Algorithmes arithmétiques : étude de la consommation d'énergie

Christophe RIOLO

24 juillet 2009

Résumé

Le but du stage effectué était de mettre en œuvre un protocole expérimental visant à mesurer physiquement la consommation énergétique d'algorithmes arithmétiques implémentés sur carte FPGA au moyen du langage de description matériel VHDL. Nous allons donc présenter ledit protocole, après avoir présenté les algorithmes choisis, puis nous analyserons les résultats.

Je remercie les membres de l'équipe CAIRN Lannion qui m'ont accueilli durant mon stage. Remerciements particuliers à Arnaud Tisserand, mon maître de stage, pour son encadrement et son aide durant mon stage ainsi que pour l'écriture du rapport, et à Arnaud Carer pour son aide pour la mise en place et la manipulation du matériel.

Table des matières

1 Introduction

Les calculs arithmétiques sont centraux dans le travail d'un processeur, c'est pourquoi il est important de chercher à les optimiser. Durant des années, les soucis principaux en architecture étaient la rapidité et la surface (miniaturisation). L'étude et l'optimisation de la consommation énergétique du matériel sont venus plus tardivement. Cette optique de baisse de la consommation rentre bien entendu parfaitement dans l'optique du *Green Computing* (le développement durable en informatique), mais est également cruciale dans le cas des systèmes embarqués.

L'équipe CAIRN, équipe de l'IRISA [1] bilocalisée à l'université de Rennes 1 et à l'ENSSAT à Lannion, a pour but d'étudier des moyens d'optimiser la programmation de circuits reconfigurables autant sur le plan matériel que logiciel, en vue de réduire la consommation en énergie, le temps de configuration des circuits, ou encore la complexité de la procédure, en se ramenant à un plus haut niveau. Parmi ces optimisations, certaines sont liées à des activités spécifiques (traitement du signal, cryptographie, ...), et utilisent des opérateurs particuliers ayant plusieurs variantes algorithmiques, dont il faut décider laquelle utiliser [2].

Nous commencerons par présenter les outils de travail, du VHDL, le langage utilisé à la carte FPGA en passant par les logiciels utilisés pour traiter le code et programmer le FPGA, pour ensuite décrire comment nous avons mis en place un protocole expérimental permettant de mesurer physiquement la consommation en énergie de l'algorithme de multiplication. Nous verrons enfin les résultats obtenus et le traitement effectué pour obtenir les informations que nous voulions.

2 Les prérequis

2.1 Les outils

2.1.1 La carte FPGA

Les cartes FPGA (Field Programmable Gate Array) sont des cartes programmables, contenant un processeur et de multiples ports externes (USB, Ethernet, ...) ou internes (signaux d'horloge, ...). Pour leur flexibilité, elles sont utilisées à des fins pédagogiques, pour tester des implémentations matérielles, ce qui est notre cas ici, ou encore dans des produits industriels [3].

2.1.2 Les logiciels

La carte que nous avons utilisé était une carte ML423 munie d'un FPGA Xilinx Virtex 4 ; nous avons donc en conséquence utilisé la suite logicielle de Xilinx, ISE, pour programmer le FPGA.

Cette étape comporte de multiples facettes. L'une d'elle est la *simulation* de la description matérielle écrit en VHDL (langage décrit à la section suivante) qui permet de vérifier que

[1] Institut de Recherche en Informatique et Systèmes Aléatoires.

[2] Éventuellement dynamiquement : l'activité du matériel n'est pas nécessairement connue à l'avance.

[3] Tels que des routeurs par exemple.

notre programme effectue bien l'action demandée.
Ensuite, le programme testé peut être *synthétisé*, c'est à dire que le programme est analysé pour pouvoir être implémenté matériellement sur la carte FPGA, ce qui est l'équivalent de la compilation pour le matériel.
Enfin, le programme, synthétisé sous forme de *bitstream* peut être chargé dans la carte FPGA.

2.1.3 VHDL, un langage de description matériel

Le langage VHDL est un langage permettant de décrire le fonctionnement d'une architecture matérielle (ou plus si affinités : ce langage est très générique). Ses seuls véritables concurrent dans la description matérielle sont Verilog et SystemC. Il est cependant à noter que sa puissance fait que tout n'est pas synthétisable, ou n'est pas bien synthétisable, et il faut donc se restreindre à une sous partie du langage.

Ce langage permet, entre autres, de définir des composants sous forme d'un bloc, avec des *ports* d'entrée et de sortie, exactement comme dans les architecture matérielles, et contenant des *signaux* (bus ou fils par exemple), des calculs, ou d'autres blocs. Il est donc bon de découper un programme VHDL en un grand nombre de blocs réutilisables, comme on découpe un programme classique en fonctions.

2.2 Algorithmes de multiplication

L'opération arithmétique qui a été choisie, dans un souci de simplicité, a été la multiplication. En effet, les algorithmes de multiplication sont plus simples et donc plus faciles à implémenter en VHDL lorsque l'on débute. Le protocole a toutefois été étudié de façon à être rapidement adapté à d'autres opérations (dans la mesure du temps d'implémentation desdites opérations).

L'étude est une étude comparative : il convenait donc d'avoir au moins deux versions de la multiplication, et ont été choisies la multiplication de base, le "$a \times b$" en VHDL, synthétisée au mieux par ISE, et la multiplication de Karatsuba.

La multiplication de Karatsuba

Il convient de présenter cet algorithme. Il repose sur une factorisation particulière de la multiplication de deux entiers, qui est la suivante :

$$\begin{aligned}(a \times 2^k + b) \times (c \times 2^k + d) &= ac \times 2^{2k} + (ad + bc) \times 2^k + bd \\ &= ac \times 2^{2k} + (ac + bd - (a-b)(c-d)) \times 2^k + bd\end{aligned}$$

Ainsi écrite, la multiplication ne nécessite que 3 sous multiplications : $a \times c$, $b \times d$ et $(a-b) \times (c-d)$, au lieu des 4 de la première ligne. Ceci se décrit schématiquement sous la forme vue à la figure 1. Il est à noter que ce découpage en blocs se prête très bien à une description en VHDL.

Habituellement, le cas de base de la multiplication de Karatsuba est le chiffre, c'est à dire dans notre cas le bit. Cependant, la carte FPGA contient des blocs dédiés au calcul et notamment à la multiplication, les blocs DSP48. Ce sont eux qui effectuent la multiplication au cas

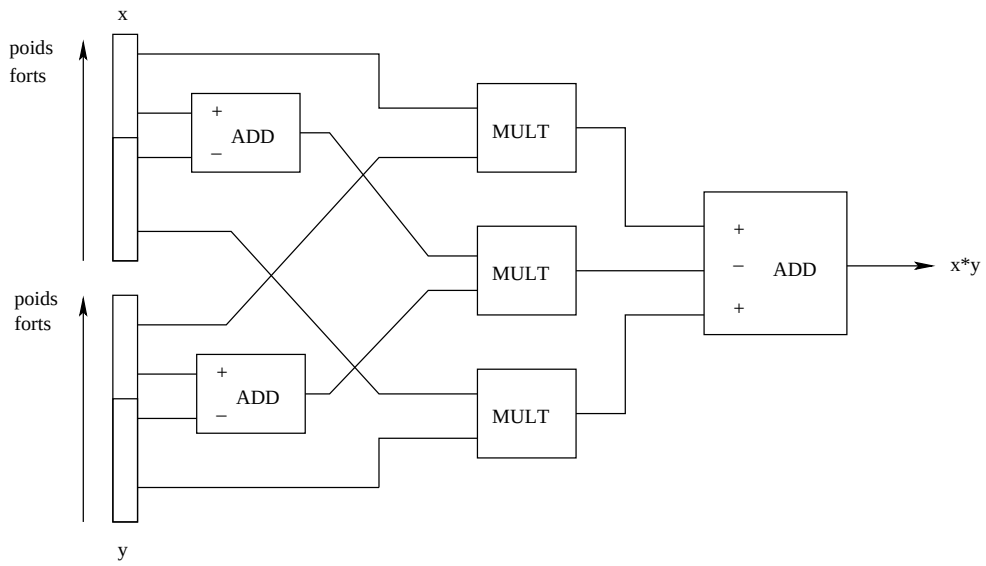


FIG. 1 – La multiplication de Karatsuba

de base [4], et on peut donc leur confier à multiplier des nombres sur plusieurs bits, ce qui a pour effet de diminuer le nombre de niveau récursifs [5]. On cherchera par la suite à voir l'impact du nombre de niveau de récursion, et donc de la taille du cas de base. Nous abrègerons par la suite "algorithme de Karatsuba de taille de cas de base n bits" en "Karatsuba base n".

3 Description de l'expérience

Notre but était d'implémenter l'algorithme dans la carte FPGA et de mesurer physiquement la consommation d'énergie induite par le bloc multiplieur. Nous allons donc voir la partie logicielle chargée sur la carte et les mesures à proprement parler.

3.1 La mise en contexte du multiplieur

Il est impossible d'implémenter une multiplication dans la carte FPGA sans lui donner des entrées, et de rediriger sa sortie, d'autant plus que les synthétiseurs VHDL sont suffisamment performants pour supprimer les éléments superfétatoires. Comme notre multiplication n'a pas vraiment de contexte, il a fallu ruser pour lui en donner un et ainsi duper le synthétiseur.

De plus, comme le coût énergétique d'une seule multiplication, même 32 bits comme dans notre cas, est relativement faible, il a fallu dupliquer les multiplieurs [6]. Il a suffi de met-

[4] Sauf bien sûr si l'on définit soi même le cas de base.

[5] Un multiplieur de Karatsuba en contient 3 autres de taille inférieure, leur nombre évolue donc en 3^k, où k est le nombre de niveaux de récursion ; diminuer ce nombre est donc un intéressant gain de surface et éventuellement de consommation d'énergie.

[6] Ce qui a encore amené à une ruse : le synthétiseur voyait 10 fois le même multiplieur, alors il n'en faisait qu'un et prenait sa sortie autant de fois qu'il le fallait. Pour contourner ce problème, il a suffi de numéroter les multiplieurs avec un paramètre générique : ainsi ils devenaient différents.

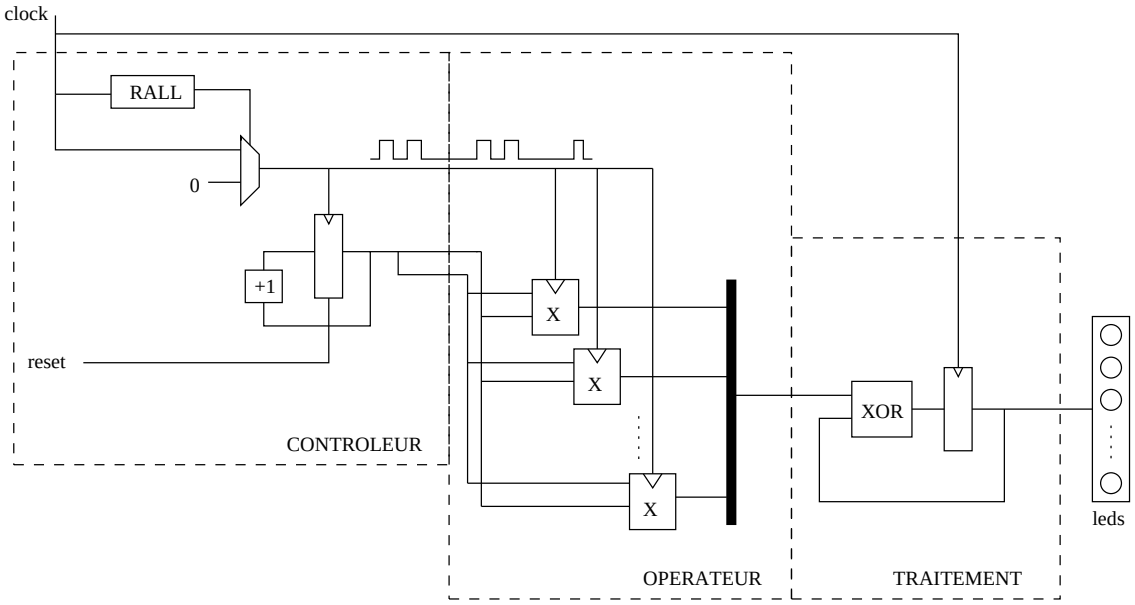


FIG. 2 – L'ensemble du programme chargé sur la carte FPGA

tre cinq multiplieurs pour observer une augmentation significative de l'intensité en période de calcul [7].

À la figure 2, on peut voir la structure générale de l'architecture matérielle mise en place pour le protocole expérimental. Nous allons détailler les fonctions des trois blocs mis en valeur.

3.1.1 L'opérateur

Ce bloc est le cœur du programme, c'est celui qui effectue les multiplications des entrées au rythme donné par l'entrée d'horloge, et qui renvoie le résultat en sortie. Les multiplications sont des multiplications p bits, où p est un paramètre *générique*, c'est à dire une variable évaluée lors de la simulation ou de la synthèse.

Afin de comparer la consommation dynamique ainsi que la consommation statique, il faut pouvoir effectuer des mesures durant des phases de calcul ainsi que des phases de repos, et ce bien évidemment sans changer les conditions expérimentales. Pour cela, plutôt que d'envoyer un signal d'horloge régulier, on envoie un signal alternant des phases d'oscillation et des phases stables, signal généré par le bloc contrôleur.

3.1.2 Le contrôleur

Ce bloc ne prend que deux entrées : un signal d'horloge "classique", généré par un oscillateur à quartz et filtré par un bloc spécialisé que je ne détaillerai pas ici [8], et une entrée de remise à zéro.

[7] Augmenter le nombre de multiplieurs augmente a priori la précision de la mesure aussi, par une plus faible sensibilité au bruit, mais alourdit beaucoup l'étape de synthèse, en temps et en mémoire nécessaire.

[8] se reporter au manuel d'utilisateur de Xilinx [1] pour plus de précisions.

Un multiplexeur qui prend en entrée le signal d'horloge initial et la valeur constante 0 actionné par un signal d'horloge ralenti permet d'obtenir une sortie alternant à la fréquence donnée par le ralentisseur entre ce 0 et le signal d'horloge. C'est comme cela qu'est généré le signal d'horloge dont nous parlions à la section précédente.

Parallèlement, un autre compteur fournit les entrées des multiplieurs, en étant branché sur les deux entrées du bloc opérateur. On voit qu'ici, c'est en réalité un carré qui est effectué, mais on peut très bien imaginer un bloc de contrôle plus complexe qui envoie d'autres entrées aux multiplieurs [9].

3.1.3 Le bloc de traitement

Ce dernier bloc est celui qui exploite la sortie du bloc opérateur, et qui redirige ce qui est calculé vers les LEDs. En effet, sans ce traitement, le synthétiseur verrait des blocs et signaux inutiles et les supprimerait sans autre forme de procès (et ici, il ne resterait plus rien, ce qui serait fâcheux). Son utilité se limitant à éviter la suppression du reste du circuit, il est inutile de plus détailler ce bloc.

3.2 Les mesures

Le protocole est simple : la carte ML423 permet d'alimenter le FPGA par une alimentation extérieure. Nous avons donc exploité cette possibilité en alimentant le FPGA avec une alimentation de laboratoire instrumentée Agilent N6705A [2] qui a permet d'analyser le courant débité avec l'oscilloscope intégré.

La tension étant fixée à 1,2 V, la consommation électrique est proportionnelle à l'intensité traversant le circuit. Avec l'alimentation, nous avons donc suivi le courant et extrait la valeur moyenne de celui-ci durant les phases de calcul et les phases d'inactivité pour les différents multiplieurs (les résultats étant répertoriés plus loin à la figure 5). La faculté d'enregistrement des données dans un fichier CSV a également permis de conserver les oscillogrammes observables aux figures 3 et 4.

4 Les résultats

4.1 Profil de l'intensité

Le tracé de la variation d'intensité en fonction du temps montre bien la variation de consommation entre les phases de calcul et les phases d'inactivité. Les critères à regarder sont la consommation statique [10] (périodes d'inactivité), la consommation totale en activité, et la consommation dynamique, la différence entre les deux.

On observe à la figure 3 que la consommation statique ainsi que la consommation dynamique augmentent avec le nombre de niveau de récursion dans l'algorithme de Karatsuba ; pour la suite de l'étude, nous nous contenterons donc d'un seul niveau de récursion,

[9] C'est également ici que l'on changera la production des entrées de l'opérateur si l'on veut tester une autre opération que la multiplication.

[10] Consommation qui a lieu même hors des périodes d'activité et qui peut être due par exemple à des courants de fuite.

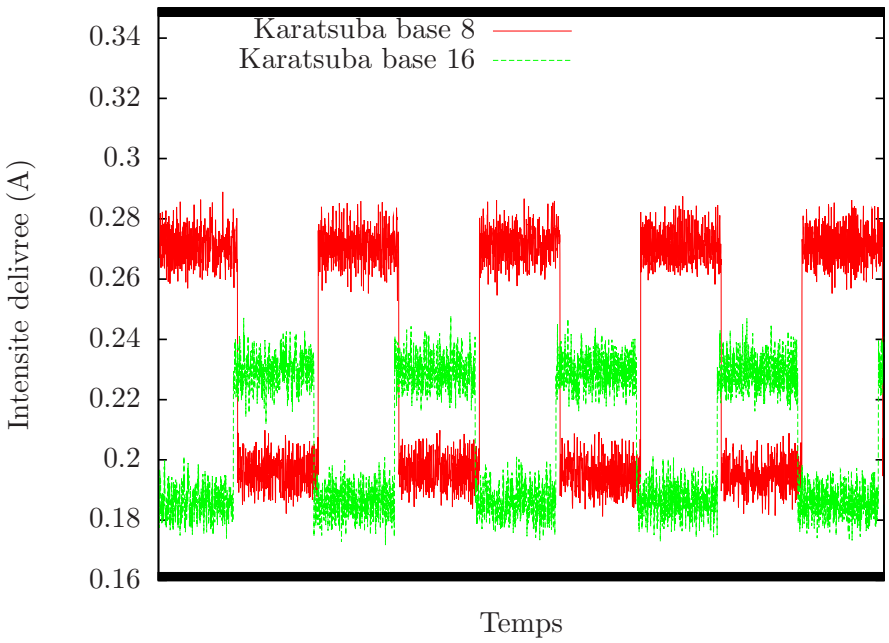


FIG. 3 – Impact du niveau de récursion dans l'algorithme de Karatsuba

c'est à dire d'un nombre de bits pour le cas de base de 16 bits dans le cas de la multiplication 32 bits.

De même, si l'on compare la multiplication de base de VHDL avec et sans les blocs DSP48 [11] on observe (fig. 4) que les blocs DSP48 permettent une diminution de la consommation dynamique mais résulte en une augmentation de la consommation statique. On peut voir à la figure 5 une comparaison plus précise des différents multiplieurs : Karatsuba, avec blocs DSP48 et sans blocs DSP48.

4.2 Consommations des deux multiplieurs

En faisant varier le nombre de multiplieurs dans le circuit, et en étudiant la variation de la consommation, il est plus aisé de déterminer la consommation d'un seul multiplieur. On observe en effet que la consommation électrique augmente linéairement en fonction du nombre de multiplieurs, et le coefficient directeur de la droite est la consommation d'un seul multiplieur (alors que le terme constant est ce que consomme le reste du circuit). En faisant une régression linéaire sur les mesures, nous avons obtenu les résultats que l'on peut observer à la fugure 6 (les mesures elles mêmes sont regroupées à la figure 5). Il en ressort que si le multiplieur avec les blocs DSP48 a une consommation dynamique plus faible que Karatsuba, ce dernier gagne en matière de consommation statique. La consommation totale des deux multiplieurs en revanche est sensiblement la même. On peut donc penser que les deux variantes sauraient trouver des applications différentes. Cette étude a été faite pour la multiplication 32 bits, mais on pourrait étudier de même d'autres tailles d'entrées [12].

[11] Blocs de calcul dédiés.

[12] Une brève étude sur 128 bits semble donner des résultats similaires (cf figure 5).

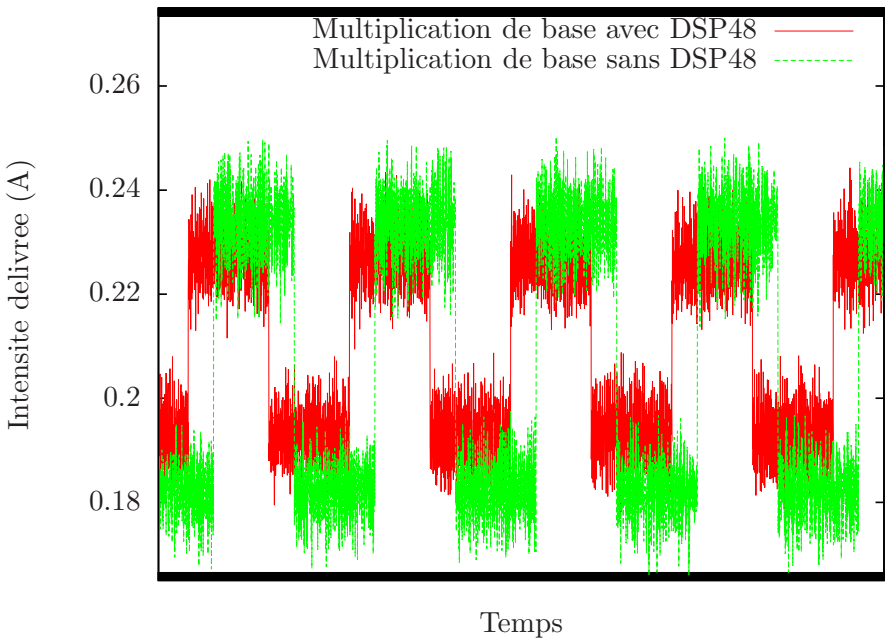


FIG. 4 – Impact de l'utilisation des blocs DSP48

4.3 Taille et vitesse

Ces facteurs n'ont pas été mesurés mais viennent du rapport de synthèse donné par ISE. Le facteur taille est difficile à comparer du fait que les blocs DSP48 équivaudraient eux mêmes à plusieurs slices [13], et il faudrait prendre en compte ceci pour faire une véritable comparaison. En conséquence, on peut juste dire que la multiplication avec DSP48 utilise moins de slices mais plus de blocs dédiés.

En ce qui concerne la vitesse, la multiplication de Karatsuba nécessite une période minimale de 17 ns alors que la multiplication avec DSP48 nécessite elle 25 ns, ce qui représente un gain de temps de 30% approximativement.

5 Conclusion

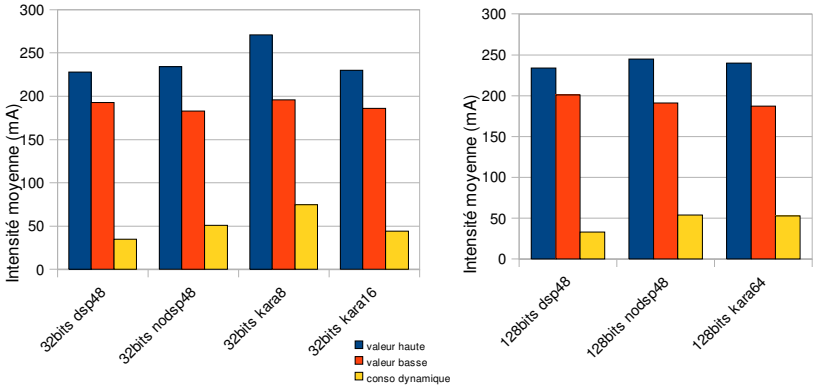
Au cours de ce stage, j'ai pu développer et tester l'environnement de mesure de l'intensité d'un opérateur. Cette architecture a été faite pour de la multiplication mais peut être adaptée pour d'autres opérateurs, d'autres tailles. Si les opérateurs de multiplication ne sont pas forcément critiques, cet environnement pourra par la suite être utilisé pour des opérateurs plus spécifiques à d'autres domaines comme le traitement du signal et la cryptographie [14].

[13] La carte FPGA est composée de slices reprogrammables, assimilables à l'unité de surface dans de telles cartes.

[14] Ce domaine utilisant généralement des entrées de grande taille.

Feuille1

Intensités (mA)	valeur haute	valeur basse	conso dynamique
5 DSP48	217	198	19
10 DSP48	230	198	32
15 DSP48	266	217	49
20 DSP48	299	225	74
25 DSP48	312	233	79
30 DSP48	342	232	110
5 Kara16	213	188	25
10 Kara16	232	190	42
15 Kara16	269	206	63
20 Kara16	289	204	85
25 Kara16	318	205	113
30 Kara16	336	212	124
32bits dsp48	228	193	35
32bits nodsp48	234	183	51
32bits kara8	271	196	75
32bits kara16	230	186	44
128bits dsp48	234	201	33
128bits nodsp48	245	191	54
128bits kara64	240	187	53



Page 1

FIG. 5 – Mesures et graphiques généraux

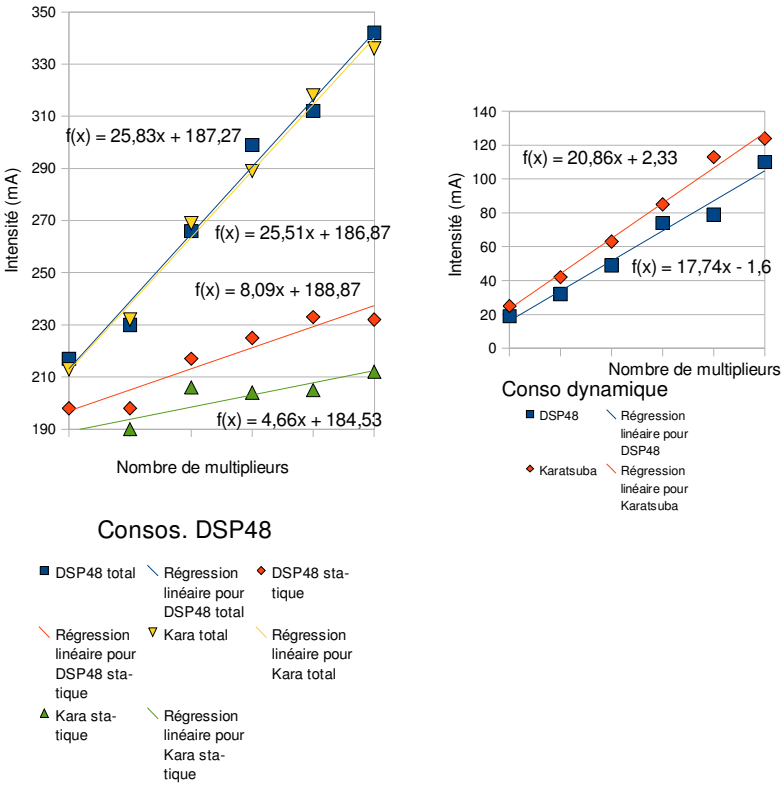


FIG. 6 – Intensité en fonction du nombre de multiplieurs

Références

[1] Xilinx Virtex4 User Guide : http://www.xilinx.com/support/documentation/user_guides/ug070.pdf

[2] Agilent Technologies N6705A : http://www.home.agilent.com/agilent/product.jspx?nid=-35558.656338.00&cc=US&lc=eng

[3] Référence de la FPGA : *ML42X User Guide* (http://www.xilinx.com/support/documentation/boards_and_kits/ug087.pdf)

[4] *VHDL*, R. Airiau, J.-M. Bergé, V. Olive, J. Rouillard, Presses polytechniques et universitaires romandes.

[5] *VHDL : méthodologie de design et techniques avancées*, Thierry Schneider, éditions Dunod.

Annexes : Code Source

A principal.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    15:47:01 05/27/2009
-- Design Name:
-- Module Name:    principal - simple
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
```

```
--use UNISIM.VComponents.all;

use work.all;

entity principal is
 generic( size : natural := 32 ; n : natural := 10 ; niveau_limite : natural := 1);
    Port ( clkin0,reset : in  STD_LOGIC;
           leds : out  STD_LOGIC_VECTOR (19 downto 0));
end principal;

architecture simple of principal is

component clock_manager
    Port ( clk_in : in  STD_LOGIC;
           clk_out : out  STD_LOGIC);
end component;
for all : clock_manager use entity clock_manager(dcm_arch);

component ctrl
    generic( size : natural);
    Port ( clk_in : in  STD_LOGIC;
  clk_out : out  STD_LOGIC;
           reset : in  STD_LOGIC;
           s0 : out  STD_LOGIC_VECTOR(size-1 downto 0);
           s1 : out  STD_LOGIC_VECTOR(size-1 downto 0));
end component;
for all : ctrl use entity ctrl(simple);

component operateur
 generic(
size : natural;
n : natural;
niveau_limite : natural := 1
);
    Port (
clk : in  STD_LOGIC;
      a : in  STD_LOGIC_VECTOR (size-1 downto 0);
      b : in  STD_LOGIC_VECTOR (size-1 downto 0);
      p : out  STD_LOGIC_VECTOR (2*size-1 downto 0)
);
end component;
for all : operateur use entity operateur(simple);

component post_traitement
generic (size : natural);
    Port ( valeur : in  STD_LOGIC_VECTOR (2*size-1 downto 0);
  clk : in STD_LOGIC;
```

```
           leds : out  STD_LOGIC_VECTOR (19 downto 0));
end component;
for all : post_traitement use entity post_traitement(dummy);

signal clk0, clk1 : std_logic;
signal a,b : std_logic_vector(size-1 downto 0);
signal p : STD_LOGIC_VECTOR(2*size-1 downto 0);

begin

-- on relie les différents éléments (cf schema.pdf, le signal d'horloge étant
-- généré par clock_manager)
CL : clock_manager port map (clk_in => clkin0, clk_out => clk0);
CT : ctrl  generic map (size => size) port map (clk_in => clk0, clk_out => clk1, reset => r
OP : operateur generic map (size => size, n => n, niveau_limite => niveau_limite) port map
TR : post_traitement generic map (size => size) port map (valeur => p, clk => clk0, leds =>

end simple;
```

B ctrl.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    15:20:56 05/27/2009
-- Design Name:
-- Module Name:    ctrl - simple
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ctrl is
    generic( size : natural := 10);
    Port ( clk_in : in  STD_LOGIC;
  clk_out : out  STD_LOGIC;
           reset : in  STD_LOGIC;
           s0 : out  STD_LOGIC_VECTOR(size-1 downto 0);
           s1 : out  STD_LOGIC_VECTOR(size-1 downto 0));
end ctrl;

use work.registre_rst;

-- le but du contrôleur est de générer les entrées de l'opération à tester
-- (e.g. générer deux entiers à multiplier entre eux) ainsi que de générer le
-- signal d'horloge irrégulier qui permet d'alterner les phases de calcul et
-- les phases de repos.
```

```
architecture simple of ctrl is

component registre_rst
generic ( size : natural := 10);
port ( e : in  STD_LOGIC_VECTOR (size-1 downto 0);
        clk : in STD_LOGIC;
  rst : in STD_LOGIC;
          s : out  STD_LOGIC_VECTOR (size-1 downto 0));
end component;
for all : registre_rst use entity registre_rst(simple);

component rallentisseur
generic (n : natural := 25);
   port ( clk_in : in  STD_LOGIC;
          clk_out : out  STD_LOGIC);
end component;
for all : rallentisseur use entity rallentisseur(visible);

component multiplexeur
    Port ( a0 : in  STD_LOGIC;
           a1 : in  STD_LOGIC;
           sel : in  STD_LOGIC;
           b : out  STD_LOGIC);
end component;
for all : multiplexeur use entity multiplexeur(simple);

signal t1,t2 : STD_LOGIC_VECTOR (size-1 downto 0);
signal sel,clk_reg : STD_LOGIC;

begin

-- registre pour les entrées de l'opération
R : registre_rst
generic map ( size => size)
port map (
e => t2,
clk => clk_reg,
rst => reset,
s => t1
);

RALL : rallentisseur generic map (n => 25) port map (clk_in => clk_in, clk_out => sel);

M : multiplexeur port map (a0 => clk_in, a1 => '0', sel =>sel, b=>clk_reg);

-- signaux pour le compteur générant les entrées pour l'opération
t2 <= t1 + "1";
```

```
s0 <= t1;
s1 <= t1;

-- horloge irrégulière
clk_out <= clk_reg;

end simple;
```

C operateur.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    17:16:10 05/26/2009
-- Design Name:
-- Module Name:    operateur - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity operateur is
 generic(
size : natural := 10;
n : natural := 1;
niveau_limite : natural := 1
);
```

```
    Port (
clk : in  STD_LOGIC;
      a : in  STD_LOGIC_VECTOR (size-1 downto 0);
      b : in  STD_LOGIC_VECTOR (size-1 downto 0);
      p : out  STD_LOGIC_VECTOR (2*size-1 downto 0)
);
end operateur;

use work.multiplier;
use work.registre;

architecture simple of operateur is

component multiplier
  generic (
    total_bits : natural := 16;
 num_bits   : natural := 16;
 niveau_limite : natural := 1;
 numero : natural := 0
  );
  port (
    x : in  std_logic_vector(total_bits -1 downto 0) ;       -- operande 1
    y : in  std_logic_vector(total_bits -1 downto 0) ;       -- operande 2
    r : out std_logic_vector(2*total_bits -1 downto 0);       -- resultat
 clk : in std_logic);
end component;

component registre
generic(size : natural := 10);
port(
e : in std_logic_vector(size-1 downto 0);
clk : in std_logic;
s : out std_logic_vector(size-1 downto 0)
);
end component;


for all : multiplier use entity multiplier(kara);
for all : registre use entity registre(simple);

signal s0,e : std_logic_vector(2*n*size-1 downto 0);

begin
-- on génère les différents blocs multiplieurs
multi : for i in 1 to n generate
M : multiplier
  generic map (
```

```
    total_bits => size,
    num_bits => size,
    niveau_limite => niveau_limite,
    numero => i)                        -- ce paramètre sert à leurrer le
                                        -- synthétiseur pour le forcer à mettre
                                        -- plusieurs blocs
  port map (
    clk=>clk,
    x=>a,
    y=>b,
    r=>s0(2*i*size -1 downto 2*(i-1)*size));
end generate multi;

-- on retient toutes les sorties des multiplieurs dans un grand vecteur
e(2*size-1 downto 0) <= s0(2*size-1 downto 0);
andgen : for k in 2 to n generate
begin
  e(2*size*k-1 downto 2*size*(k-1)) <= s0(2*size*k-1 downto 2*size*(k-1)) and e(2*size*(k-1
end generate andgen;

p <= e(2*size*n-1 downto 2*size*(n-1));

end simple;
```

D karatsuba.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

entity karatsuba is
  generic (
    total_bits : natural := 32;
 num_bits   : natural := 16;
 niveau_limite : natural := 1
  );
  port (
    x : in  std_logic_vector(total_bits -1 downto 0) ;       -- operande 1
    y : in  std_logic_vector(total_bits -1 downto 0) ;       -- operande 2
    r : out std_logic_vector(total_bits -1 downto 0));       -- resultat

end karatsuba;

architecture recursive of karatsuba is

signal r0_0,r0_1,r1_0,r1_00,r1_01,r1_1,r1_10,r1_11,r2_0,r2_1 : std_logic_vector(  total_bit
signal r0,r1,r2  : std_logic_vector(total_bits -1 downto 0) ;
signal buff : std_logic_vector(total_bits -1 downto 0);
signal zeros : std_logic_vector(num_bits -1 downto 0);

begin
-- si on doit faire encore un niveau récursif on génère ce qui suit, sinon on
-- génèrera le cas de base situé plus bas. Pour l'algorithme récursif, cf karatsuba_pspdfte
recur : if num_bits > niveau_limite generate
r0_0(total_bits -1 downto num_bits/2) <= (others => '0');
r0_1(total_bits -1 downto num_bits/2) <= (others => '0');
r0_0(num_bits/2-1 downto 0) <= x(num_bits/2-1 downto 0);
r0_1(num_bits/2-1 downto 0) <= y(num_bits/2-1 downto 0);
mult_r0 : entity work.karatsuba
  generic map(
    total_bits => total_bits,
    num_bits => num_bits/2,
    niveau_limite => niveau_limite)
  port map (
    x =>r0_0,
    y => r0_1,
    r => r0);

r2_0(total_bits -1 downto num_bits/2) <= (others => '0');
```

```
r2_1(total_bits -1 downto num_bits/2) <= (others => '0');
r2_0(num_bits/2-1 downto 0) <= x(num_bits -1 downto num_bits/2);
r2_1(num_bits/2-1 downto 0) <= y(num_bits -1 downto num_bits/2);
mult_r2 : entity work.karatsuba
  generic map(
    total_bits => total_bits,
    num_bits => num_bits/2,
    niveau_limite => niveau_limite)
  port map (
    x =>r2_0,
    y => r2_1,
    r => r2);

r1_00(total_bits -1 downto num_bits/2) <= (others => '0');
r1_01(total_bits -1 downto num_bits/2) <= (others => '0');
r1_10(total_bits -1 downto num_bits/2) <= (others => '0');
r1_11(total_bits -1 downto num_bits/2) <= (others => '0');
r1_00(num_bits/2-1 downto 0) <= x(num_bits -1 downto num_bits/2);
r1_01(num_bits/2-1 downto 0) <= x(num_bits/2-1 downto 0);
r1_10(num_bits/2-1 downto 0) <= y(num_bits -1 downto num_bits/2);
r1_11(num_bits/2-1 downto 0) <= y(num_bits/2-1 downto 0);
r1_0 <= r1_00 - r1_01;
r1_1 <= r1_10 - r1_11;
mult_r1 : entity work.karatsuba
  generic map(
    total_bits => total_bits,
    num_bits => num_bits/2,
    niveau_limite => niveau_limite)
  port map (
    x =>r1_0,
    y => r1_1,
    r => r1);

zeros <= (others => '0');
r <= (r2(total_bits - num_bits -1 downto 0)&zeros) - (r1(total_bits - num_bits/2 -1 downto 0
end generate recur;

-- si on a atteint le cas de base, alors on fait la multiplication par défaut
-- de VHDL. L'algorithme de Karatsuba classique est obtenu avec niveau_limite
-- valant 1.
base : if num_bits <= niveau_limite generate
buff(num_bits*2 +1 downto 0) <= (x(num_bits downto 0)*y(num_bits downto 0));
buff(total_bits -1 downto num_bits*2 +2) <= (others => buff(num_bits*2 +1));
r <= buff;
end generate base;
end recursive;
```