LOGIQUE PROGRAMMABLE

Implanter des *fonctions DSP sur FPGA* de façon simple et efficace

L'objectif est de tirer le meilleur parti des FPGA Virtex-4 de Xilinx, orientés traitement du signal. Experte en conception dans ce domaine, la société MVD démontre, exemples à l'appui, qu'écrire soi-même le code VHDL assure une implantation optimale en performances des algorithmes DSP. La bonne surprise: cette écriture s'avère simple dès que la structure matérielle est bien assimilée.

Par Edgard Garcia, MVD

Spécialisée dans le traitement d'images et référence française en matière de FPGA Xilinx, MVD a été créée par Edgard Garcia. Depuis de nombreuses années, cette société conçoit des FPGA en langage VHDL pour tous types d'applications. Elle assure également des stages de formation pour les concepteurs de systèmes embarqués.

L'architecture des matrices de logique programmable Virtex-4 de Xilinx a été enrichie de structures optimisées pour l'implantation d'algorithmes de traitement du signal (DSP). Le bloc de base de ces structures, baptisé «slice» DSP48, est conçu pour effectuer toute une variété de fonctions mathématiques essentielles au traitement du signal telles que l'addition/soustraction ou la multiplication/accumulation (que nous appellerons aussi MAC). Pour chaque bloc DSP48, la dynamique de ces opérations (signées ou non) peut atteindre 48 bits. Sur la matrice, deux slices DSP48 et leur routage forment une tuile («tile» DSP48). Ces tuiles sont empilées sous forme de colonnes ce qui autorise un routage très performant d'une tuile à l'autre sans pénaliser le reste des ressources logiques classiques de la matrice. Les FPGA Virtex-4 peuvent abriter de 32 à 512 blocs DSP48 (*Electronique* n°151, p.19).
Associés à cette nouvelle structure matérielle pour le traitement du signal, des outils logiciels mis à disposition par Xilinx accompagnent l'utilisateur dans la conception de son système. C'est là que trois solutions se présentent à lui. Dans la première que nous qualifierons d'efficace mais fastidieuse, chaque bloc DSP48 est appelé individuellement et «instancié». Puis, par l'affectation de toute une mécanique d'attributs et de constantes, ce module est configuré pour réaliser la fonction de base souhaitée. L'aspect lourd et répétitif de cette méthode se révèle rapidement dès que l'algorithme à implanter croît en complexité.
A l'inverse, la deuxième solution traite le problème à un très haut niveau. Dans ce cas, l'utilisateur travaille avec des outils de type Matlab/Simulink puis des interfaces vers les outils d'implantation. Il spécifie la fonction désirée, par exemple un filtre FIR de caractéristiques données, programmables dans le logiciel de haut niveau. Ensuite, d'un simple «clic», la fonction est implantée. Néanmoins, l'aspect «haut niveau» de cette méthode maintient l'utilisateur éloigné des réalités de l'implantation physique. De ce fait, le résultat en termes de performances et de densité de ressources utilisées reste hors de portée du concepteur. Si le bilan final ne concorde pas parfaitement avec les exigences du cahier des charges, il lui est impossible d'intervenir et d'optimiser. La qualité et l'efficacité de ces outils amènent donc un confort indéniable à l'utilisateur mais certains points stratégiques de l'implantation risquent de lui échapper.
La troisième option s'adresse aux concepteurs expérimentés désirant tirer le meilleur profit de l'architecture des DSP48, tout en préservant la portabilité et la pérennité de leur conception. Pour cela, ils s'appuieront sur les toutes dernières avancées technologiques en matière de synthèse logique. Cette méthode fait l'objet de cet article. Consistant à écrire le code VHDL (ou Verilog) des fonctions souhaitées, elle agit à un niveau plus élevé que la première et plus décisionnel que

Exemples d'arrondi symétrique

Sortie du multiplieur (décimal)	Sortie du multiplexeur (binaire)	Valeur du registre C	Cin générée en interne	Multiplieur + C + Cin	Après troncation (binaire)	Après troncation (décimal)
2,4375	0010,0111	0000,0111	1	0010,1111	0010	2
2,5	0010,1000	0000,0111	1	0011,0000	0011	3
2,5625	0010,1001	0000,0111	1	0011,0001	0011	3
- 2,4375	1101,1001	0000,0111	0	1110,0000	1110	- 2
- 2,5	1101,1000	0000,0111	0	1101,1111	1101	- 3
- 2,5625	1101,0111	0000,0111	0	1101,1110	1101	- 3

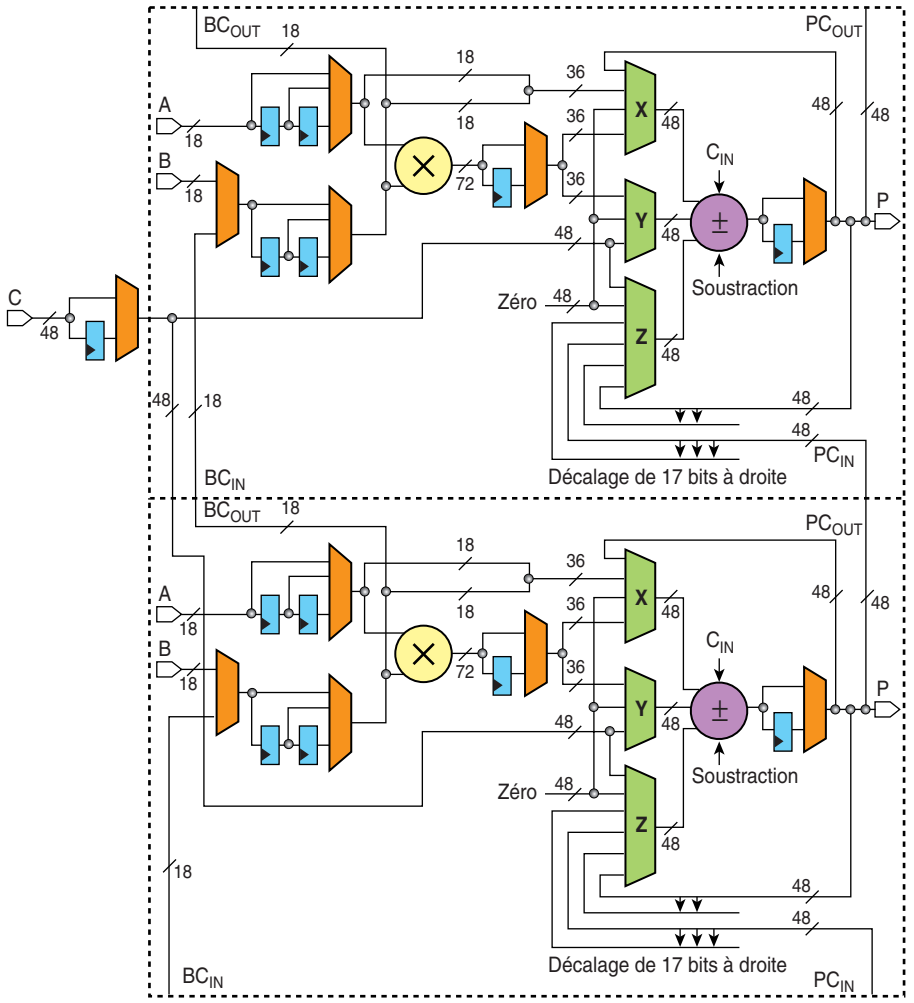
la deuxième. Son efficacité pour une optimisation des performances coule de source, par contre, la bonne surprise vient du fait que cette écriture en VHDL s'avère simple et élégante.
Enfin, si développer des algorithmes en VHDL (ou Verilog) est une bonne façon de pérenniser des conceptions, il est impératif que les résultats de la synthèse coïncident avec les performances requises. Cet article décrit comment écrire du code RTL pour bénéficier de tous les avantages des blocs DSP48. La démonstration repose sur des exemples choisis parmi ceux présentés dans le guide de l'utilisateur, référencé « XtremeDSP design considerations user guide », accessible sur le site de Xilinx. Notons également que le principe fondamental de la méthode se résume en cette phrase : plus le code RTL est simple, plus efficaces seront les résultats de la synthèse.

Bien connaître l'architecture matérielle

Un bloc DSP48 est composé d'un multiplieur à deux opérandes, complémentés à deux, sur 18 bits chacun, suivi de multiplexeurs et d'un additionneur/soustracteur à trois entrées sur 48 bits avec extension de signe (figure 1). Ce module configurable matérialise des opérations arithmétiques très répandues dans les algorithmes DSP. Il est également doté de toute une panoplie de caractéristiques améliorant son efficacité, sa souplesse d'utilisation ou ses performances en vitesse. Ainsi, les opérandes d'entrée sont « pipelinables » par programme, des produits intermédiaires sont aisément accessibles ou encore un bus interne sur 48 bits permet d'agréger un nombre pratiquement illimité de blocs DSP48. Il est possible de cascader directement un résultat d'une cellule DSP48 à l'autre. Une caractéristique bien utile pour la réalisation d'un filtre.

Architecture d'une tuile DSP48 dans les FPGA Virtex-4 de Xilinx FIGURE 1

Une tuile est composée de deux blocs (ou slices) DSP48 identiques disposés verticalement. Sur la matrice, les tuiles DSP48 sont superposées en colonne.



A l'entrée du multiplieur, si les données sont non signées, la largeur maximale est alors de 17 bits. Il ne faut pas oublier d'étendre les données/coefficients non signés en concaténant un ou plusieurs « 0 » au bit le plus significatif (MSB). De même, pour l'addi-

tionneur/soustracteur, ses entrées/sorties devront être de 48 bits ou moins pour de l'arithmétique signée et de 47 bits ou moins si non signée.
Pour les exemples décrits dans cet article, nous travaillerons avec des données signées.
Autre point important : tous les registres internes de ces blocs DSP48 possèdent un reset synchrone (l'usage d'un reset asynchrone empêcherait le logiciel de synthèse d'utiliser les registres internes DSP48). La fonctionnalité de reset est prioritaire, indépendamment du code opérationnel ou autres entrées de contrôle.
Notons enfin que le dernier étage d'additionneur/soustracteur peut être piloté dynamiquement pour prendre une entrée sur 48 bits issue soit du retour de l'étage de sortie, soit de la retenue, puis l'additionner ou la soustraire à une autre entrée de 48 ou 36 bits qui est, dans la majorité des cas, la sortie du multiplieur.

La fonction multiplieur-accumulateur en modèle de base

Comme premier exemple, nous choisissons la fonction la plus basique du traitement du signal, soit la réalisation d'un multiplieur-accumulateur (MAC). Cette fonction est très présente dans les filtres FIR et autres modules du traitement du signal. Voici le code source en VHDL :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;  -- Important to make sure signed arithmetic
                                   is used

entity MULT_ACC is
    Port ( CK : in std_logic;
           RST : in std_logic;                          -- Synchronous Reset
           Ain : in std_logic_vector(17 downto 0);      -- A input of the multiplier
           Bin : in std_logic_vector(17 downto 0);      -- B input of the multiplier
           S : out std_logic_vector(47 downto 0)        );  -- Accumulator output
end MULT_ACC;
architecture Behavioral of MULT_ACC is
signal MULT : std_logic_vector(35 downto 0);            -- Multiplier output
signal ACC : std_logic_vector(47 downto 0);             -- Accumulator output
begin
MULT <= AIN * BIN;
process(CK)  begin
          if CK'event and CK = '1'  then
                  if RST = '1'  then   ACC <= (others => '0');
                  else                 ACC <= ACC + MULT;
                  end if;
          end if;
end process;
S <= ACC;
end Behavioral;
```

Cet exemple, comme tous ceux qui vont suivre, a été synthétisé avec les logiciels XST 8.1 de Xilinx et Synplify Pro 8.1 de Synplify, puis implanté sur une matrice Virtex-4 avec l'outil ISE 7.1 SP3. Cette fonction MAC est réalisée sur un seul bloc DSP48 et aucune autre ressource logique n'est nécessaire. La performance obtenue est de 180 à 200 MHz, suivant le placement et le routage.
Si votre application exige plus de performances et moins de dépendance par rapport aux outils de placement/routage, il est possible d'améliorer ce multiplieur-accumulateur basique en le dotant d'entrées et sorties pipelinées. En effet, les blocs DSP48 ont des registres d'entrée internes (à zéro, un ou deux étages pour les entrées A et B), ainsi qu'un registre sélectionnable en sortie du multiplieur. Le code VHDL suivant utilise un étage de registres aux entrées A et B, ainsi que le registre de sortie du multiplieur. On obtient un multiplieur-accumulateur pipeliné. Les premières lignes de code servant aux déclarations étant les mêmes que dans l'exemple précédent, il était inutile de les redonner.

```
architecture Behavioral of MULT_ACC is
signal AinR, BinR : std_logic_vector(17 downto 0);      -- Registered Ain and Bin
signal MULTR : std_logic_vector(35 downto 0);           -- Registered Multiplier output
signal ACC : std_logic_vector(47 downto 0);             -- Accumulator output
begin
process(CK)  begin
          if CK'event and CK = '1'  then
                  if RST = '1'  then  AinR <= (others => '0');
                                      BinR <= (others => '0');
                                      MULTR <= (others => '0');
                                      ACC <= (others => '0');
                  else                AinR <= Ain;
                                      BinR <= Bin;
                                      MULTR <= AinR * BinR;
                                      ACC <= ACC + MULTR;
                  end if;
          end if;
end process;
S <= ACC;
end Behavioral;
```

Ce multiplieur-accumulateur pipeliné sera synthétisé en utilisant juste un bloc DSP. L'appel à ces registres internes permet d'accroître la fréquence de fonctionnement jusqu'à plus de 400 MHz pour la matrice Virtex-4 d'entrée de gamme, indépendamment des outils d'implantation.

Multiplier sur plus de 18 bits avec un seul bloc DSP48

Etudions l'exemple du multiplieur non pas 18 x 18 bits mais 35 x 18 bits implanté sur une seule cellule DSP48. Cette version est particulièrement adéquate pour les multiplications de données sur des bus de plus de 18 bits. Comme le montre la figure 2, cette opération se déroulera de façon séquentielle sur deux coups d'horloge mais au sein d'un seul et même bloc DSP48, configuré avec un code opératoire différent (Opcode) pour chacune des deux phases de la séquence. L'opérande A comprend 35 et non 36 bits car, lors de sa découpe en deux mots de 18 bits, la partie basse aura son bit le plus significatif (le 18e) qui devra être mis à zéro (bit de signe positif). Lors du premier coup d'horloge, la partie basse de l'opérande A est multipliée par l'opérande B. Ce premier résultat sur 36 bits est chargé dans le registre de sortie. Les 17 bits de plus faible poids de ce produit partiel correspondent à la partie basse du produit final. Le second coup d'horloge voit le décalage à droite sur 17 bits du résultat précédent, puis l'addition du nombre décalé au deuxième produit entre A[34:17] et B[17:0].
Dans le code VHDL suivant, précisons que cette addition de Pout à ZERO équivaut à une opération de chargement dans P_{OUT} (note 1). La note 2, adressée à la ligne :

```
Pout <= Pout +MULT
```

indique que cette instruction peut être remplacée par le décalage à droite sur 17 bits décrit ci-dessus et s'écrira alors :

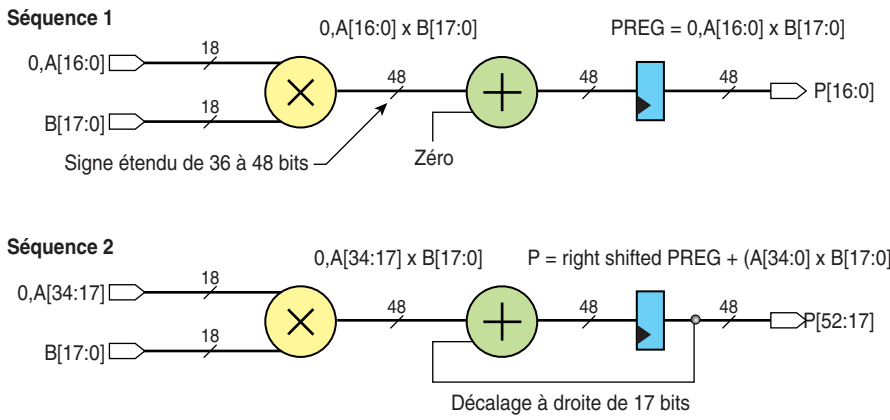
```
else Pout <= ZERO + Pout(47 downto 17) + MULT;
```

Mais attention, dans ce cas, cette caractéristique est supportée seulement par Synplify Pro 8.2.
Le multiplieur sur 35 x 18 bits se décrit comme suit en VHDL :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
entity MULT_ACC_ADD is
    Port ( CK : in std_logic;
           RST : in std_logic;
           SEL : in std_logic;
           A_in : in std_logic_vector(17 downto 0);
           B_in : in std_logic_vector(17 downto 0);
           C_in : in std_logic_vector(47 downto 0);
           S : out std_logic_vector(47 downto 0));
end MULT_ACC_ADD;
```

Implantation d'un multiplieur 35x18 bits sur un seul bloc DSP48 — FIGURE 2

Ce multiplieur 35 x 18 bits utilise deux coups d'horloge. La première séquence effectue la multiplication de la donnée B par les 17 bits de poids faibles de A. La seconde détermine les bits de poids forts du résultat final.



```
architecture Behavioral of MULT_ACC_ADD is
constant ZERO : std_logic_vector(47 downto 0) := (others => '0');
signal AR, BR : std_logic_vector(17 downto 0);
signal MULT : std_logic_vector(35 downto 0);
signal Pout : std_logic_vector(47 downto 0);
begin
process(CK)  begin
  if CK'event and CK = '1' then
    if RST = '1'  then  AR <= (others => '0');
                        BR <= (others => '0');
                        MULT <= (others => '0');
                        Pout <= (others => '0');
    else
          AR <= A_in;
          BR <= B_in;
          MULT <= AR * BR;

          if SEL = '0'  then  Pout <= C_in + MULT;    -- Opcode = 0x35 for C input
--                                                                0x15 for PCIN input
--          if SEL = '0'  then  Pout <= ZERO + MULT;  -- Opcode = 0x05 for ZERO
--                                                            constant as input (Note 1)
          else        Pout <= Pout + MULT;            -- Opcode = 0x25 (Note 2)
          end if;
  end if;
end process;
S <= Pout;
end Behavioral;
```

En fait, les résultats de la synthèse ne sont pas aussi optimisés qu'on pourrait l'espérer avec la version 8.1 du logiciel de synthèse XST. Ainsi, de la logique combinatoire sera utilisée pour implanter le multiplexeur entre C_in et Pout, alors que cette même fonction est disponible à l'intérieur du bloc DSP48. Avec Synplify Pro 8.2 l'implantation est idéale pour le même code VHDL. La fréquence de fonctionnement obtenue (dans les deux cas de synthèse) varie de 220 à 270 MHz suivant la caractéristique « speed grade » du FPGA.
Si pour quelque raison vous ne voulez pas utiliser le registre de sortie du multiplieur, vous pouvez écrire :

```
Pout <= Pout + (AR*BR);
```

Cela évite de déclarer une sortie de multiplieur combinatoire. En plus, le code VHDL est ainsi plus compact.

Optimiser l'arrondi symétrique

Une autre fonction très utile dans les algorithmes de traitement du signal est l'arrondi symétrique. Celle-ci remplace les nombres fractionnaires par l'entier le plus proche. Par exemple, 2,8 est arrondi à 3 et 2,2 à 2. Les nombres négatifs comme -2,8 et -2,2 sont respectivement mis à -3 et -2. Le tableau I de la page 47, montre le déroulement du processus. La valeur dans le registre C indique sur quels bits doit porter l'arrondi. Le code VHDL suivant correspond à l'arrondi symétrique sur 20 bits du résultat de la multiplication Ain x Bin :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
entity ROUNDING is
    Port ( CK : in std_logic;
           RST : in std_logic;
           Ain : in std_logic_vector(17 downto 0);
           Bin : in std_logic_vector(17 downto 0);
           P : out std_logic_vector(19 downto 0));
end ROUNDING;
architecture Behavioral of ROUNDING is
constant ZERO : std_logic_vector(47 downto 0) := (others => '0');
signal AR, BR : std_logic_vector(17 downto 0);
signal MULTR : std_logic_vector(35 downto 0);
signal Pout : std_logic_vector(47 downto 0);
signal Carry_in, Carry_inR : std_logic;
begin
process(CK)  begin
  if CK'event and CK = '1'  then
        Carry_in <= not(Ain(17) xor Bin(17));
        Carry_inR <= Carry_in;
        if RST = '1'  then                AR <= (others => '0');
                                                BR <= (others => '0');
                                                MULTR <= (others => '0');
                                                Pout <= (others => '0');
        else
                AR <= Ain;
                BR <= Bin;
                MULTR <= AR * BR;
  -- Note that the following 4 operands adder will be implemented as a 3 operand one :
  -- ZERO is a constant that allows easy sign extension for the VHDL syntax
                Pout <= ZERO + MULTR + x"7FFF" + Carry_inR;
        end if;
  end if;
end process;
P <= Pout(35 downto 16);
end Behavioral;
```

La synthèse de cette fonction amène à une implantation sur un seul bloc DSP48 avec en plus un module LUT et deux flips-flops associées (pour l'entrée de la retenue notamment). La fréquence de fonctionnement va de 400 à 500 MHz suivant le FPGA.

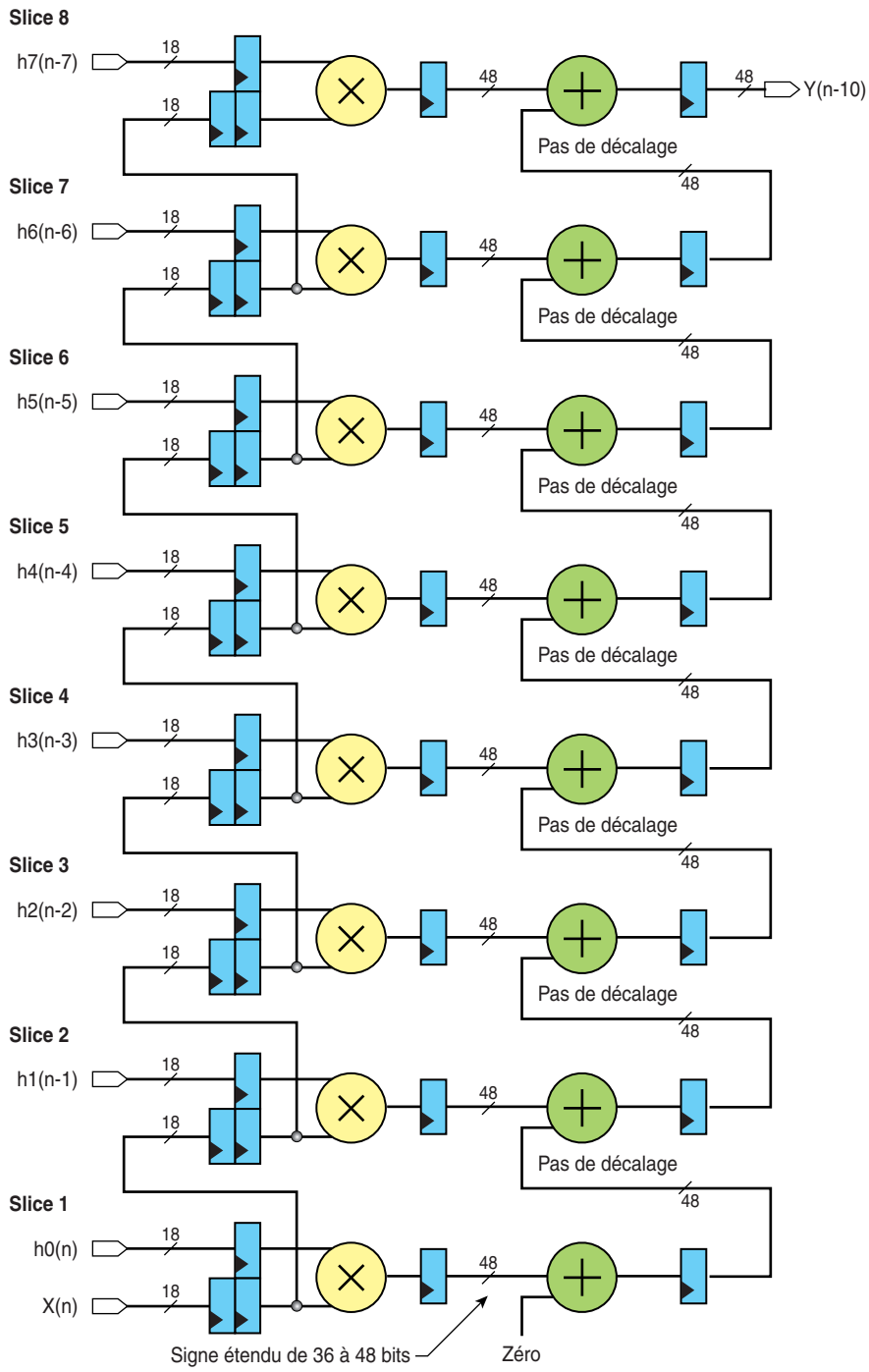
Un filtre FIR à 500 MHz

La simplicité et l'efficacité d'une description en langage VHDL ont été démontrées sur des exemples basiques, mais la méthode est aussi probante pour des fonctions beaucoup plus complexes. En effet, les algorithmes de traitement du signal sont souvent une succession d'opérations relativement simples appliquées à un grand nombre de données. De ce fait, la structure des matrices de logique programmable Virtex-4 convient particulièrement bien à ces traitements de type parallèle. Prenons l'exemple d'un filtre FIR d'architecture systolique (figure 3). Le traitement de chaque coefficient peut être implanté dans un seul bloc DSP48. La chaîne d'additions profite des connexions directes entre entrées et sorties des additionneurs voisins (P_IN et P dans le code ci-dessous ou PC_{IN} et PC_{OUT}

Implantation d'un filtre FIR systolique

FIGURE 3

Cet exemple de filtre FIR à huit coefficients en version systolique utilise huit slices DSP48. Les additionneurs de ces slices sont mis en cascade, ce qui améliore les performances de traitement et diminue la consommation.



sur la figure 1). De plus, la mise en cascade des opérandes B est également utilisée (via B et BS dans le code ci-dessous ou BC_{IN} et BC_{OUT} sur la figure 1) pour propager les données d'un bloc DSP48 à l'autre.
Pour l'écriture du programme VHDL, un code « générique », implantable dans un seul bloc DSP48, définit le traitement d'un coefficient du filtre. Ce module est ensuite implanté autant de fois qu'il y a de coefficients.
Pour faciliter les modifications, des paramètres génériques sont utilisés qui sont déclarés dans le code suivant :

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
package CONSTANTS is
        constant K_Ndat : integer := 18;                        -- Number of bits for DATA
        constant K_Ncoef : integer := 18;               -- Number of bits for COEFFICIENTS
        constant K_Nacc : integer := 48;                -- Number of bits for FIR output
        constant Number_of_coeffs : integer := 15;              -- Number of TAPs
        type COEFFS_TYPE is array(0 to Number_of_coeffs-1) of integer;
        constant COEFFS : COEFFS_TYPE := (
        88899,                          -- 1
        -12167,                         -- 2
        114259,  -- 3
        -67391,                         -- 4
        91300,   -- 5
        -115433,-- 6
        95591,   -- 7
        99375,                          -- 8
        -27666,                         -- 9
        9599,                           -- 10
        77421,                          -- 11
        -33333,  -- 12
        27654,                          -- 13
        44321,                          -- 14
        -921    );                      -- 15
end CONSTANTS;
```

Le code pour un coefficient est le suivant (multiplieur pipeliné et additionneur avec deux niveaux de registres sur le chemin de données) :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
entity MADD_GEN is
        generic(Ndat : integer := 18;
                        Ncoef : integer := 18;
                        Nacc : integer := 48  );
  Port ( CK : in std_logic;
    A : in std_logic_vector(Ncoef-1 downto 0);      -- Coefficient input
    B : in std_logic_vector(Ndat-1 downto 0);                       -- Data input
    P_IN : in std_logic_vector(Nacc-1 downto 0);  -- Adder chain input from the previous TAP
    RST : in std_logic;
    BS : out std_logic_vector(Ndat-1 downto 0);    -- Pipelined Data
    P : out std_logic_vector(Nacc-1 downto 0));    -- Adder chain output
end MADD_GEN;
architecture Behavioral of MADD_GEN is
signal AR : std_logic_vector(Ncoef-1 downto 0);
signal BR, BS_INT : std_logic_vector(Ndat-1 downto 0);
signal MLT : std_logic_vector(Ndat + Ncoef-1 downto 0);
begin
process(CK)   begin
  if CK'event and CK = '1' then
                if RST = '1' then
                                MLT <= (others => '0');
                                P <= (others => '0');
                                AR <= (others => '0');
                                BR <= (others => '0');
                                BS_INT <= (others => '0');
                else
                        MLT <= AR * BS_INT;
                        P <= P_IN + MLT;
                        AR <= A;
                        BR <= B;
                        BS_INT <= BR;
                end if;
  end if;
end process;
BS <= BS_INT;
end Behavioral;
```

Ensuite le nombre de coefficients requis est sélectionné (dans Number_of_coeffs déclaré dans le module VHDL CONSTANT), soit ici 15 pour implanter notre filtre.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
library WORK;
use WORK.CONSTANTS.all;
entity TOP_GEN is
        generic(  Ncoef : integer := K_Ncoef;          -- Coef bus width
                        Ndat : integer := K_Ndat;          -- Data bus width
                        Nacc : integer := K_Nacc  );     -- Accumulator bus width
   Port ( CK : in std_logic;
          RST : in std_logic;
          DIN : in std_logic_vector(Ndat-1 downto 0);
          PS : out std_logic_vector(Nacc-1 downto 0));
end TOP_GEN;
architecture Behavioral of TOP_GEN is
component MADD_GEN
        generic(Ndat : integer := 18;
                  Ncoef : integer := 18;
                  Nacc : integer := 48  );
   Port ( CK : in std_logic;
          A : in std_logic_vector(Ncoef-1 downto 0);
          B : in std_logic_vector(Ndat-1 downto 0);
          P_IN : in std_logic_vector(Nacc-1 downto 0);
          RST : in std_logic;
          BS : out std_logic_vector(Ndat-1 downto 0);
          P : out std_logic_vector(Nacc-1 downto 0)  );
end component;
type COEFF_TYPE is array(Number_of_coeffs downto 0) of std_logic_vector
(Ncoef-1 downto 0);
signal COEFF : COEFF_TYPE;
type B_TYPE is array(Number_of_coeffs downto 0) of std_logic_vector(Ndat-1 downto 0);
signal B : B_type;
type P_TYPE is array(Number_of_coeffs downto 0) of std_logic_vector(Nacc-1 downto 0);
signal P : P_type;
signal DINR, DINS : std_logic_vector(Ndat-1 downto 0);
signal PR : std_logic_vector(Nacc - 1 downto 0);
begin
B(0) <= DINS;
P(0) <= (others => '0');
-- Converting the coefficients from integer to std_logic_vector
GEN_CO : for I in 0 to Number_of_coeffs-1 generate
                COEFF(I) <= conv_std_logic_vector(COEFFS(I), Ncoef);
end generate;
```

```
FILTER_GEN : for I in 0 to Number_of_coeffs-1 generate
        F : MADD_GEN
                generic map (
                        Ndat  => K_Ndat,
                        Ncoef => K_Ncoef,
                        Nacc  => K_Nacc)
                port map (
                        CK => CK,
                        RST => RST,
                        A => COEFF(I),
                        B => B(I),
                        P_IN => P(I),
                        BS              => B(I+1),
                        P  => P(I+1)  );
end generate;
-- Input and output registers
-- Two levels of inputs and output registers have been used in this example, just like
-- mentioned in the Complex_multiplier example
process(CK, RST)  begin
  if RST = '1'  then            DINR <= (others => '0');
                                DINS <= (others => '0');
                                PR <= (others => '0');
                                PS <= (others => '0');
  elsif CK'event and CK = '1'  then
                DINR <= DIN;
                DINS <= DINR;
                PR <= P(Number_of_coeffs);
                PS <= PR;
  end if;
end process;
end Behavioral;
```

Après synthèse, l'implantation de ce filtre à 15 coefficients requiert 15 blocs DSP48 sur une colonne et aucun routage à usage général (sauf pour les coefficients et les données d'entrée/sortie). La fréquence de fonctionnement est de 500 MHz pour les matrices Virtex-4 de « speedgrade » -12 (400 MHz pour « speedgrade » -10). ■