circuler non pas la valeur issue du registre, mais la valeur *value_c* combinatoire ! Il s'agit d'un cas d'inférence de *boucle combinatoire*, qui on le rappelle, sont proscrites des conceptions synchrones. Ce choix délibéré du synthétiseur peut paraître étonnant, mais c'est ainsi : il serait compliqué de décider de refaire circuler une autre valeur : pourquoi par exemple la valeur du registre, plutôt qu'une valeur par défaut ? Et si jamais aucun registre n'avait été décrit, etc. La leçon à tirer est de bien vérifier la complétude des assignations dans un processus combinatoire.

2.3 Conversions entre types

L'exemple du compteur précédent nous donne également l'opportunité d'introduire la notion de conversions entre type. Dans l'exemple précédent, la valeur *value* et la sortie *output* n'ont pas été typées de la même manière, par le concepteur. C'est un choix délibéré de sa part : il aurait très bien pu utiliser un type *unsigned* pour la sortie également. L'énorme majorité des types sont synthétisables, y compris les types définis par l'utilisateur comme les *record* (nous y reviendrons). La ligne 35 force donc une conversion entre le type *unsigned* et le type *std_logic_vector*.

Ces conversions entre type sont très contraignantes : VHDL est fortement typé et veille scrupuleusement au respect des types, pour plus de sécurité (lors des traductions notamment). Fort heureusement, il existe un petit ensemble de principe de conversions simples qui rend la tâche aisée. Ce petit ensemble a été parfaitement illustré par la société Doulos, à qui nous empruntons le schéma 2.1 suivant.

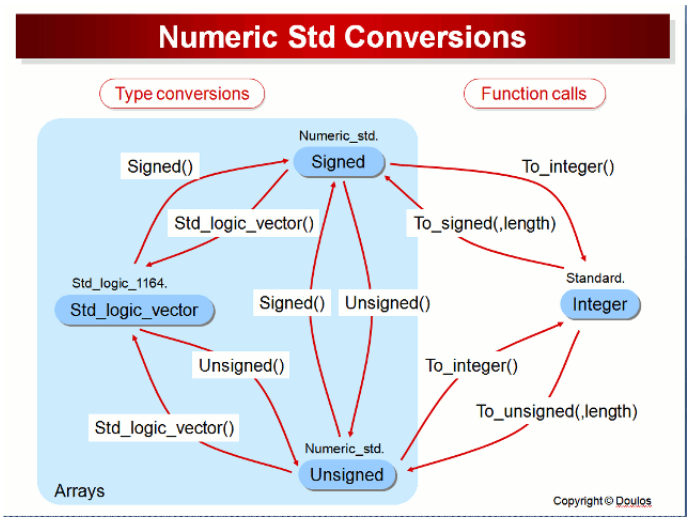


FIGURE 2.1 – Conversions usuelles dans la bibliothèque *numeric_std* [Doulos]

2.4 Multiplexeurs

Parmi les éléments matériels essentiels à la conception RTL figurent les multiplexeurs. Pour rappel, ces circuits permettent de faire le choix entre plusieurs entrées et acheminer l'une d'entre elles sur une sortie unique. Ce choix se fait par une entrée dédiée ou *commande* : généralement, pour un nombre n d'entrées, le signal de commande porte simplement le numéro de l'entrée qui doit être sélectionnée. On parle de multiplexeurs *n-to-1*. Il existe plusieurs manière d'écrire de tels multiplexeurs.

Utilisation des *if* statements Comme nous l'avons déjà vu pour le compteur, le *if* de VHDL conduit naturellement à un multiplexeur : dans le corps du *if* il faut observer quel signal est assigné ; il possédera telle ou telle valeurs, en fonction de la condition du *if*. Cette manière de procéder ne peut se retrouver que dans un processus (combinatoire ou non). Dans le cas combinatoire, on rappelle l'importance de vérifier la complétude des assignations.

Utilisation des *when* statements En dehors des processus, il est possible de décrire également des multiplexeurs :

Utilisation des *select* statements

Autres manière d'inférer des multiplexeurs

2.5 Instanciation de composants

L'instanciation de composants est similaire, dans la démarche, à l'instanciation de Classes en languages orientés objets (Ruby, Python, Java, C++ etc) : dans notre cas, une entité déclarée reste l'équivalent de la classe : une sorte de modèle dont on fera usage si nécessaire. On instancie cette entité de deux manières.

Instanciation de *components* La plus ancienne des manières de procéder consiste à déclarer, au sein d'une architecture (ou d'un package) un **component** ayant le même nom et la même interface que l'entité. Cela autorise à effectivement utiliser le composant instancié. Cette manière de procéder est en perte de vitesse dans la communauté. La seconde

Instanciation d'*entity* La seconde manière de procéder est plus directe. Il suffit d'appeler l'entité elle-même. La seule contrainte est de devoir rappeler la librairie VHDL où a été compilée cette entité. Il n'y a pas besoin de rappeler une quelconque définition de "component". C'est désormais un style recommandé.

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

entity system_comp is
  port(
    reset_n               : in std_logic;
    clk                   : in std_logic;
    cnt_up_1, cnt_down_1  : in std_logic;
    cnt_up_2, cnt_down_2  : in std_logic
   -- warn : no output? => nothing synthesized !!!
    );
end system_comp;

architecture rtl of system_comp is

  -- we recall here what a counter is...as a component...
  component counter is
    generic (
      NB_BITS : natural);
    port (
      reset_n          : in  std_logic;
      clk              : in  std_logic;
      cnt_up, cnt_down : in  std_logic;
      output           : out std_logic_vector(31 downto 0));
  end component counter;

  signal cnt_up, cnt_down : std_logic;
  signal output           : std_logic_vector(31 downto 0);

begin

  -- then we instanciate the component...
  counter_1 : counter
    generic map (
      NB_BITS => 32)
    port map (
      reset_n  => reset_n,
      clk      => clk,
      cnt_up   => cnt_up,
      cnt_down => cnt_down,
      output   => output);

```

```
end rtl;
```

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

entity system is
  port(
    reset_n              : in std_logic;
    clk                  : in std_logic;
    cnt_up_1, cnt_down_1 : in std_logic;
    cnt_up_2, cnt_down_2 : in std_logic
   -- warn : no output? => nothing synthesized !!!
    );
end entity;

-- entities instanciation : shorter way !
architecture rtl of system is

  signal output_1 : std_logic_vector(NB_BITS-1 downto 0);
  signal output_2 : std_logic_vector(15 downto 0);  --16 bits
  signal output_1 : std_logic_vector(7 downto 0);
  signal output_1 : std_logic_vector(7 downto 0);

begin

  --classical way. explicit
  counter_1 : entity work.counter(rtl)
    generic map(NB_BITS => 10)          -- no ';'
    port map(
      reset_n  => reset_n,
      clk      => clk,
      cnt_up   => cnt_up_1,
      cnt_down => cnt_down_1,
      output   => output_1);

  counter_2 : entity work.counter(rtl)
    generic map(16)
    port map(
      reset_n  => reset_n,
      clk      => clk,
      cnt_up   => cnt_up_2,
      cnt_down => cnt_down_2,
      output   => output_2);

  --default generic to 8 (see counter.vhd)
  counter_3 : entity work.counter(rtl)
    port map(
      reset_n  => reset_n,
      clk      => clk,
      cnt_up   => cnt_up_1,
      cnt_down => cnt_down_1,
      output   => output_3);

  -- labels are optional
  entity work.counter(rtl)
    port map(
      reset_n  => reset_n,
      clk      => clk,
      cnt_up   => cnt_up_2,
      cnt_down => cnt_down_2,
      output   => output_4);

end rtl;
```

2.6 Mémoires synthétisables

Lors d'une conception, il est généralement utile de pouvoir stocker des données dans une *mémoire adressable*. Il est tout à fait possible de décrire de telles mémoires en VHDL et de les synthétiser. Un exemple est donné ici.

Le problème des mémoires Le synthétiseur va simplement agglomérer un ensemble (généralement grand !) de bascules initialement "dispersées" afin de donner l'illusion de l'existence d'un composant "mémoire" bien constitué. Ceci n'est généralement pas très efficace et il est donc nécessaire d'envisager une autre solution. En fait, il en existe 3 : la première des solutions de repli est d'écrire un code VHDL particulier qui permettra au synthétiseur de reconnaître une ressource plus efficace : le block-ram. C'est une sorte d'accord tacite entre le synthétiseur et vous : si vous écrivez d'une certaine manière, la synthèse conduira aux block-rams, sinon ce sera des registres distribués...

```
-------------------------------------------------------------------
-- RAM with asynchronous read => distributed RAM (based on DFF)
-- only recommended for small memories
-------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ram is
  generic (WORD_SIZE : natural := 8;
           ADDR_SIZE : natural := 10);
  port(
    reset_n : in  std_logic;
    clk     : in  std_logic;
    we      : in  std_logic;
    address : in  std_logic_vector(ADDR_SIZE-1 downto 0);
    datain  : in  std_logic_vector(WORD_SIZE-1 downto 0);
    dataout : out std_logic_vector(WORD_SIZE-1 downto 0)
    );
end entity;

architecture rtl of ram is
  type mem_type is array(0 to 2**ADDR_SIZE-1) of std_logic_vector(WORD_SIZE-1 downto
      0);
  signal mem : mem_type;
begin

  ram_p : process(reset_n, clk)
  begin
    if rising_edge(clk) then
      if we = '1' then
        mem(to_integer(unsigned(address))) <= datain;
      end if;
    end if;
  end process;

  dataout <= mem(to_integer(unsigned(address)));  --asynchronous read

end rtl;
```

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

entity ram is
  generic (WORD_SIZE : natural := 8;
           ADDR_SIZE : natural := 10);
  port(
    reset_n : in  std_logic;
    clk     : in  std_logic;
    we      : in  std_logic;
    address : in  std_logic_vector(ADDR_SIZE-1 downto 0);
    datain  : in  std_logic_vector(WORD_SIZE-1 downto 0);
    dataout : out std_logic_vector(WORD_SIZE-1 downto 0)
    );
end entity;

architecture rtl of ram is
  type mem_type is array(0 to 2**ADDR_SIZE-1) of std_logic_vector(WORD_SIZE-1 downto
      0);
  signal mem : mem_type;
  signal addr_r : std_logic_vector(ADDR_SIZE-1 downto 0);
begin

  ram_p : process(reset_n, clk)
```

```
25    begin
26      if rising_edge(clk) then
27        if we = '1' then
28          mem(to_integer(unsigned(address))) <= datain;
29        end if;
30        addr_r <= address;
31      end if;
32    end process;
33
34    dataout <= mem(to_integer(unsigned(addr_r))); --SYNCHRONOUS read
35
36  end rtl;
```

Chapitre 3

Machines d'états finis

3.1 Introduction

Les machines d'états finis (indifféremment appelées *automates* OU FSM, *finite state machines* ici) sont essentielles aux systèmes embarqués : elles assurent à la fois la notion de séquencement ainsi que les tâches de contrôle de dispositifs. Dans un premier temps, nous reviendrons sur des rappels concernant ces automates. Par la suite, nous proposerons un codage spécifique en VHDL : une des difficultés des HDL et –selon moi– une de leur faiblesse réside dans l'*absence de mots clés* réservés à la description de telles machines d'états finis. On palie cette absence par un codage rigoureux et systématique de ces notions, à l'aide des concepts du langage (signaux et processus).

3.2 Rappels

Il y a lieu de distinguer deux types d'automates : Moore et Mealy. Très proches conceptuellement, il est toutefois nécessaires de bien comprendre leurs différence pour l'Ingénieur Numéricien. Nous verrons notamment que le couplage de tels automates peut-être source de tracasseries délicates.

Automate de Moore Une automate de Moore est un sextuplet $(Q, \Sigma, \Delta, \sigma, \lambda, q_0)$:

— Q est un ensemble fini d'états, q_0 est l'état initial
— Σ est l'alphabet d'entrée, Δ est l'alphabet de sortie
— δ est une application de $Q \times \Sigma$ dans Q
— λ est une application de Q dans Δ, donnant la sortie associée à chaque état

La sortie de l'automate de Moore en réponse à une entrée $a_1 a_2 \dots a_n$, $n \geq 0$ est $\lambda(q_0), \lambda(q_1) \dots \lambda(q_n)$ où $q_0, \dots, q_n$ est la séquence d'états tels que $\lambda(q_{i-1}, a_i) = q_i$ pour $1 \leq i \leq n$. Remarque : Un automate de Moore retourne la sortie $\lambda(q_0)$ pour toute entrée.

Automate de Mealy Une automate de Mealy est un sextuplet $(Q, \Sigma, \Delta, \sigma, \lambda, q_0)$:

— Q est un ensemble fini d'états, q_0 est l'état initial
— Σ est l'alphabet d'entrée, Δ est l'alphabet de sortie
— δ est une application de $Q \times \Sigma$ dans Q
— λ est une application de $Q \times \Sigma$ dans Δ, donnant la sortie associée à chaque état

$_l ambda(q, a)$ donne la sortie associée à une transition d'un état q sur l'entrée a. La sortie de l'automate de Mealy,en réponse à une séquence d'entrées $a_1, \dots a_n$ est $\lambda(q_0, a_1)\lambda(q_1, a_2) \dots \lambda(q_{n-1}, a_n)$ où $q_0, q_1, \dots, q_n$ est la séquence des états tels que $\lambda(q_{i-1}, a_i) = q_i$ pour $1 \leq i \leq n$.

Comparaison Les deux définitions sont très proches l'une de l'autre. On doit seulement comprendre que dans le cas d'une machine de Mealy, les sorties dépendent des entrées et de l'état courant, alors que dans le cas de la machine de Moore, ces sorties dépendent uniquement de l'état courant. En règle générale, les machines de Mealy sont plus rapides : leur chemin critique est plus court. Par contre, elles sont souvent proscrites des bonnes règles de conception –en vigueur dans la plupart des sociétés– car elles peuvent induire des bugs difficiles à localiser. En effet, l'interconnexion de plusieurs automates de Mealy peut présenter des cycles combinatoires. On rappelle que ces cycles (ou boucles) combinatoires sont interdites en logique synchrone car elle ne permettent pas de déterminer la fréquence propre de

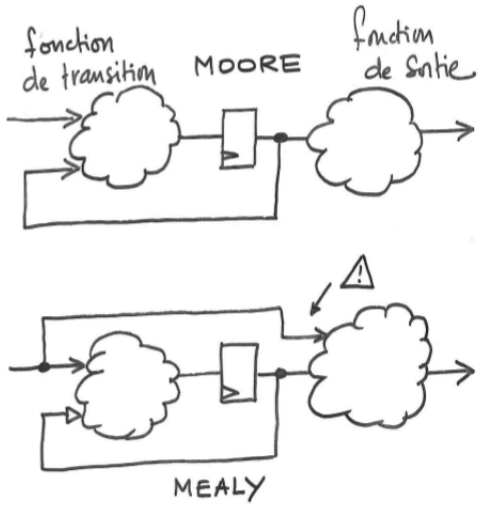


FIGURE 3.1 – Automates de Moore et de Mealy

l'horloge du circuit.

En règle générale, toutefois, on plus code naturellement avec des machines de Mealy. La seule chose à prendre en compte est de bien clore le chemin des sorties combinatoires par un registre adéquat. Cela fait partie des bonnes règles applicables par ailleurs : les entrées et les sorties d'un circuit un tant soit peu complexe doivent être "clockées", c'est-à-dire échantillonnées dans des registres.

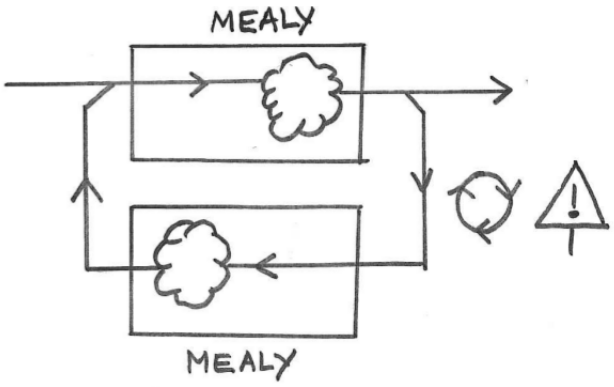


FIGURE 3.2 – Composition d'automates de Mealy et risque de cycles combinatoires

3.3 Diagramme état-transitions

Le diagramme état-transitions (aussi souvent appelé diagramme à bulles) est la forme graphique la plus pratique pour représenter à la fois les états et les transitions associées à ces états.

3.4 Principes du codage proposé

A première vue, il peut paraître tentant de réaliser ces circuits de contrôle et de séquencement sans réelle structuration du code sous forme d'automate. C'est là une pratique dangereuse et rapidement non-viable.

Nous proposons ici un codage systématique des FSM : ceci permet de palier l'absence dans la syntaxe du langage de ces notions pourtant essentielles à la conception numérique. Ce codage peut être décrit par les points suivants :

— On crée deux processus séquentiels, visant à capturer d'une part l'état de la FSM, et d'autre part les registres de travail (contenant notamment les variables applicatives, ou autres signaux utiles éventuels).
  — Tous ces signaux sont déclarés avec un suffix "_r", pour indiquer leur nature séquentielle.
  — On leur affecte leur homologue combinatoire suffixé par "_c". C'est le fil d'entrée (combinatoire) du registre.
— L'ensemble de la logique combinatoire est regroupée en un seul processus (un label adéquat peut être utilisé comme "comb"...).
— Au sein de cette logique combinatoire, nous basculons sur des variables de VHDL,toutes suffixées par "_v" :
  — En début de processus, les variables sont affectées par les signaux "_r"
  — Au cours du processus, Le traitement combinatoire est effectué sur les seules variables.
  — En fin de processus, on réaffecte les signaux combinatoire "_c" avec les variables "_v"

3.5 Exemple du PGCD

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

entity pgcd_fsmd is
  port(
    reset_n : in  std_logic;
    clk     : in  std_logic;
    go      : in  std_logic;
    a_valid : in  std_logic;
    b_valid : in  std_logic;
    a, b    : in  unsigned(31 downto 0);
    result  : out unsigned(31 downto 0);
    done    : out std_logic
    );
end entity;

architecture rtl of pgcd_fsmd is

  type state_type is (IDLE, WAITING_A, WAITING_B, COMPUTING);
  signal state_r, state_c : state_type;

  signal a_r, a_c           : unsigned(31 downto 0);
  signal b_r, b_c           : unsigned(31 downto 0);
  signal result_r, result_c : unsigned(31 downto 0);
  signal done_r, done_c     : std_logic;

begin

  tick : process(reset_n, clk)
  begin
    if reset_n = '0' then
      state_r <= IDLE;
    elsif rising_edge(clk) then
      state_r <= state_c;
    end if;
  end process;

  comb : process(a, a_r, a_valid, b, b_r, b_valid, go, state_r)
    variable a_v      : unsigned(31 downto 0);
    variable b_v      : unsigned(31 downto 0);
    variable result_v : unsigned(31 downto 0);
    variable done_v   : std_logic;
    variable state_v  : state_type;

  begin
    -- default variable assignments
    state_v  := state_r;
    a_v      := a_r;
    b_v      := b_r;
    result_v := to_unsigned(0,32);
    done_v   := '0';

```

```
54      -- algorithm with software-like variables
55      case state_v is
56        when IDLE =>
57          if go = '1' then
58            state_v := WAITING_A;
59          end if;
60        when WAITING_A =>
61          if a_valid = '1' then
62            a_v     := a;
63            state_v := WAITING_B;
64          end if;
65        when WAITING_B =>
66          if b_valid = '1' then
67            b_v     := b;
68            state_v := COMPUTING;
69          end if;
70        when COMPUTING =>
71          if a_v = b_v then
72            result_v := a_v;
73            done_v   := '1';
74            state_v  := IDLE;
75          else
76            if a_v > b_v then
77              a_v := a_v-b_v;
78            else
79              b_v := b_v-a_v;
80            end if;
81          end if;
82        when others=>
83          null;
84      end case;
85
86      -- combinatorial signal assignments
87      a_c      <= a_v;
88      b_c      <= b_v;
89      result_c <= result_v;
90      done_c   <= done_v;
91      state_c  <= state_v;
92    end process;
93
94    -- (sequential) variables assignment (storage)
95    vars : process(reset_n, clk)
96    begin
97      if reset_n = '0' then
98        a_r      <= to_unsigned(0, 32);
99        b_r      <= to_unsigned(0, 32);
100       result_r <= to_unsigned(0, 32);
101       done_r   <= '0';
102     elsif rising_edge(clk) then
103       a_r      <= a_c;
104       b_r      <= b_c;
105       result_r <= result_c;
106       done_r   <= done_c;
107     end if;
108   end process;
109
110   result <= result_r;
111
112 end rtl;
```

3.6 FSMD : considération sur le chemin de données

Nous nous devons ici de signaler que les Electroniciens numériques et les architectes des ordinateurs tiennent à distinguer deux formes de FSMD :

FSMD implicite Les FSMD implicite correspondent à la manière la plus naturelle de décrire le matériel. C'est ce que nous avons fait lors de l'exemple précédent du PGCD : le code de séquencement et le code de calcul sont entremélés. On peut le voir de manière positive : ce code compact est plus explicite. Le cheminement du flot de contrôle permet également de bien suivre les calculs afférents, pour

chaque état ou chaque transition. On parle de FSMD implicite car la partie Datapath est effectivement noyée dans une seule et même description.

FSMD explicite A l'inverse, on peut chercher à isoler la partie séquencement (et contrôle) de la partie purement calculatoire : les additionneurs, multiplieurs etc ainsi que les ressources de multiplexages peuvent être explicitement séparées dans la description. Le rôle de la partie FSM (sans le 'D') est alors uniquement de réaliser le flot de contrôle et de lancer les ordres à la partie Datapath. En retour, le Datapath envoie les *signaux de status* qui permette à la FSM de réaliser ses transitions. Ces signaux d'échange sont uniquement binaires. Cette séparation FSM et Datapath conduit naturellement à la *microprogrammation*, où le contrôleur pilote chaque ressource de manière précise. Cette manière d'envisager le Système est due à Maurice Wilkes, un pionner des machines informatiques.

Quel style adopter ? Il n'y a pas de réponse toute faite. Toutefois, les capacités du langage, son expressivité et le mécanisme d'inférence poussent à recourir de manière intensive à des FSMD Implicites. La décomposition du contrôle et des calculs possède par contre un avantage majeur : en cas de nécessité de partager des opérateurs gourmands en surface, les FSMD explicites sont indispensables. En général, en effet, les synthétiseurs ne sont guère efficaces pour réaliser eux-mêmes ces partages de ressources.

3.7 Machines d'états finis hiérarchique et concurrentes : Statecharts

Un certain nombre de scientifiques (dont David Harel) et ingénieurs ont tenté de généraliser la notion d'automate d'états finis en leur adjoignant deux nouvelles caractérisitiques intéressante pour la capture de comportements complexes :

- La **concurrence** : il s'agit de la possibilité de décrire des automates qui évolueront en parallèle.
- La **hiérarchie** : il s'agit de la possibilité de décrire un automate complet au sein d'un ou plusieurs états particuliers.

Cette nouvelle forme d'automates s'appele des *statecharts*.

3.8 Machine d'états finis...en logiciel

Il peut être intéressant de s'interresser à la manière de coder un tel automate à l'aide d'un langage informatique traditionnel. C'est loin d'être un exercice purement récréatif : la majeure partie des logiciels dans les systèmes critiques (aéronautique, ferroviaire, nucléaire, etc) sont écrits ou générés sous cette forme. Dans la pratique, il existe deux grandes manières de coder de tels automates : la "boucle réactive" et la "pattern Etat".

Boucle réactive Par "boucle réactive" on entend un code organisé autour d'une boucle infinie, au sein de laquelle on sélectionne l'état courant grâce à un *switch*. Un code simple est donnée ci-dessous.

```
#include <stdio.h>
#include <unistd.h>

enum states {PING, PONG};

enum states state = PING;

void led_on(){
  printf("␣led␣ON\n");
}

void led_off(){
  printf("␣led␣OFF\n");
}

int main(){

  while(1) {
    switch(state){
      case PING:
        led_on();
```

```
22          state = PONG;
23          break;
24        case PONG:
25          led_off();
26          state = PING;
27          break;
28      }
29    }
30
31    return 0;
32  }
```

Nous laissons en annexe d'autres exemples de code C plus évolués.

Pattern Etat

3.9 Conclusion

Les automates d'états finis permettent de coder le séquencement et le contrôle des dispositifs. C'est une représentation à la fois étudiée par les théoriciens (informaticiens, automaticiens et électroniciens) et les ingénieurs. Ils constituent un point de rendez-vous interdisciplinaire important. Le chapitre nous a permis de proposer un schéma d'encodage de tels automates, en VHDL.

Chapitre 4

Codage avancé

VHDL possède des constructions syntaxiques puissantes, qui permettent au concepteur d'élaborer des structures RTL complexes. Ces constructions nécessitent une compréhension globale des limites de synthétisabilité des outils EDA. La compréhension des mécanismes de base survolés dans les chapitres précédents sont un prérequis à la lecture de ce chapitre, plus avancé. Nous recensons ici quelques unes de ces constructions, ainsi que quelques pratiques d'usage dans le domaine.

4.1 Instructions *for ... generate*

VHDL permet de décrire des *structures génériques*. Les structures génériques correspondent à du code qui se déroule non pas dans le temps, mais dans l'espace, c'est-à-dire sur la surface de la puce, comme une structure mécanique qui se déploirerait. Il est difficile de trouver immédiatement un équivalent dans les langages de programmation traditionnels : cela se situe probablement entre de la métaprogrammation et des approches par macros : cela permet demander au compilateur (ici le synthétiseur) d'écrire du code pour vous. Ces manières de procéder en Electronique sont par contre très naturelles : par exemple, nous savons que certaines structures de calculs sont répétitives spatiallement : nous donnons ici l'exemple basique d'un simple additionneur, qui se construit à partir d'additionneurs 1 bits. Quel que soit la cardinalité de l'additionneur final, il est possible de décrire cette réutilisation à l'aide du *for ... generate*.

On commence par rappeler la constitution d'un *additionneur 1 bit* (pour rappel il est possible de composer cet additionneur à partir d'un *demi-additionneur*, ce qui n'est pas fait ici).

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

entity full_adder_1 is
  port(
    a, b, cin : in  std_logic;
    f         : out std_logic;
    cout      : out std_logic
    );
end entity;

architecture rtl of full_adder_1 is
begin
  f    <= a xor b xor cin;
  cout <= (a and b) or (cin and (a xor b));
end rtl;
```

Nous passons maintenant à l'utilisation du *for ... generate*.

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

entity adder is
  generic (NB_BITS : natural := 8);
  port(
    a, b  : in  std_logic_vector(NB_BITS-1 downto 0);
    c     : out std_logic_vector(NB_BITS-1 downto 0);
    carry : out std_logic
```

```
    );
end entity;

architecture rtl of adder is
  signal f,cin : std_logic_vector(NB_BITS-1 downto 0);
begin

  cin(0) <= '0'; --entry carry fixed to '0'

  GEN: for i in 0 to NB_BITS-1 generate
    inst_i : entity work.full_adder_1(RTL)
      port map(
        a    => a(i),
        b    => b(i),
        cin  => cin(i),
        f    => f(i),
        cout => cin(i+1));
  end generate;

  c     <= f;
  carry <= cin(NB_BITS-1);

end rtl;
```

Dans cet exemple, on voit que l'on doit être méticuleux en ce qui concerne les indices des signaux à connecter. Dans des exemples plus complexes, cela est encore plus vrai : le jeu consiste à préparer soigneusement le "déroulement spatial" de la boucle. Des exemples beaucoup plus complexes peuvent être imaginés : par exemple des structures non plus 1D, mais 2D voire plus : il suffit alors d'avoir de telles boucles imbriquées. Là encore, un concepteur novice peut être impressionné par de telles constructions et douter de leur synthétisabilité. Ces structures bidimensionnelles sont très fréquentes : on les retrouve par exemple dans la description de certains multiplications sur silicium, ou des réseaux réguliers de calculateurs identiques : calcul systolique, FPGA virtuels, matrice d'ALU ou de processeurs,etc.

```
band : for I in 1 to 10 generate
  b2 :      for J in 1 to 11 generate
  b3 :         if abs(I-J)<2 generate
                   part: foo port map ( a(I), b(2*J-1), c(I, J) );
               end generate b3;
           end generate b2;
        end generate band;
```

Le *for ... generate* peut s'utiliser dans une telle interconnexion de composants de base, dans le but de construire une structure plus large, mais également permet de composer plusieurs processus ou assignations concurrentes.

4.2 Cas des boucles *for* et *while* synthétisables

Notre style de codage VHDL retenu jusqu'ici se voulait descriptif : le concepteur visualise un système composé de parties logiques et de partie combinatoire et les code de manière appropriée. Le grand absent de cette stratégie est la notion de boucle "traditionnelle". Effectivement, en première approche, les boucles *for* et *while* sont à proscrire. Il existe toutefois des cas où le synthétiseur RTL est capable d'inférer un matériel (combinatoire) respectant la sémantique de la boucle : notamment lorsque la boucle peut être *déroulée* (statiquement) par le compilateur. Un exemple illustratif a trait à l'initialisation de registres.

```
architecture rtl of circuit is
begin
  registers_p:process(reset_n,clk)
    if reset_n='0' then
      for i in 0 to 255 loop
        reg(i) <= to_unsigned(0,8);
      end loop;
    elsif rising_edge(clk) then
      -- etc....
    end if;
  end process;
end rtl;
```

4.3 Paramétrisation à *compile time*

Une pratique fréquence en VHDL est de s'appuyer sur une calcul préalable, réalisé par le synthétiseur, lors de la paramétrisation de certains signaux, types ou composants. Ce calcul abouti à un résultat (scalaire voire vectoriel) : par exemple une valeur entière calculée grâce à une formule complexe faisant appel à des librairies VHDL. Ce calcul ne se retrouvera pas implémenté in-situ, mais peut être vu comme une aide à la conception. Des fonctions comme *log,log2, etc* sur des flottants, ramenés dans un domaine synthétisable (signed, unsigned etc) sont donc parfaitement utilisables lors de cette paramétrisation, y compris dans un composant synthétisable. Un principe de base consiste à se restreindre à une telle utilisation uniquement lors des déclarations d'architecture.

Nous illustrons ici un calcul d'exponentiation. En toute logique, une exponentiation correspond à une multiplication itérée, coûteuse en matériel. Toutefois, comme ce calcul peut être réalisé par le synthétiseur à *compile-time* (ou *design-time*, ce qui est équivalent), ce calcul coûteux n'apparaîtra aucunement dans le circuit réalisé : c'est une simple valeur, constante précalculée.

```
-- in testbench
constant bits    : integer := 13;

-- in architecture
constant pow     : integer := 2**bits;
constant squared : integer := pow**2;
```

4.4 Utilisation des *packages*

Les *packages* permettent de regrouper des définitions utiles et partagées à travers différents autres fichiers VHDL : il peut s'agir de définitions de constantes utiles, mais également de types, procédures et fonctions. L'exemple qui suit présente une définition de constante, ainsi que la définition d'une fonction *log2* très utile en conception.

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

package util_pkg is

  constant magic_number : unsigned(31 downto 0) := x"DEADBEEF";  -- hexa !
  function log2(x           : natural) return natural;
  function floor(x : natural) return natural;

end package;

package body util_pkg is

  function log2(x : natural) return natural is
    variable i : natural;
  begin
    i := 0;
    while x > 2**i loop
      i := i+1;
    end loop;
    return i;
  end function;


  function floor(x : natural) return natural is
  begin
    return integer(floor(real(x)));
  end function;

end package body;
```

L'exemple qui suit permet de rendre concrète l'utilisation d'un tel package : dans l'en-tête, on rappelle l'existence de ce package (compilé au préalable). Dès lors, les élements du package sont connus et directement utilisables.

```
library ieee;
```

```
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.my_package.all;

library std;
use std.textio.all;

entity test_log2 is
end test_log2;

architecture bhv of test_log2 is
begin
  process
  begin
    report "log2(13)     = " & integer'image(log2(13));
    report "log2(1023)   = " & integer'image(log2(1023));
    wait;
  end process;

  process is
  begin
    report "magic number = " & to_string(std_logic_vector(magic_number));
    wait;
  end process;

end bhv;
```

4.5 Utilisation des *record*s

Comme tout langage de haut niveau, VHDL permet également de recourir aux types structurés. Ces définitions de types sont des **record**. Nous donnons ici un exemple de définition d'un type record dans une bibliothèque custom appelée "my_complex". Dans cette bibliothèque sont définis le type complex_16, ainsi qu'une première fonction opérant sur ce nouveau type. Il s'agit en l'occurence d'une simple addition de nombres complexe.

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

package my_complex is

  subtype int_16 is signed(15 downto 0);

  type complex_16 is record
    real : int_16;
    imag : int_16;
  end record;

  function "+"(x,y: complex_16) return complex_16;
end package;

package body my_complex is

  function "+"(x,y : complex_16) return complex_16 is
    variable tmp : complex_16;
  begin
    tmp.real := x.real + y.real;
    tmp.imag := x.imag + y.imag;
    return tmp;
  end function;

end package body;
```

L'exemple qui suit est un simple test montrant l'utilisation du package et du nouveau type complexe ainsi défini.

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

```

```
5  use work.my_complex.all;
6
7  entity complex_test is
8  end complex_test;
9
10 architecture bhv of complex_test is
11 begin
12   process
13   variable c1,c2,c3 : complex_16;
14   begin
15     report "testing complex_pkg";
16     c1:=(to_signed(3,16),to_signed(5,16));
17     c2:=(to_signed(4,16),to_signed(-1,16));
18     c3:=c1+c2;
19     report "C3=(" & integer'image(to_integer(c3.real)) & "," & integer'image(
           to_integer(c3.imag)) & ")";
20     wait;
21   end process;
22 end bhv;
```

4.6 Types non contraints

4.7 Tableaux indexés par des types énumérés

4.8 Compilation dans une *library* particulière

Par défaut, les fichiers VHDL (les unités de compilation qu'ils contiennent, en réalité) sont compilés dans une bibliothèque particulière, appellé **work**. Il est possible de préciser dans quelle bibliothèqe *library* on souhaite voir ces fichiers affectés. Par exemple, dans GHDL, l'option est simplement :

```
ghdl -a comp.vhd --work=MY_LIB
```

Deuxième partie

Modélisation comportementale et bancs de test

Chapitre 5

Bancs de tests ou *Testbenches*

5.1 Principe du test. Modèle de référence

L'industrie du semi-conducteur est exemplaire en matière de test, bien loin devant l'industrie du logiciel : les milliards d'artefacts manipulés, à des niveaux d'abstractions variés (RTL, logique, transistor, masque symbolique, masque physique, polygones, electro-chimie, aux comportements numériques ou analogiques) ont forcé les microélectroniciens à mettre en place des stratégies de vérifications systématiques et performants. Un des principes fondamentaux retenus est celui de la *remplacabilité* : lorsqu'un composant abstrait est remplacé par un composant concret (et vice versa), le test doit pouvoir établir la conservation de propriétés essentielles du composant, significatives au niveau d'abstraction retenu. Pour cela, un modèle de référence doit faire foi concernant ces propriétés : il assure le bon fonctionnement du circuit au regard d'une certaine abstraction.

Vérification de composants RTL Dans notre cas, nous cherchons à mettre en place des procédure de vérification, appelées ıbancs de test ou *testbenches* : il s'agit de l'analogie du laboratoire d'électronique, où l'on trouve divers types de générateurs de signaux, ainsi que des sondes de mesure et des oscilloscopes. On retrouvera l'ensemble de ces instruments, *virtualisés* dans l'environnement VHDL. Un tel banc de test na toutefois pas vocation à être implémenté sur circuit.

Afin de vérifier un circui décrit au niveau RTL, la mise en place de l'environnement de test passe fort heureusement par un niveau d'abstraction supérieur : le niveau *comportemental*. Cela signifie que les contraintes liées à la synthèse vont disparaître concernant ces "instruments virtuels". On pourra alors utiliser toute la puissance du langage, qui s'étend bien au delà du niveau RTL.

5.2 Générateurs d'horloges et de reset

Le testbench représente un "banc de test" : comme on l'a dit, c'est une *paillasse virtuelle* où l'on trouve toute sorte d'instruments et de générateurs. Parmi ceux ci, on doit disposer d'un générateur de reset et un générateur d'horloge. Concernant le reset, nous présentons un code très simple : on indique que le reset actif bas se relève au bout de 666 nanosecondes.

Concernant l'horloge elle-même, on fait osciller un signal VHDL entre 0 et 1, après une demi-période d'horloge. Cette oscillation numérique peut se faire éternellement, jusqu'à l'interruption plus ou moins "sauvage" de la personne qui a lancé le simulateur : cela peut-être un arrêt brutal grâce à "control-c" (sous Linux). Les simulateurs possèdent également des arguments passés en ligne de commande qui permettent d'indiquer la durée à simuler. Notre code propose une troisième manière de stopper la simulation : un signal appelé **runnin** *interne* au design dit si oui ou non on doit effectivement réaliser l'oscillation. Si le signal running est à false, l'horloge n'est pas générée : cela a une conséquence intéressante, puisque le simulateur à événement discret n'aura rapidement plus aucun événement à ordonnancer. Ce sera l'*arrêt par famine* (**starvation** en anglais). Le simulateur considère alors qu'il doit rendre la main. Le signal running est lui-même pilotable par un autre processus VHDL : généralement c'est celui qui injecte des stimuli sur les ports d'entrée du design à tester (DUT).

```
1 architecture bhv of mon_design is
2
```

```
constant HALF_PERIOD : time := 5 ns;
signal clk : std_logic := '0';

begin

  clk <= not(clk) after HALF_PERIOD when running else clk; --generateur

  reset_n <= '0', '1' after 666 ns;

  stimuli : process
   begin
     -- ....code des stimuli ....
     running <= false;
     wait;
   end
end
```

5.3 Lecture et écriture de fichiers

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

-- hexadecimal read is provided by Synopsys pseudo IEEE package.
use IEEE.std_logic_textio.all; -- hread, hwrite,...
-- to use it with GHDL, compile with --ieee=synopsys.
-- ghdl -a --ieee=synopsys lecture_fichier_hexa.vhd

library std;
use std.textio.all;

entity lecture_fichier_tb is
end entity;

architecture bhv of lecture_fichier_tb is
  constant HALF_PERIOD : time := 5 ns;
  signal clk           : std_logic := '1';
  signal reset_n       : std_logic;
  signal running       : boolean:=true;
  signal signal1, signal2 : unsigned(7 downto 0);
begin

  file_read_proc : process
      file F : text;
      variable L: line;
      variable status : file_open_status;
      variable nb_lines : natural := 0;
      variable value1, value2    : std_logic_vector(7 downto 0);
      variable v_SPACE : character;
    begin
      FILE_OPEN(status,F,"stim.txt",read_mode);
      if status/=open_ok then
        report "problem to open stimulus file stim.txt" severity error;
      else
        while not(ENDFILE(f)) loop
          wait until rising_edge(clk);
          nb_lines:=nb_lines+1;
          readline(F,l);
          --report "before value1: '" & l.all & "'";
          hread(l,value1);
          read(l,v_SPACE);
          hread(l,value2);
          signal1 <= unsigned(value1);
          signal2 <= unsigned(value2);
        end loop;
        report integer'image(nb_lines) & " lines read. Good.";
      end if;
    end process;

end bhv;
```

5.4 Vecteurs de tests et de vérification

Outre la lecture de fichiers de stimuli, il est souvent pratique d'embarquer les vecteurs de tests (ou *stimuli* directement dans le banc de test, accompagné des valeurs attendues. Il est alors possible de comparer ces valeurs attendues aux valeurs calculées par le composant en cours de test. Nous donnons ici l'exemple d'une ALU minimaliste, ainsi que son testbench. Le testbench calcule le nombre de succès et d'échecs lors des tests et les reporte au final.

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

use work.ual_pkg.all;

entity ual is
  port(
    a, b : in  std_logic_vector(15 downto 0);
    op   : in  opcode;
    res  : out std_logic_vector(15 downto 0)
    );
end entity;


architecture rtl of ual is
begin

  ual_proc : process(a, b, op)
  begin
    res <= (others=>'0');
    case op is
      when OP_ADD =>
        res <= std_logic_vector(signed(a) + signed(b));
      when OP_SUB =>
        res <= std_logic_vector(signed(a) - signed(b));
      when OP_MUL =>
        res <= std_logic_vector(resize(signed(a) * signed(b),16));
      when OP_AND =>
        res <= a and b;
      when OP_OR =>
        res <= a or b;
      when OP_XOR =>
        res <= a xor b;
      when OP_NOTA =>
        res <= not(a);
      when others =>
        null;
    end case;
  end process;

end rtl;
```

```
--------------------------------------------------------------------------------
-- This file was partially generated automatically by tb_gen Ruby utility
-- date : (d/m/y) 08/03/2018 14:14
-- Author : Jean-Christophe Le Lann - 2014
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.ual_pkg.all;
use work.print_pkg.all;

entity ual_tb is
end entity;

architecture bhv of ual_tb is

  constant HALF_PERIOD : time := 5 ns;

  signal clk     : std_logic := '0';
  signal reset_n : std_logic := '0';
  signal sreset  : std_logic := '0';
```

```
23    signal running : boolean   := true;
24
25    procedure wait_cycles(n : natural) is
26    begin
27      for i in 1 to n loop
28        wait until rising_edge(clk);
29      end loop;
30    end procedure;
31
32    signal a, b : std_logic_vector(15 downto 0);
33    signal op   : opcode;
34    signal res  : std_logic_vector(15 downto 0);
35
36    type stimulus is record
37      a, b : std_logic_vector(15 downto 0);
38      op   : opcode;
39      res  : std_logic_vector(15 downto 0);
40    end record;
41
42    type stimuli_type is array(integer range <>) of stimulus;
43
44    constant stimuli : stimuli_type := (
45      0  => (op => OP_ADD, a => x"0000", b => x"0000", res => x"0000"),
46      1  => (op => OP_ADD, a => x"0001", b => x"0001", res => x"0002"),
47      2  => (op => OP_ADD, a => x"0abc", b => x"0def", res => x"18ab"),
48      --
49      3  => (op => OP_SUB, a => x"ABCD", b => x"ABCC", res => x"0001"),
50      4  => (op => OP_SUB, a => x"AB00", b => x"CD00", res => x"DE00"),
51      --
52      5  => (op => OP_MUL, a => x"00AB", b => x"00b0", res => x"7590"),
53      6  => (op => OP_MUL, a => x"AB00", b => x"CD00", res => x"0000")--strong
            truncation !
54      );
55
56  begin
57    -------------------------------------------------------------------
58    -- clock and reset
59    -------------------------------------------------------------------
60    reset_n <= '0', '1' after 12 ns;
61
62    clk <= not(clk) after HALF_PERIOD when running else clk;
63
64    -------------------------------------------------------------------
65    -- Design Under Test
66    -------------------------------------------------------------------
67    dut : entity work.ual(rtl)
68      port map (
69        a   => a,
70        b   => b,
71        op  => op,
72        res => res);
73
74    -------------------------------------------------------------------
75    -- sequential stimuli
76    -------------------------------------------------------------------
77    stim : process
78      variable success, failure_nb : natural;
79      variable stim                : integer := -1;
80    begin
81      success    := 0;
82      failure_nb := 0;
83      a          <= (others=>'0');
84      b          <= (others=>'0');
85      op         <= OP_ADD;
86      report "running testbench for ual(rtl)";
87      report "waiting for asynchronous reset";
88      wait until reset_n = '1';
89      wait_cycles(10);
90      report "applying stimuli...";
91      for i in stimuli'range loop
92        stim := stim+1;
93        wait_cycles(1);
94        a    <= stimuli(i).a;
95        b    <= stimuli(i).b;
```

```
96        op   <= stimuli(i).op;
97        wait until falling_edge(clk);
98        if res /= stimuli(i).res then
99          failure_nb := failure_nb + 1;
100         report("ERROR : for stimuli number " & integer'image(stim) & " (" & hstr(a) &
                ",," & hstr(b) & "," & opcode'image(op) & ")");
101         report("        Expecting " & hstr(stimuli(i).res) & ". Got " & hstr(res) & ".
                ");
102       else
103         success := success+1;
104       end if;
105     end loop;
106     report "end of simulation";
107     report "number of success : " & integer'image(success);
108     report "number of failure : " & integer'image(failure_nb);
109     running <= false;
110     wait;
111   end process;
112
113 end bhv;
```

```
ual_tb.vhd:86:5:@0ms:(report note): running testbench for ual(rtl)
ual_tb.vhd:87:5:@0ms:(report note): waiting for asynchronous reset
ual_tb.vhd:90:5:@105ns:(report note): applying stimuli...
ual_tb.vhd:106:5:@180ns:(report note): end of simulation
ual_tb.vhd:107:5:@180ns:(report note): number of success : 7
ual_tb.vhd:108:5:@180ns:(report note): number of failure : 0
```

Chapitre 6

Packages utiles en simulation

6.1 Affichage des bits vectors

Les bibliothèques du langage VHDL sont relativement modestes concernant l'affichage : que ce soit dans un terminal ou dans un fichier, l'affichage nécessite de passer par des bibliothèques plus confidentielles, voire de se créer ses propres packages. Nous proposons ici un embryon d'un tel package.

```
1
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.numeric_std.all;
5
6  package print_pkg is
7
8    function str(sl : std_logic) return string;
9    function str(slv : std_logic_vector) return string;
10   function hstr(s : std_logic_vector) return string;
11
12 end package;
13
14
15 package body print_pkg is
16
17   function str(sl : std_logic) return string is
18     variable sl_str_v : string(1 to 3);  -- std_logic image with quotes around
19   begin
20     sl_str_v := std_logic'image(sl);
21     return "" & sl_str_v(2);             -- "" & character to get string
22   end function;
23
24   function str(slv : std_logic_vector) return string is
25     alias slv_norm    : std_logic_vector(1 to slv'length) is slv;
26     variable sl_str_v : string(1 to 1);  -- String of std_logic
27     variable res_v    : string(1 to slv'length);
28   begin
29     for idx in slv_norm'range loop
30       sl_str_v   := str(slv_norm(idx));
31       res_v(idx) := sl_str_v(1);
32     end loop;
33     return res_v;
34   end function;
35
36   function hstr(s: std_logic_vector) return string is
37     --- Locals to make the indexing easier
38     constant s_norm: std_logic_vector(4 to s'length+3) := s;
39     variable result: string (1 to s'length/4);
40     --- A subtype to keep the VHDL compiler happy
41     --- (the rules about data types in a CASE are quite strict)
42     subtype slv4 is std_logic_vector(1 to 4);
43   begin
44     assert (s'length mod 4) = 0
45       report "SLV must be a multiple of 4 bits"
46       severity FAILURE;
47     for i in result'range loop
48       case slv4'(s_norm(i*4 to i*4+3)) is
```

```
49          when "0000" => result(i) := '0';
50          when "0001" => result(i) := '1';
51          when "0010" => result(i) := '2';
52          when "0011" => result(i) := '3';
53          when "0100" => result(i) := '4';
54          when "0101" => result(i) := '5';
55          when "0110" => result(i) := '6';
56          when "0111" => result(i) := '7';
57          when "1000" => result(i) := '8';
58          when "1001" => result(i) := '9';
59          when "1010" => result(i) := 'A';
60          when "1011" => result(i) := 'B';
61          when "1100" => result(i) := 'C';
62          when "1101" => result(i) := 'D';
63          when "1110" => result(i) := 'E';
64          when "1111" => result(i) := 'F';
65          when others => result(i) := 'x';
66        end case;
67      end loop;
68      return result;
69    end;
70  end package body;
```

Troisième partie

Composants utiles

Chapitre 7

Composants utiles

7.1 Timer

7.2 Afficheur 7 segments

Les afficheurs 7-segments sont constitués d'un ensemble de LEDs (diodes électro-luminescentes), contrôlables individuellement. Un afficheur 7-segments permet l'affichage d'un unique digit hexadécimal. Notons que nous ne sommes pas strictement restreint à l'hexadécimal, puisque nous avons le contrôle sur chacune des LEDs. Il faut par contre bien comprendre que lorsque plusieurs digits sont disponibles sur une carte électronique, le contrôle des digits est global, car ils *partagent un certains nombre de signaux* : il faut un contrôleur qui séquence l'affichage successif sur les digits. Par conséquent, le simple contrôle de ces afficheurs représente un exercice intéressant.

Utilisation basique L'utilisation d'un afficheur 7-segment est très pratique et immédiat si on cherche à afficheur des données hexadécimal. Par exemple, si on manipule une donnée codée sur 8 bits non signés, l'affichage de cette valeur sur les LEDs consiste à router les 4 bits de poids fort vers un digits (à gauche) et les 4 bits de poids faible (LSB) vers le digit de droite. Cette extraction des quartets est immédiate.

Carte Nexys4DDR Sur la carte Nexys4DDR, nous disposons de 8 afficheurs 7-segments : comme expliqué à l'instant, ces 8 afficheurs partagent des signaux.

7.3 Transformation décimal vers BCD : algorithme Double-Dabble

Dans le cas où on cherche à afficher une donnée non plus sous forme hexadécimale, mais sous forme décimale (base 10), les choses se compliquent : par exemple, à partir d'un nombre décimal 123 contenu dans un registre 8 bits, l'extraction des 3 digits décimaux ne peut se faire par un routage direct de certains fils. Nous savons que nous devons représenter le nombre 123_{10} en représentation BCD. Dans cette représentation chaque digit décimal nécessite 4 bits. On a : $123_{10} = 000100100011_{2,BCD} = 291_{10,BCD}$
La transformation $\mathbb{N} \rightarrow \mathbb{N}$ est non-triviale. Il existe un algorithme qui réalise cette transformation : c'est l'algorithme Double-Dabble.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.numeric_std.all;
4
5
6  entity bin2bcd_12bit is
7      Port ( binIN : in  STD_LOGIC_VECTOR (11 downto 0);
8             ones : out  STD_LOGIC_VECTOR (3 downto 0);
9             tens : out  STD_LOGIC_VECTOR (3 downto 0);
10            hundreds : out  STD_LOGIC_VECTOR (3 downto 0);
11            thousands : out  STD_LOGIC_VECTOR (3 downto 0)
12           );
13 end bin2bcd_12bit;
14
15 architecture Behavioral of bin2bcd_12bit is
```

```
16
17 begin
18
19 bcd1: process(binIN)
20
21   -- temporary variable
22   variable temp : STD_LOGIC_VECTOR (11 downto 0);
23
24   -- variable to store the output BCD number
25   -- organized as follows
26   -- thousands = bcd(15 downto 12)
27   -- hundreds = bcd(11 downto 8)
28   -- tens = bcd(7 downto 4)
29   -- units = bcd(3 downto 0)
30   variable bcd : UNSIGNED (15 downto 0) := (others => '0');
31
32   -- by
33   -- https://en.wikipedia.org/wiki/Double_dabble
34
35   begin
36     -- zero the bcd variable
37     bcd := (others => '0');
38
39     -- read input into temp variable
40     temp(11 downto 0) := binIN;
41
42     -- cycle 12 times as we have 12 input bits
43     -- this could be optimized, we do not need to check and add 3 for the
44     -- first 3 iterations as the number can never be >4
45     for i in 0 to 11 loop
46
47       if bcd(3 downto 0) > 4 then
48         bcd(3 downto 0) := bcd(3 downto 0) + 3;
49       end if;
50
51       if bcd(7 downto 4) > 4 then
52         bcd(7 downto 4) := bcd(7 downto 4) + 3;
53       end if;
54
55       if bcd(11 downto 8) > 4 then
56         bcd(11 downto 8) := bcd(11 downto 8) + 3;
57       end if;
58
59       -- thousands can't be >4 for a 12-bit input number
60 ------- so don't need to do anything to upper 4 bits of bcd
61
62       -- shift bcd left by 1 bit, copy MSB of temp into LSB of bcd
63       bcd := bcd(14 downto 0) & temp(11);
64
65       -- shift temp left by 1 bit
66       temp := temp(10 downto 0) & '0';
67
68     end loop;
69
70     -- set outputs
71     ones <= STD_LOGIC_VECTOR(bcd(3 downto 0));
72     tens <= STD_LOGIC_VECTOR(bcd(7 downto 4));
73     hundreds <= STD_LOGIC_VECTOR(bcd(11 downto 8));
74     thousands <= STD_LOGIC_VECTOR(bcd(15 downto 12));
75
76   end process bcd1;
77
78 end Behavioral;
```

7.4 FIFO

Une FIFO est une structure de données qui permet de mémoriser une quantité finie d'informations, au fur et à mesure que ces informations arrivent. La relecture des informations stockées ne peut se faire que dans un certain ordre : c'est d'abord la plus ancienne des informations qui est accédée, et ainsi de suite, jusqu'à ce que la FIFO ne contienne plus d'informations. Ce mode de fonctionnement est important car il permet d'introduire de l'asynchronisme entre deux traitements émetteurs et récepteurs.

Le mot "asynchrone" a déjà été utilisé ici à propos des resets, mais il est utilisé ici d'un point de vue algorithmique : il s'agit d'offrir la possibilité à l'émetteur et au recepteur de travailler à leur rythme. En ce sens, émetteur et récepteur peuvent très bien fonctionner à la même horloge *clk*, mais s'échanger des données "quand ils le peuvent". Bien évidemment, la FIFO est à l'occasion le composant idéal capable d'introduire, si on le souhaite effectivement, des horloges physiques différentes : une horloge pour l'entrée et une horloge pour la sortie, mais ceci n'est pas une obligation. On rappelle que le fait de recourir à plusieurs horloges physiques peut être source de problèmes techniques fins et est généralement à proscrire en première approche.

7.5 UART

Chapitre 8

Interaction PC-FPGA

8.1 Choix de l'UART

8.2 Présentation du composant

8.3 Protocole d'interaction avec les registres de configuration

8.4 Exemple d'utilisation sous Linux

Quatrième partie

Travaux pratiques

Chapitre 9

Moyenne mobile

9.1 Présentation du sujet

La notion de moyenne mobile est utilisée pour monitorer des flux rapides de données : c'est le cas du high-speed trading, mais un grand nombre de systèmes de traitement du signal se basent sur cette moyenne, plutôt que sur la valeur des échantillons bruts. La moyenne mobile d'un flux continu d'échantillons se définit comme la moyenne des m derniers échantillons reçus. Un algorithme de moyenne mobile doit donc supprimer la donnée la plus ancienne au profit de la plus récente, et actualiser le calcul de la moyenne, à chaque échantillon.

9.2 Travail à réaliser

En supposant que $m = 2^n$:

— Prototyper un algorithme de moyenne mobile dans votre langage préféré (15mn max).
— Dessiner une solution RTL à partir des éléments suivants : additionneurs et registres (15mn).
— Ecrire le code VHDL de votre système.
— Ecrire un banc de test (inspiré du code de la LED), permettant de simuler dans les mêmes conditions qu'en 1.
— On cherche maintenant à améliorer le système précédent, en détectant des franchissements de seuils paramétrés par l'utilisateur : dès que la moyenne franchit les valeurs MIN et MAX, un signal 'alerte_low' et alerte_high' sont émis par le système. Modifier le circuit, et le système précédent pour incorporer ces nouvelles features.

9.3 Solution proposée

9.3.1 Algorithme de référence : *golden model* en Ruby

Nous présentons ici un algorithme de référence écrit dans le langage Ruby, sous la forme d'une fonction très simple, écrite en moins de 10 lignes de code.

```
1 def moyenne_mobile t,length=4
2   res=[]
3   for i in 0..t.size
4     res << t.take(length).inject(0,:+)/length
5     t.shift
6   end
7   res
8 end
```

Ruby, Python et Matlab sont très utilisés pour étudier l'algorithme visé, mais également générer les fichiers de stimuli, ainsi que les fichiers de référence, qui pourront être utilisés au cours de la simulation du banc de test.

9.3.2 Cas simple à profondeur de 4

Commençons par une moyenne mobile de profondeur fixe, 4 en l'occurrence : les 4 derniers échantillons sont sommés, puis la somme résultant est divisée par 4. Cet manière de procéder reste naïve, mais nous permet de prendre rapidement connaissance du problème, de manière Agile.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
-------------------------------------------------------------------------
-- The entity is shared (the same) between several architectures.
-- Hence some generic parameters are not all used in each architecture.
-------------------------------------------------------------------------
entity moving_average is
  generic (
    n : natural :=2;
    sample_width : natural := 8; -- nb bits for signed sample
    output_width : natural := 8
  );
  port(
    reset_n           : in  std_logic;
    clk               : in  std_logic;
    sample            : in  signed(sample_width-1 downto 0);
    sample_valid      : in  std_logic;
    mean              : out signed(output_width-1 downto 0)
  );
end entity;

architecture depth_4 of moving_average is
  signal r1,r2,r3,r4 : signed(sample_width-1 downto 0);
begin

  process(reset_n, clk)
  begin
    if reset_n = '0' then
      r1 <= to_signed(0,sample_width);
      r2 <= to_signed(0,sample_width);
      r3 <= to_signed(0,sample_width);
      r4 <= to_signed(0,sample_width);
    elsif rising_edge(clk) then
      if sample_valid = '1' then
        r1 <= sample;
        r2 <= r1;
        r3 <= r2;
        r4 <= r3;
      end if;
    end if;
  end process;

  process(reset_n, clk)
   variable tmp : signed(sample_width-1 downto 0);
  begin
    if reset_n = '0' then
      mean <= to_signed(0,sample_width);
    elsif rising_edge(clk) then
      tmp  := (r1+r2)+(r3+r4);
      mean <= "00" & tmp(sample_width-1 downto 2);
    end if;
  end process;

end depth_4;
```

9.3.3 Cas générique à profondeur n : version 1

Version générique 1 Nous proposons désormais une solution plus évoluée, qui prend en paramètre générique le nombre des n échantillons conservés dans le calcul de la moyenne mobile. Très vite, nous nous sommes rendu à l'évidence : il n'est pas raisonnable d'augmenter le nombre d'additionneurs, comme la solution précédente le laisser envisager. Nous avons donc modifié la manière de procéder algorithmiquement : lorsqu'un nouvel échantillon est introduit dans notre registre à décallage, il est additionné à la somme courante tandis que le dernier échantillon est soustrait à cette même somme : un seul additionneur et un soustracteur sont donc nécessaires, quel que soit le nombre n d'échantillons

pris en considération. Notons que le calcul de la moyenne finale *mean* fait appel à la fonction prédéfinie *shift_right*

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

use work.util_pkg.log2;

architecture depth_generic of moving_average is
  type reg_array is array(0 to n-1) of signed(sample_width-1 downto 0);
  signal reg : reg_array;
  signal sum : signed(sample_width-1 downto 0);
begin

  process(reset_n, clk)
  begin
    if reset_n = '0' then
      for i in 0 to n-1 loop
        reg(i) <= to_signed(0,sample_width);
      end loop;
    elsif rising_edge(clk) then
      if sample_valid = '1' then
        reg(0)<= sample;
        for i in 0 to n-2 loop
          reg(i+1) <= reg(i);
        end loop;
      end if;
    end if;
  end process;

  process(reset_n, clk)
  begin
    if reset_n = '0' then
      sum <= to_signed(0,sample_width);
    elsif rising_edge(clk) then
      sum <= sum + reg(0) - reg(n-1);
    end if;
  end process;

  mean <= shift_right(sum,log2(n));

end depth_generic;
```

Test de la solution Le banc de test se présente sous la forme du design implementé, et d'un processus de génération de stimuli. Cette génération est basée sur la lecture d'un tableau d'échantillon, codé en dur dans l'architecture du banc de test.

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

use work.util_pkg.all;--log2

entity moving_average_tb is
  generic (
    n : natural :=4;--number of samples taken into account
    sample_width : natural := 8
  );
end entity;

architecture bhv of moving_average_tb is
  constant HALF_PERIOD : time := 5 ns;
  signal clk           : std_logic := '1';
  signal reset_n       : std_logic;
  signal running       : boolean:=true;
  signal sample        : signed(sample_width-1 downto 0);
  signal sample_valid  : std_logic;
  signal mean          : signed(sample_width-1 downto 0);

  type samples_t is array(integer range <>) of integer;

```

```
  signal samples : samples_t(0 to 199) :=(
       12, 14,  5, 15, 22, 11, 16, 34, 27, 20,
       36, 27, 19, 28, 22, 39, 32, 30, 33, 26,
       38, 30, 15,  5, 15, 11,  5, 23, 19,  4,
       12, -8,-10,  7, -4, -4,-23, -5,-32,-26,
      -24,-29,-35,-28,-41,-43,-31,-20,-14,-37,
      -17,-12,-25,-20,-19,-14, -2,-19, -1,-12,
       -7,  7, 12, -3, -8, 23, 13, 20, 32, 12,
       32, 37, 32, 18, 34, 39, 36, 16, 37, 18,
       32, 31, 21, 28, 21, 24, 19, 14, 15, 23,
       24, 10,  0,-13, 11, -2,  5,  0, -3, -4,
      -12,-14,-19,-39,-32,-35,-31,-28,-35,-15,
      -22,-44,-30,-21,-34,-19,-18,-32, -9,-26,
      -25,  3,  5,-20,-16, 12, -7, -5,  5, 15,
       17, 25, 10,  8, 15, 20, 20, 35, 34, 25,
       39, 25, 25, 26, 14, 22, 34, 18, 32, 25,
       20, 21, 26,  5, 17, 14,  0,-10,-21, -2,
      -12,-14,-26,-30,-27,-21,-36,-33,-42,-16,
      -23,-17,-33,-20,-23,-30,-14,-31,-25,-38,
      -23,-27,-26,-19,-11,-15,-16,-10, 12,  8,
       21,  5, 11,  4, 23, 27, 34, 37, 22, 37
  );

begin

  DUT : entity work.moving_average(depth_generic)
    generic map (
      n => n,
      sample_width => sample_width,
      output_width => sample_width -- same range as input
    )
    port map(
      reset_n          => reset_n,
      clk              => clk,
      sample           => sample,
      sample_valid     => sample_valid,
      mean             => mean
    );

  clk <= not(clk) after HALF_PERIOD when running else '0';
  reset_n <= '0','1' after 66 ns;

  stim:process
  begin
    sample <= to_signed(0,sample_width);
    sample_valid <= '0';
    wait until reset_n='1';
    report "size of moving average : " & integer'image(n);
    report "starting samples...";
    for i in 0 to samples'length-1 loop
      wait until rising_edge(clk);
      sample <= to_signed(samples(i),sample_width);
      sample_valid <= '1';
    end loop;
    sample_valid <= '0';
    report "end of simulation";
    running <= false;
    wait;
  end process;

end bhv;
```

Nous testons notre implémentation générique. Les premiers tests s'avèrent fructueux, comme exposé sur le chronogramme 9.1.

9.4 Cas générique à profondeur n : version 2

Notre seul test précédent, dont les échantillons étaient codés en dur, ne peut suffir à se convaincre du bon fonctionnement du circuit. Il faut pour cela explorer son comportement un peu plus...Pour cela, on modifie le banc de test de manière à lire dansun fichiers externe les échantillons générés par notre modèle de référence. Ce listing est proposé ici.

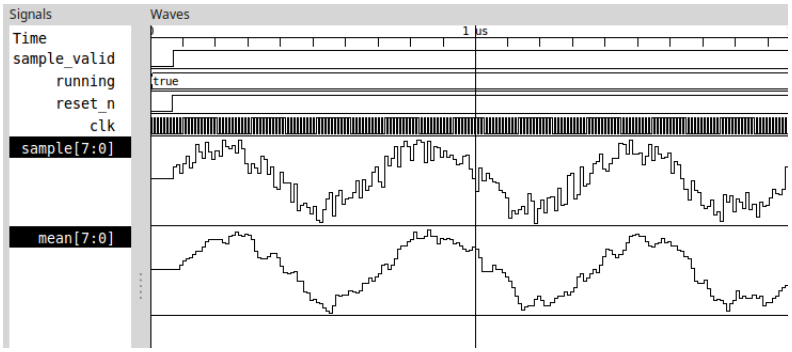


FIGURE 9.1 – Premier test de la moyenne mobile générique : profondeur 4, échantillons signés sur 8 bits

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  library std;
6  use std.textio.all;
7
8  use work.util_pkg.all;--log2
9
10 entity moving_average_file_stim_tb is
11   generic (
12     n : natural :=4;--number of samples taken into account
13     sample_width : natural := 8
14   );
15 end entity;
16
17 architecture bhv of moving_average_file_stim_tb is
18   constant HALF_PERIOD : time := 5 ns;
19   signal clk           : std_logic := '1';
20   signal reset_n       : std_logic;
21   signal running       : boolean:=true;
22   signal sample        : signed(sample_width-1 downto 0);
23   signal sample_valid  : std_logic;
24   signal mean          : signed(sample_width-1 downto 0);
25
26 begin
27
28   DUT : entity work.moving_average(depth_generic)
29     generic map (
30       n => n,
31       sample_width    => sample_width,
32       output_width    => sample_width
33     )
34     port map(
35       reset_n          => reset_n,
36       clk              => clk,
37       sample           => sample,
38       sample_valid     => sample_valid,
39       mean             => mean
40     );
41
42   clk <= not(clk) after HALF_PERIOD when running else '0';
43   reset_n <= '0','1' after 66 ns;
44
45   stimuli_proc : process
46    file F : text;
47    variable L: line;
48    variable status : file_open_status;
49    variable data : integer;
50    variable nb_samples : natural := 0;
51   begin
52     FILE_OPEN(status,F,"samples.txt",read_mode);
53     if status/=open_ok then
54       report "problem to open stimulus file samples.text" severity error;
```

```
55      else
56        sample <= to_signed(0,sample_width);
57        sample_valid <= '0';
58        wait until reset_n='1';
59        report "size of moving average : " & integer'image(n);
60        report "starting samples...";
61        while not(ENDFILE(f)) loop
62          nb_samples:=nb_samples+1;
63          wait until rising_edge(clk);
64          readline(F,l);
65          read(l,data);
66          sample <= to_signed(data,sample_width);
67          sample_valid <= '1';
68        end loop;
69        sample_valid <= '0';
70        report "end of simulation";
71        report integer'image(nb_samples) & " samples processed.";
72      end if;
73      running <= false;
74      wait;
75    end process;
76
77  end bhv;
```

Analyse de la simulation Une politique de tests massifs révèle que, dans certains cas, la moyenne mobile se révélait erratique : comme le montre le chronogramme, il existe en effet des cas, où la somme des échantillons n'est pas correctement calculée. Cela peut arriver lorsque la somme de n échantillons dépasse la valeur 2^{nb_bits}. Il s'agit d'un bug de conception dans le dimensionnement des variables du problème (signaux en l'occurence !).

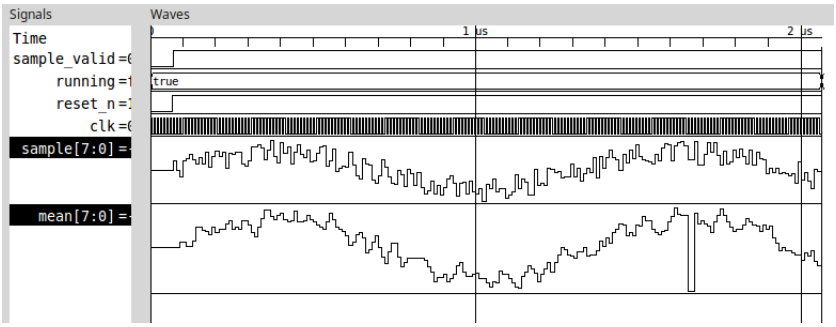


FIGURE 9.2 – Test de la moyenne mobile générique révélant un bug (profondeur 16, échantillons signés sur 8 bits)

Calcul de la dynamique exacte Combien de bits faut-il pour coder le résultat de la somme de n valeurs codées sur m bits ? On rappelle que le nombre de bits nécessaires au codage d'un entier naturel x est donné par :

$$D(x) = \lfloor log_2(x) + 1 \rfloor$$

Posons : $S = \sum_{i=0}^{n-1} x_i$, avec $x_i \in \{0 \dots 2^m - 1\}$. On a $S_{max} = n.(2^m - 1)$ et donc :

$$D(S_{max}) = \lfloor log_2(2^m - 1) + log_2(n) + 1 \rfloor$$

Dans le cas d'un entier *relatif* codé sur m bits, sa plage de valeur est $-2^{m-1}, \dots, +2^{m-1} - 1$. En sommant n fois, la plus grande valeur absolue est donc : $|S|_{max} = n.2^{m-1}$, qui s'encode avec

$$D(|s|_{max}) = \lfloor log_2(n.2^{m-1}) + 1 \rfloor \text{ bits}$$

En prenant en compte le bit de signe supplémentaire, le nombre de bits nécessaires à la somme de n entiers relatifs codés sur m bits est donc :

$$\lfloor m + log_2(n) - 1 + 1 + 1 \rfloor = \lfloor m + log_2(n) + 1 \rfloor$$

.

9.4.1 Cas générique à profondeur n : version 2, sans bug

Une correction du bug est proposée ici, respectant une nouvelle dynamique de sortie, différente de celle d'entrée, mais fonction d'elle.

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

use work.util_pkg.all;

architecture depth_generic_v2 of moving_average is
  type reg_array is array(0 to n-1) of signed(sample_width-1 downto 0);
  signal reg : reg_array;
  signal sum : signed(output_width-1 downto 0);
begin

  process(reset_n, clk)
  begin
    if reset_n = '0' then
      for i in 0 to n-1 loop
        reg(i) <= to_signed(0,sample_width);
      end loop;
    elsif rising_edge(clk) then
      if sample_valid = '1' then
        reg(0)<= sample;
        for i in 0 to n-2 loop
          reg(i+1) <= reg(i);
        end loop;
      end if;
    end if;
  end process;

  process(reset_n, clk)
  begin
    if reset_n = '0' then
      sum <= to_signed(0,output_width);
    elsif rising_edge(clk) then
      sum <= sum + resize(reg(0),output_width) - resize(reg(n-1),output_width);
    end if;
  end process;

  mean <= shift_right(sum,log2(n));

end depth_generic_v2;
```

Le testbench légèrement modifié est également proposé : dans ce testbench, on calcule notamment la dynamique de la sortie exacte.

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

library std;
use std.textio.all;

use work.util_pkg.all;--log2

-- ================ V2 architecture ========================
-- FIXES : bug in sum dynamics.
-- =========================================================

entity moving_average_file_stim_v2_tb is
  generic (
    n : natural :=16;--number of samples taken into account
    sample_width : natural :=8
  );
end entity;

architecture bhv of moving_average_file_stim_v2_tb is

  constant output_width : natural := floor(log2(sample_width)+n+1);

  constant HALF_PERIOD : time := 5 ns;
```

```
  signal clk            : std_logic := '1';
  signal reset_n        : std_logic;
  signal running        : boolean:=true;
  signal sample         : signed(sample_width-1 downto 0);
  signal sample_valid   : std_logic;
  signal mean           : signed(output_width-1 downto 0);



begin

  DUT : entity work.moving_average(depth_generic_v2) -- V2
    generic map (
      n => n,
      sample_width => sample_width,
      output_width => output_width
    )
    port map(
      reset_n           => reset_n,
      clk               => clk,
      sample            => sample,
      sample_valid      => sample_valid,
      mean              => mean
    );

  clk <= not(clk) after HALF_PERIOD when running else '0';
  reset_n <= '0','1' after 66 ns;

  stimuli_proc : process
   file F : text;
   variable L: line;
   variable status : file_open_status;
   variable data : integer;
   variable nb_samples : natural := 0;
  begin

    report "n (window size) = " & integer'image(n);
    report "sample_width    = " & integer'image(sample_width);
    report "output_width    = " & integer'image(output_width);

    FILE_OPEN(status,F,"samples.txt",read_mode);
    if status/=open_ok then
      report "problem to open stimulus file samples.text" severity error;
    else
      sample <= to_signed(0,sample_width);
      sample_valid <= '0';
      wait until reset_n='1';
      report "size of moving average : " & integer'image(n);
      report "starting samples...";
      while not(ENDFILE(f)) loop
        nb_samples:=nb_samples+1;
        wait until rising_edge(clk);
        readline(F,l);
        read(l,data);
        sample <= to_signed(data,sample_width);
        sample_valid <= '1';
      end loop;
      sample_valid <= '0';
      report "end of simulation";
      report integer'image(nb_samples) & " samples processed.";
    end if;
    running <= false;
    wait;
  end process;

end bhv;
```

Analyse La simulation semble désormais correcte, du moins visuellement.

9.4.2 Annexe : makefile

A toutes fins utiles un Makefile simple est donné pour automatiser l'exploration de notre design.

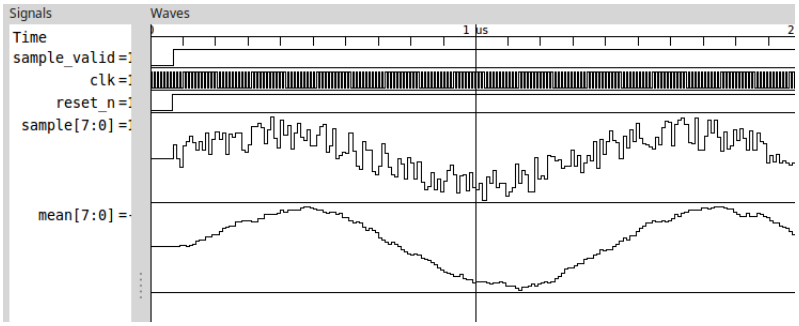


FIGURE 9.3 – Test de la moyenne mobile générique révélant un bug (profondeur 16, échantillons signés sur 8 bits)

```
GHDL=ghdl
GHDLFLAGS=
MODULES=\
     util_pkg.o\
     moving_average.o \
     moving_average_generic.o \
     moving_average_tb

all: moving_average_tb.ghw
        gtkwave moving_average_tb.ghw moving_average_tb.sav

moving_average_tb.ghw: $(MODULES)
        ghdl -r moving_average_tb --wave=moving_average_tb.ghw

# Binary depends on the object file
%: %.o
        $(GHDL) -e $(GHDLFLAGS) $@

# Object file depends on source
%.o: %.vhd
        $(GHDL) -a $(GHDLFLAGS) $<

clean:
        echo "Cleaning up..."
        rm -f *.o *_tb work*.cf *.ghw *.sav
```

Chapitre 10

Chemin de données contrôlable

10.1 Présentation du sujet

Nous cherchons ici à modéliser en VHDL un chemin de données contrôlé par un séquenceur. Ce chemin de données (ou *datapath* en anglais) peut-être vu comme l'embryon des unités de calculs d'un microprocesseur. Toutefois, le datapath d'un microprocesseur est contrôlé par un programme résidant en mémoire ; ici, nous cherchons à contrôler ce datapath par une machine à états finis (non reprogrammable). Cette séparation FSM-datapath est fréquente en électronique numérique.

10.2 Travail à réaliser

Soit le circuit suivant, représentatif d'un chemin de données (datapath) d'un petit processeur. Afin de réaliser un calcul donné (ici $y = a*x+b$) , on cherche à piloter ce chemin de données par un automate qui séquence et envoie les bonnes commandes aux bons cycles d'horloge. On suppose également que la trame d'arrivée des données a,x et b est donnée (voir schéma).

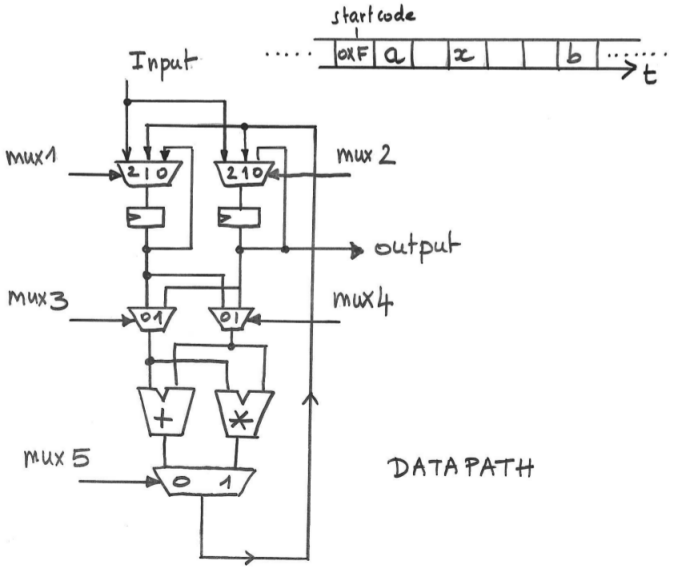


FIGURE 10.1 – Datapath

1. Entourer le nuage combinatoire (quels éléments en présence ?) et localiser les registres.
2. Comment stocker une valeur qui se présente sur l'entrée ?
3. Comment router des valeurs stockées dans les registres vers les opérateurs ?
4. Coder le datapath en VHDL. Tester à l'aide d'un testbench.
5. Dessiner l'automate ("diagramme à bulles") qui permet de séquencer le calcul (bien prendre en compte la trame d'arrivée des données)

6. Coder l'automate en VHDL et tester.
7. Assembler le système complet en instanciant les deux composants précédents (fsm+datapath).
8. Tester.

10.3 Solution proposée

10.3.1 Combinatoire et séquentiel

La partie combinatoire a été entourée sur le schéma suivant. On y distingue :
— des multiplexeurs à l'entrée des registres, permettant de router des données vers ces registres, et provenant de différentes sources (entrées, opérateurs, etc).
— des opérateurs arithmétiques. Ici + et ∗

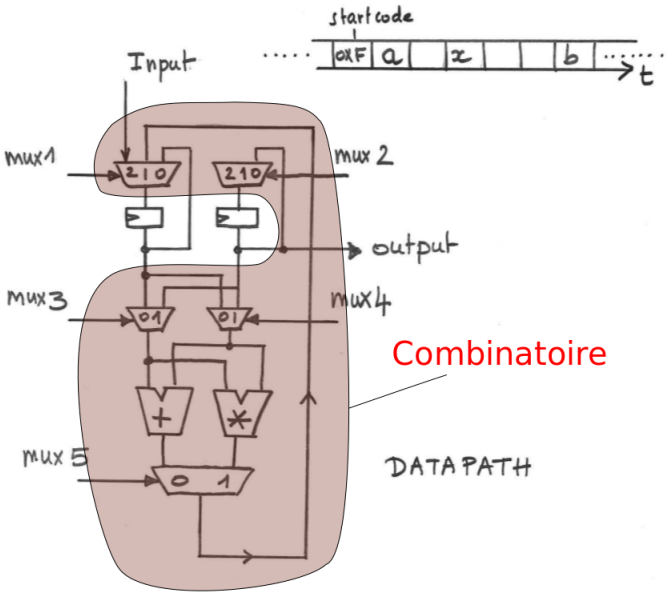


FIGURE 10.2 – Datapath : partie combinatoire (entourée) et séquentielle (registres)

10.3.2 Routage des données

Il est aisé de constater que les commandes des multiplexeurs permettent de router les données dans ce datapath. Prenons le registre de gauche : en le pilotant en combinatoire par le signal *mux1=2*, la donnée d'entrée est dirigée vers l'entrée du registre, qui l'échantillonne. Au cycle suivant, si l'on souhaite conserver cette valeur stockée, il ne faut pas oublier de piloter le multiplexeur par *mux1=0*, afin que la donnée soit effectivement "piégée" (on parle de *recirculation* de la donnée). A noter que cette recirculation est souvent masquée par la mise à disposition d'un signal "enable" dans la bascule : il s'agit en réalité d'un multiplexeur qui, en interne, fait recirculer la donnée ; dans notre schéma, nous avons "banalisé" de signal "enable" parmi les 3 chemins du multiplexeur. Cette banalisation est très naturelle dans le codage VHDL (*if...elsif* imbriqués).

De la même manière, on comprend désormais que pour router les valeurs stockées dans les registres vers les opérateurs, il faut piloter les signaux *mux3* et *mux4* de manière adéquate. On comprend alors bien désormais la terminologie "chemin de données".

10.3.3 Datapath en VHDL

Nous proposons ici un codage de ce datapath. Il existe plusieurs alternatives possibles.

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

```
use ieee.numeric_std.all;

entity datapath is
  generic(N : natural := 16);
  port(
    reset_n : in std_logic;
    clk     : in std_logic;
    input   : in signed(N-1 downto 0);
    mux1    : in std_logic_vector(1 downto 0);
    mux2    : in std_logic_vector(1 downto 0);
    mux3    : in std_logic;
    mux4    : in std_logic;
    mux5    : in std_logic;
    output  : out signed(N-1 downto 0)
  );
end datapath ;

architecture rtl of datapath is
  signal reg_l,reg_r     : signed(N-1 downto 0);
  signal m1,m2,m3,m4,m5 : signed(N-1 downto 0);
begin

  -- synchronous (a.k.a 'clocked') process
  process(reset_n,clk)
  begin
    if reset_n='0' then
      reg_l <= to_signed(0,16);
      reg_r <= to_signed(0,16);
    elsif rising_edge(clk) then
      reg_l <= m1;
      reg_r <= m2;
    end if;
  end process;

  -- conditional assignments
  m1 <= input when mux1="10" else
        m5 when mux1="01" else
        reg_l;

  m2 <= input when mux2="10" else
        m5 when mux2="01" else
        reg_r;

  m3 <= reg_l when mux3='0' else
        reg_r;

  m4 <= reg_l when mux4='0' else
        reg_r;

  m5 <= (m3+m4) when mux5='0' else
        resize((m3*m4),N);

  output <= reg_r;

end rtl;
```

10.3.4 Test du datapath à l'aide d'un testbench (sans contrôleur)

```
--------------------------------------------------------------------------------
-- This file was generated automatically by tb_gen Ruby utility
-- date : (d/m/y) 24/10/2017 11:51
-- Author : Jean-Christophe Le Lann - 2014
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity datapath_tb is
end entity;

architecture bhv of datapath_tb is

```

```
15   constant HALF_PERIOD : time := 5 ns;
16
17   constant N : natural :=16;
18
19   signal clk     : std_logic := '0';
20   signal reset_n : std_logic := '0';
21   signal sreset  : std_logic := '0';
22   signal running : boolean   := true;
23
24   procedure wait_cycles(n : natural) is
25    begin
26      for i in 1 to n loop
27        wait until rising_edge(clk);
28      end loop;
29    end procedure;
30
31   signal input : signed(N-1 downto 0);
32   signal output : signed(N-1 downto 0);
33
34   type controls_type is record
35     mux1 : std_logic_vector(1 downto 0);
36     mux2 : std_logic_vector(1 downto 0);
37     mux3 : std_logic;
38     mux4 : std_logic;
39     mux5 : std_logic;
40   end record;
41
42   constant controls_default : controls_type := (
43     mux1 => "00",
44     mux2 => "00",
45     mux3 => '0',
46     mux4 => '0',
47     mux5 => '0'
48   );
49
50   signal datapath_control : controls_type;
51   signal done : std_logic :='0';
52 begin
53   -------------------------------------------------------------------
54   -- clock and reset
55   -------------------------------------------------------------------
56   reset_n <= '0','1' after 666 ns;
57
58   clk <= not(clk) after HALF_PERIOD when running else clk;
59
60   -------------------------------------------------------------------
61   -- Design Under Test
62   -------------------------------------------------------------------
63   dut : entity work.datapath(rtl)
64         generic map(N=> N)
65         port map (
66           reset_n => reset_n,
67           clk     => clk,
68           input   => input,
69           mux1    => datapath_control.mux1,
70           mux2    => datapath_control.mux2,
71           mux3    => datapath_control.mux3,
72           mux4    => datapath_control.mux4,
73           mux5    => datapath_control.mux5,
74      output  => output);
75
76   -------------------------------------------------------------------
77   -- input generator : .....0xf A . X . . B .....
78   -------------------------------------------------------------------
79   input_gen : process
80     variable  a,x,b : signed(N-1 downto 0);
81     variable nb_tests : natural :=0;
82    begin
83      a := to_signed(0,N);
84      x := to_signed(1,N);
85      b := to_signed(2,N);
86      input <= to_signed(0,N);
87
88      report "running testbench for datapath(rtl)";
```

```
89      report "waiting for asynchronous reset";
90      wait until reset_n='1';
91      wait_cycles(4);
92      report "generating input stream";
93      while true loop
94        wait_cycles(1);
95        nb_tests:=nb_tests+1;
96        report "generation " & integer'image(nb_tests);
97        a := a + 1;
98        x := x + 1;
99        b := b + 1;
100       input <= resize(x"0F",N);
101       wait_cycles(1);
102       input <= a;
103       wait_cycles(1);
104       input <= to_signed(0,N);
105       wait_cycles(1);
106       input <=x;
107       wait_cycles(1);
108       input <= to_signed(0,N);
109       wait_cycles(2);
110       input <= b;
111       wait_cycles(1);
112       input <= to_signed(0,N);
113       wait_cycles(10);
114     end loop;
115     wait;
116   end process;
117
118 ------------------------------------------------------------------
119 -- sequential stimuli
120 ------------------------------------------------------------------
121 fsm_stim : process
122   begin
123     report "running controler *emulation* process";
124
125     for i in 0 to 1 loop
126       datapath_control <= controls_default;
127       done <= '0';
128       report "waiting for start code";
129       report "state Idle";
130       wait until input=resize(x"0F",N);
131
132       wait_cycles(1);
133       report "state SA";
134       datapath_control <= controls_default;
135       datapath_control.mux1 <= "10";
136
137       wait_cycles(1);
138       report "state SW1";
139       datapath_control <= controls_default;
140
141       wait_cycles(1);
142       report "state SX";
143       datapath_control <= controls_default;
144       datapath_control.mux2 <= "10";
145
146       wait_cycles(1);
147       report "state SW2";
148       datapath_control <= controls_default;
149       datapath_control.mux4 <= '1';
150       datapath_control.mux5 <= '1';
151       datapath_control.mux1 <= "01";
152       wait_cycles(1);
153       report "state SW3";
154       datapath_control <= controls_default;
155
156       wait_cycles(1);
157       report "state SB";
158       datapath_control <= controls_default;
159       datapath_control.mux2 <= "10";
160
161       wait_cycles(1);
162       report "state SY";
```

```
163         datapath_control <= controls_default;
164         datapath_control.mux4 <= '1';
165         datapath_control.mux5 <= '0';
166         datapath_control.mux2 <= "01";
167
168         wait_cycles(1);
169         datapath_control <= controls_default;
170         done <= '1';
171         wait_cycles(1);
172         done <= '0';
173
174       end loop;
175
176       wait_cycles(100);
177       report "end_of_simulation";
178       running <=false;
179       wait;
180     end process;
181
182 end bhv;
```

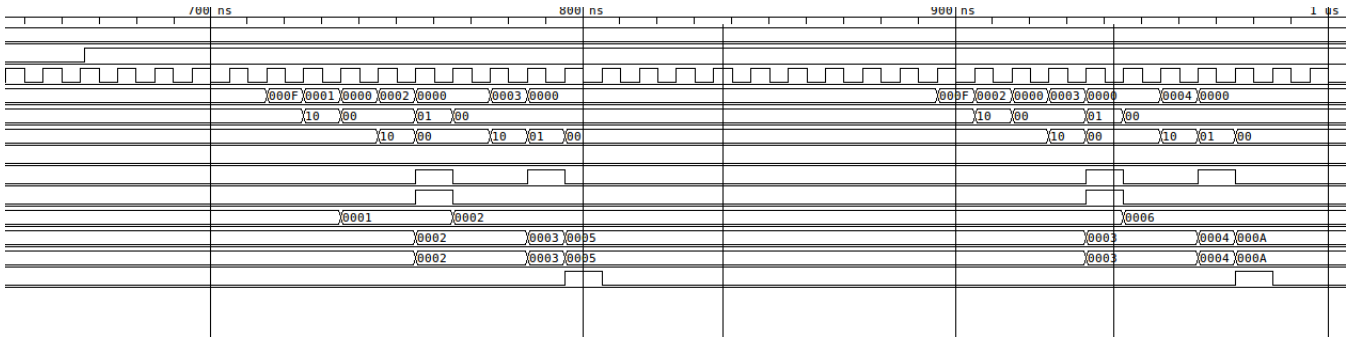


FIGURE 10.3 – Testbench du datapath *seul* : un processus (non syntétisable) génère une entrée respectant le protocole. Un second processus pilote le datapath de manière à réaliser le calcul de $y = ax + b$

10.3.5 Contrôleur en VHDL

Un schéma du contrôleur est proposé ici.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  use work.controler_pkg.all;
6
7  entity controler is
8    generic(N : natural := 16);
9    port(
10     reset_n : in  std_logic;
11     clk     : in  std_logic;
12     input   : in  signed(N-1 downto 0);
13     control : out controls_type;
14     done    : out std_logic
15     );
16 end entity;
17
18 architecture rtl of controler is
19   type state_type is (IDLE,SA,SW1,SX,SW2,SW3,SB,SY);
20   signal state_r, state_c : state_type;
21   signal done_s : std_logic;
22 begin
23
24
25   state_p : process(reset_n, clk)
26   begin
27     if reset_n = '0' then
28       state_r <= IDLE;
29     elsif rising_edge(clk) then
```

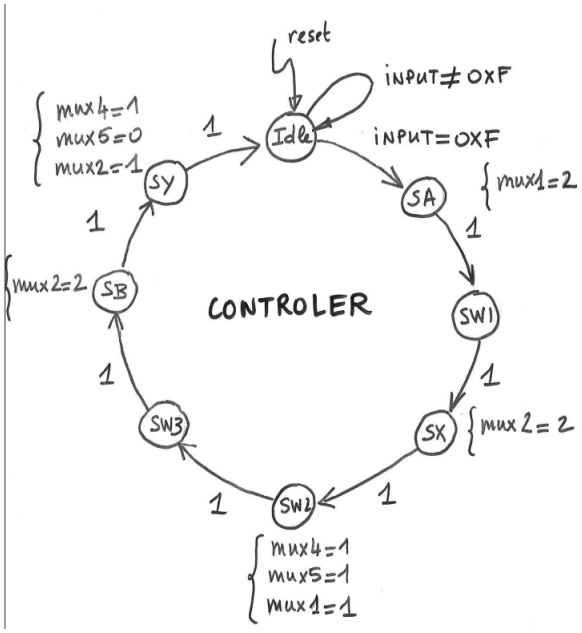


FIGURE 10.4 – Contrôleur sous forme de machine d'états finis(FSM), pilotant le chemin de données précédent.

```
30        state_r <= state_c;
31      end if;
32    end process;
33
34    next_state_function : process(input, state_r)
35      variable state_v : state_type;
36      variable control_v : controls_type;
37      variable done_v : std_logic;
38    begin
39      done_v := '0';
40      state_v   := state_r;
41      control_v := CONTROLS_DEFAULT;
42      case state_v is
43        when IDLE =>
44          if input = resize(x"0F",N) then
45            state_v := SA;
46          end if;
47        when SA =>
48          control_v.mux1 := "10";
49          state_v        := SW1;
50        when SW1 =>
51          state_v := SX;
52        when SX =>
53          control_v.mux2 := "10";
54          state_v        := SW2;
55        when SW2 =>
56          control_v.mux4 := '1';
57          control_v.mux5 := '1';
58          control_v.mux1 := "01";
59          state_v        := SW3;
60        when SW3 =>
61          state_v := SB;
62        when SB =>
63          state_v        := SY;
64          control_v.mux2 := "10";
65        when SY =>
66          control_v.mux4 := '1';
67          control_v.mux5 := '0';
68          control_v.mux2 := "01";
69          state_v        := IDLE;
70          done_v := '1';
71        when others =>
72          null;
73      end case;
```

```
74      state_c <= state_v;
75      control <= control_v;
76      done_s <= done_v;
77    end process;
78
79    delay_1t: process(reset_n,clk)
80    begin
81      if reset_n='0' then
82        done <= '0';
83      elsif rising_edge(clk) then
84        done <= done_s;
85      end if;
86    end process;
87
88  end rtl;
```

Notons que nous avons fait le choix de placer quelques définitions dans une *package* : il s'agit ici d'une structure de données de type "record" (struct en C) définissant un ensemble structuré de signaux de contrôle. Cette programmation structurée est fortement conseillée en VHDL, car elle simplifie la manipulation de signaux, tout en diminuant le nombre de signaux réellement manipulés par l'ordonnanceur lors de la simulation.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  package controler_pkg is
6
7    type controls_type is record
8      mux1 : std_logic_vector(1 downto 0);
9      mux2 : std_logic_vector(1 downto 0);
10     mux3 : std_logic;
11     mux4 : std_logic;
12     mux5 : std_logic;
13   end record;
14
15   constant CONTROLS_DEFAULT : controls_type := (
16     mux1 => "00",
17     mux2 => "00",
18     mux3 => '0',
19     mux4 => '0',
20     mux5 => '0'
21   );
22
23 end package;
```

10.3.6 FSMD : FSM+Datapath

L'assemblage du couple contrôleur et datapath conduit à la formation d'un nouveau composant appelé "FSMD" (finite state machine & datapath). En général le datapath émet des status vers le contrôleur, qui prend des décisions quant au flot de contrôle à exécuter et change d'état en conséquence. Le contrôleur émet en retoiur des signaux de contrôle vers le datapath, afin de router les données vers les unités de calculs et les résultats de ces calculs vers les registres de travail. Un schéma générique est proposé sur la figure 10.5. Lors de notre exercice, nous avons légèrement dévié de ce schéma générique : dans notre cas, nous n'avons pas de signal de démarrage (go), ni de signaux de status. La raison tient à la simplicité du circuit que nous concevons ici. Le seul status éventuel aurait été la détection vraie ou fausse d'un start code; il a été remplacé par un accès direct du signal d'entrée par le contrôleur.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  use work.controler_pkg.all;
6
7  entity fsmd is
8    generic(N : natural := 16);
9    port(
10     reset_n : in  std_logic;
11     clk     : in  std_logic;
12     input   : in  signed(N-1 downto 0);
```

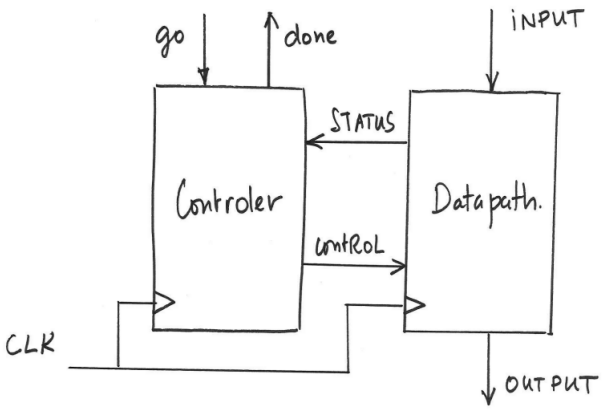


FIGURE 10.5 – Assemblage FSM et Datapath ou "FSMD" explicite.

```
13      done    : out std_logic;
14      output  : out signed(N-1 downto 0)
15      );
16  end fsmd;
17
18  architecture rtl of fsmd is
19    signal control : controls_type;
20  begin
21
22    controler_1: entity work.controler
23      generic map (
24        N => N)
25      port map (
26        reset_n => reset_n,
27        clk     => clk,
28        input   => input,
29        control => control,
30        done    => done);
31
32    datapath_1: entity work.datapath
33      generic map (
34        N => N)
35      port map (
36        reset_n => reset_n,
37        clk     => clk,
38        input   => input,
39        mux1    => control.mux1,
40        mux2    => control.mux2,
41        mux3    => control.mux3,
42        mux4    => control.mux4,
43        mux5    => control.mux5,
44        output  => output);
45
46  end rtl;
```

Nosu ne présentons pas ici le banc de test, très similaire au précédent.

10.4 Conclusion

Ce TP nous a permis de coder un chemin de données, de le tester isolément, puis de concevoir la machine d'états finis qui permet de piloter ce chemin de données. L'ensemble résultant est une FSMD. Au passage, l'exercice nous a également permis, lors de la phase de montage du testbench, de décrire une trame (ou protocole) arrivant d'une source externe.

Comme précisé en introduction, le chemin de donnée peut être amélioré pour effectuer d'autres opérations, le rapprochant d'un datapath de véritable microprocesseur. La structure basée sur des multiplexeurs ne changerait pas fondamentalement. Le remplacement du contrôleur cablé (FSM) par un véritable séquenceur programmable est toutefois nécessaire.

De plus, le codage d'une telle FSMD *explicite* est un travail certes formateur (les concepts se dégagent clairement), mais fastidieux. Dans un TP prochain, nous allons coder de tels processeurs (programmables) en poussant le codage VHDL et en décrivant le datapath de manière *implicite*.

Chapitre 11

Conception d'un *Softcore* simple

11.1 Enoncé

Dans ce TP, nous entamons la conception d'un softcore : c'est un processeur décrit et délivré en VHDL (ou Verilog) au niveau RTL, et synthétisables sur FPGA ou ASIC, selon un flot RTL classique.

Par opposition, il existe des hard macros qui sont délivrés sous une forme bien moins portable. Les hard macros sont spécifiques à une technologie Silicium donnée (TSMC, Intel, IBM, Infineon,...).

Les plus connus des softcores sont :

— le processeur Sparc LEON, issu de l'Agence Spatiale Européenne.
— les processeurs PicoBlaze et Microblaze, fournis par Xilinx
— le processeur Nios-II fournir par Intel-Altera.
— une liste complète est disponible sur wikipedia, avec les différentes licences logicielles associées.

Note : ce type de softcores est un cauchemar pour les juristes ! Un softcore est-il du matériel ou du logiciel ?

Dans un temps imparti très serré, nous allons concevoir un processeur respectant le jeu d'instructions que nous avons retenu au début de ce cours : nous avions conçu un simulateur de jeu d'instruction (ISS). Nous allons maintenant répéter le même travail, mais en VHDL RTL. Après synthèse Xilinx, le softcore existera sur FPGA ! Ceci signifie que nous serrons capables d'exécuter le même code binaire sur le softcore synthétisé que sur notre simulateur de jeu d'instruction. L'ISS nous permettra aussi de mettre au point notre VHDL, en alimentant notre banc de test avec des exemples bien maîtrisés.

Partie 1 : préparer le banc de test Dans le banc de test, nous décidons de placer :

— La partie matérielle :
  — le softcore
  — la mémoire instruction
  — la mémoire de données
— Les stimuli nécessaires à la simulation :
  — génération de l'horloge
  — génération du reset
  — processus divers (lecture fichier etc)

Le **travail à réaliser** est le suivant :

1. Relancer votre logiciel d'assemblage pour générer un fichier un fichier ASCII contenant deux colonnes en hexadécimal : Adresse-Instruction.
2. Le banc de test est classique et devra incorporer un générateur de clock et reset.
3. Il devra également permettre la lecture du fichier ASCII contenant nos instructions "binaires" et les charger dans la mémoire d'instructions. Il faut donc créer un processus VHDL qui ouvre, puis lit ces deux informations sous forme hexadécimale et les écrit sur deux signaux. Ces deux signaux seront connectés à la mémoire d'instructions. Un exemple est fourni dans le booklet du cours.

Partie 2 : conception

1. Dessiner scrupuleusement l'assemblage des parties combinatoires et séquentielle d'un tel processeur, en vous posant les bonnes questions :

— Où est l'instruction initialement ?
— A quelle adresse ?
— Vers où se déplace-t-elle ?
— Où sont les données ?

2. Nommer scrupuleusement les signaux afférents : adresses, instructions, etc. Appuyez vous sur le booklet concernant les mémoires.
3. Dimensionner correctement votre système : taille des mémoires, cardinalité des signaux, etc

Partie 3 : codage

1. Décrire l'entité du circuit
2. Décrire un embryon de l'architecture (synthétisable) du circuit :
   — les registres de travail (au nombre de 32, mais cette taille peut être générique) en VHDL.
   — Le PC (program counter)
3. On s'intéresse maintenant au coeur du coeur :
   — Ecrire le processus combinatoire qui recoit une instruction et la décompose : opcode, r1,o,r2 etc...
   — Utiliser un branchement VHDL (case ... when) pour effectuer les modifications de registres (ADD,SUB,...)
   — Le mécanisme précédent ne suffit pas...Pourquoi ? Que faut-il ajouter ? Proposer un premier codage.

11.2 Modèle de programmation et jeu d'instructions

Pour rappel, notre processeur est de type RISC : reduced instruction set. Il dispose de deux mémoires distinctes, que l'on pourra considérer comme physiquement séparées : la mémoire d'instruction et la mémoire de données. Le processeur possède également une batterie de registres de travail, au nombre de 32. Chacun de ces registres est codé sur 32 bits. Le jeu d'instructions est rappelé dans le tableau suivant.

11.3 Simulateur de jeu d'instruction ou *ISS*

Un simulateur de jeu d'instruction, écrit en Ruby, est proposé ici. Bien entendu, Ruby n'est pas le premier choix instinctif pour ce genre de simulateur : il serait a priori plus intéressant de recourir à un langage compilé, comme le C. Néanmoins, Ruby possède de très nombreux avantages, dont la concision. Par ailleurs, notons que le langage Crystal, développé depuis peu, tend aujourd'hui à donner à Ruby la même performance que le C, avec l'Agilité de Ruby en prime.

```
require 'pp'
require_relative 'opcodes'
require_relative 'eda_utils'

class ISS

  def initialize
    puts "ISS/VM for 4.5"
  end

  def apply filename
    load_in_memory(filename)
    run
  end

  def load_in_memory filename
    @mem=[]
    IO.readlines(filename).each do |line|
      addr,data=line.split(" ").collect{|e| e.to_i(16)}
      @mem[addr]=data
    end
    show_mem
  end

  def show_regs
```

```
26      @reg.each_with_index do |v,regnum|
27        puts "reg#{regnum} = #{v}"
28      end
29    end
30
31    def show_mem
32      @mem.each_with_index do |code,idx|
33        puts "0x#{idx.to_s(16).rjust(8,'0')} 0x#{code.to_s(16).rjust(8,'0')}"
34      end
35    end
36
37    def init
38      @pc=0
39      @running=true
40      @reg=Array.new(32,0)
41      @data=Array.new(1024,0) #1 Ko
42    end
43
44    def decode code
45      #puts code.to_s(2).rjust(32,'0')
46      opcode=code.bit_field(31..27)
47
48      r1,flag,o,r2=extract [26..22,21..21,20..5,4..0],code
49      addr=@reg[r1]+(flag==0 ? o : @reg[o])
50
51      puts OPCODE.invert[opcode]
52
53      case opcode
54      when OPCODE[:add]
55        @reg[r2]=@reg[r1] + (flag==0 ? o : @reg[o]) if r2!=0
56      when OPCODE[:sub]
57        @reg[r2]=@reg[r1] - (flag==0 ? o : @reg[o]) if r2!=0
58      when OPCODE[:mul]
59        @reg[r2]=@reg[r1] * (flag==0 ? o : @reg[o]) if r2!=0
60      when OPCODE[:div]
61        @reg[r2]=@reg[r1] / (flag==0 ? o : @reg[o]) if r2!=0
62      when OPCODE[:and]
63        @reg[r2]=@reg[r1] & (flag==0 ? o : @reg[o]) if r2!=0
64      when OPCODE[:or]
65        @reg[r2]=@reg[r1] | (flag==0 ? o : @reg[o]) if r2!=0
66      when OPCODE[:xor]
67        @reg[r2]=@reg[r1] ^ (flag==0 ? o : @reg[o]) if r2!=0
68      when OPCODE[:shl]
69        @reg[r2]=@reg[r1] << (flag==0 ? o : @reg[o]) if r2!=0
70      when OPCODE[:slt]
71        @reg[r2]=((@reg[r1] < (flag==0 ? o : @reg[o])) ? 1 : 0) if r2!=0
72      when OPCODE[:sle]
73        @reg[r2]=((@reg[r1] <= (flag==0 ? o : @reg[o])) ? 1 : 0)  if r2!=0
74      when OPCODE[:seq]
75        @reg[r2]=(@reg[r1] == ((flag==0 ? o : @reg[o])) ? 1 : 0) if r2!=0
76      when OPCODE[:load]
77        @reg[r2]=@data[addr]  if r2!=0
78      when OPCODE[:store]
79        @data[addr]=@reg[r2] if r2!=0
80      when OPCODE[:jmp]
81        flag_jmp,o_jump,r_jmp = extract [26..26,25..5,4..0],code
82        addr = flag_jmp==1 ? @reg[o_jump] : o_jump
83        @reg[r_jmp] = @pc+1 if r_jmp!=0
84        return addr
85      when OPCODE[:braz]
86        r,a=extract [26..22,21..0],code
87        return a if @reg[r]==0
88      when OPCODE[:branz]
89        return a if @reg[r]!=0
90      when OPCODE[:scall]
91        n=extract [26..0],code
92        n=n.shift
93        case n
94        when 0
95          puts "scall 0 : read an integer"
96          @reg[1]=$stdin.gets.chomp.to_i
97        when 1
98          puts "scall 1 : write"
99          puts @reg[1]
```

```
100         else
101           raise "scall #{n} unknown"
102         end
103       when OPCODE[:stop]
104         abort
105       else
106         puts "unknown opcode"
107         abort
108       end
109       return nil
110     end
111
112     def extract_fields,code
113       fields.collect{|field| code.bit_field(field)}
114     end
115
116     def run
117       init
118       while @running
119         puts "fetching @ #{@pc}"
120         key=$stdin.gets.chomp
121         code=@mem[@pc]
122         next_addr=decode(code)
123         @pc= next_addr || @pc+1
124         show_regs
125       end
126     end
127 end
128
129 ISS.new.apply(ARGV.first)
```

11.4 Architecture non-pipelinée

Nous savons que le chemin de données d'un un processeur est généralement découpé en différents étages de *pipeline* : pendant qu'un étage réalise une tâche realtive à une instruction n, l'étage suivant est en train de réaliser une autre tâche, relative à l'instruction précédente $n-1$ etc... Cette découpe du travail permet d'augmenter la fréquence du processeur, en isolant par des registres les parties combinatoires afférentes aux différents étages de traitement.

Notre réalisation est plus modeste : nous ne cherchons pas à pipeliner notre processeur. C'est une tâche intéressante, mais elle demande un peu de mise au point. On se cantonnera ici à une architecture organisée autour d'un seul étage de traitement. Ce principe est représenté sur la figure 11.1. A partir de la seule instruction, issue de la mémoire d'instruction, on conçoit une logique combinatoire de décodage et d'exécution de l'instruction. Cette exécution conduit à modifier soit les registres de travail $r_0, \ldots, r_{31}$, soit la mémoire de données, soit le PC (programe counter), soit bien entendu plusieurs de ces élements, simultanément.

La réalisation du processeur nous montre également qu'il est nécessaire d'adjoindre une notion d'*état* du processeur : nous avons adjoint un registre d'état, qui permet de connaître l'état courant de la machine d'états finis sous jacente. Cette FSM est nécessaire pour une instruction : l'instruction *load* nécessite, dans l'instant courant, de générer une addresse vers la mémoire de données, puis, dans l'état suivant, de stocker cette donnée. On rappelle que la mémoire de données est également synchrone, ce qui explique ce cycle d'attente. Une mémoire asynchrone aurait été parfaitement possible sur FPGA, mais conduit à synthétiser cette mémoire à l'aide des registres des CLB, c'est à dire de manière *très distribuée* sur la puce : cette manière de procéder aurait immanquablement consommer des ressources précieuses. A l'inverse, le recours aux mémoires synchrones conduit la synthèse à inférer des *block ram*, beaucoup plus denses.

11.5 Banc de test

Le banc de test VHDL est exposé dans le listing suivant. Il permet en premier lieu de lire un fichier ASCII qui contient les addresses et instructions exprimées en hexdécimal. La lecture de tels nombres

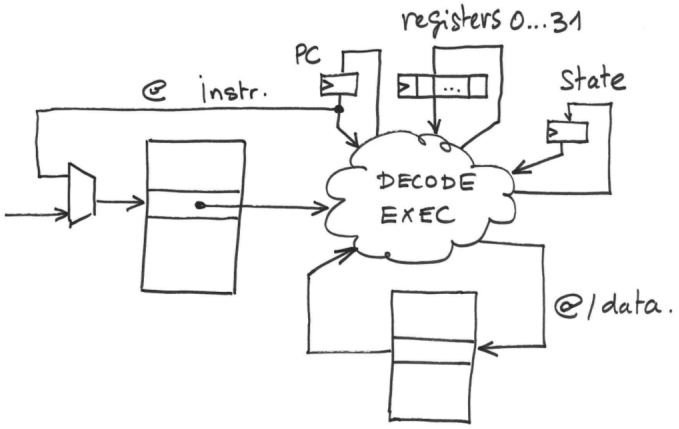


FIGURE 11.1 – Principe architectural du softcore non pipeliné.

est simplifiée par l'utilisation de la bibliothèque IEEE "Synopsys" appelée *std_logic_textio*. Puis le banc de test émet un signal *go* qui démarre le processeur : c'est le *boot* de notre machine !

11.6 Synthèse du softcore

Bibliographie

Annexes

Chapitre 12

Codage d'automates en C

12.1 Recours à une table de lookup

Ce premier code (dû à John Santic) utilise eune table de lookup : cette table liste les fonctions appelables lorsque la machine se trouve dans un état donné et qu'elle reçoit un événement donné. La fonction de transition est encodé dans chacune de ces fonctions.

```
/* Define the states and events. If your state machine program has multiple
source files, you would probably want to put these definitions in an "include"
file and #include it in each source file. This is because the action
procedures need to update current_state, and so need access to the state
definitions. */

enum states { STATE_1, STATE_2, STATE_3, MAX_STATES } current_state;
enum events { EVENT_1, EVENT_2, MAX_EVENTS } new_event;

/* Provide the fuction prototypes for each action procedure. In a real
program, you might have a separate source file for the action procedures of
each state. Then you could create a .h file for each of the source files,
and put the function prototypes for the source file in the .h file. Instead
of listing the prototypes here, you would just #include the .h files. */

void action_s1_e1 (void);
void action_s1_e2 (void);
void action_s2_e1 (void);
void action_s2_e2 (void);
void action_s3_e1 (void);
void action_s3_e2 (void);
enum events get_new_event (void);

/* Define the state/event lookup table. The state/event order must be the
same as the enum definitions. Also, the arrays must be completely filled -
don't leave out any events/states. If a particular event should be ignored in
a particular state, just call a "do-nothing" function. */

void (*const state_table [MAX_STATES][MAX_EVENTS]) (void) = {

    { action_s1_e1, action_s1_e2 }, /* procedures for state 1 */
    { action_s2_e1, action_s2_e2 }, /* procedures for state 2 */
    { action_s3_e1, action_s3_e2 }  /* procedures for state 3 */
};

/* This is the heart of the state machine - where you execute the proper
action procedure based on the new event you have to process and your current
state. It's important to make sure the new event and current state are
valid, because unlike "switch" statements, the lookup table method has no
"default" case to catch out-of-range values. With a lookup table,
out-of-range values cause the program to crash! */

void main (void)
{
    new_event = get_new_event (); /* get the next event to process */

    if (((new_event >= 0) && (new_event < MAX_EVENTS))
    && ((current_state >= 0) && (current_state < MAX_STATES))) {

```

```
50          state_table [current_state][new_event] (); /* call the action procedure */
51
52      } else {
53
54          /* invalid event/state - handle appropriately */
55      }
56  }
57
58  /* In an action procedure, you do whatever processing is required for the
59  particular event in the particular state. Among other things, you might have
60  to set a new state. */
61
62  void action_s1_e1 (void)
63  {
64      /* do some processing here */
65
66      current_state = STATE_2; /* set new state, if necessary */
67  }
68
69  void action_s1_e2 (void) {}  /* other action procedures */
70  void action_s2_e1 (void) {}
71  void action_s2_e2 (void) {}
72  void action_s3_e1 (void) {}
73  void action_s3_e2 (void) {}
74
75  /* Return the next event to process - how this works depends on your
76  application. */
77
78  enum events get_new_event (void)
79  {
80      return EVENT_1;
81  }
```

12.2 Exemple 2