Politecnico di Torino
Dipartimento di Elettronica e Telecomunicazioni

Laboratory #1
Integrated Systems Architecture
Design and Implementation of a Digital Filter

Master Degree in Electronic Engineering

Authors: Group 1

Alberto Aimaro 253196
Beatrice Bussolino 251190
Alessio Colucci 251197
Fabio Zanoni 232113

January 31, 2019

Contents

1 Setting up

First of all, filter order N and the bit-witdh n_b corresponding to group number and group participants must be computed, using the following equations:

$$N = 2^p \left[(x \bmod 2) + 1\right] + 6p \tag{1}$$

$$n_b = (y \bmod 7) + 8 \tag{2}$$

The parameters are x = 6 (from Aimaro) and y = 9 (from Bussolino), with p = 1 (since the group is the first one so it is in odd position). The final results are N = 8 and n_b = 10, with the filter to be designed being a Finite-Impulse-Response filter. Resulting coefficients are:

$$[-4 \quad -7 \quad 26 \quad 136 \quad 207 \quad 136 \quad 26 \quad -7 \quad -4]$$

Firstly, the filter is modeled using provided MATLAB and C codes, as described in the next section.

2 MATLAB/C Models for the Filter

Given codes are used as a basis to develop specific models, which are presented in the appendices.

C code is adapted to handle overflow, that doesn't happen when working with 32-bits integers but must be taken into account to correctly dimension adders of the filter.

Number are represented in Q1.9 format; when performing a multiplication, the result is in Q1.18 format and must be right-shifted of 9 positions ($n_b - 1$) to restore original representation.

Performing nine sums using 10-bits numbers that can assume any value in possible range, it is necessary to add four bits to avoid overflow. However, using C model with worst case inputs (all samples equal to 511/-512), it is find out that a 1-bit extension is sufficient to handle overflow. In C model, 32-bits integer are used, therefore no sign extension needs to be specified; however, final result must be right shifted of one position and truncated to be representable on 10 bits.

```
1  /// shift and insert new sample in x shift register
2  for (i=NT-1; i>0; i--)
3  sx[i] = sx[i-1];
4  sx[0] = x;
5
6  /// make the convolution
7  /// Moving average part
8  y = 0;
9  for (i=0; i<NT; i++)
10 y += (sx[i]*b[i]) >> (NB-1) ;
11
12 /// update the y shift register
13 for (i=NT-2; i>0; i--)
14 sy[i] = sy[i-1];
15 sy[0] = y;
16
17 return y>>1;
```

Reported code represents the behavior of the filter. At line 10 results of multiplication are shifted of n_b-1=9 positions. At line 17, returned result is shifted of one position, to handle overflow.

3 VHDL Implementation of the FIR Filter

3.1 Design

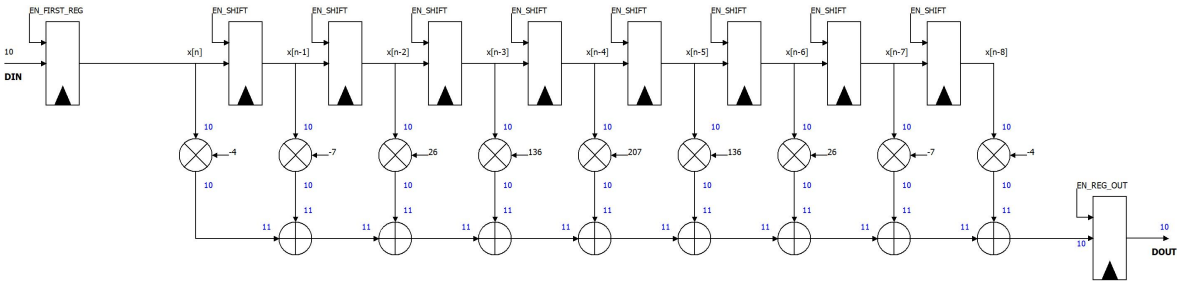

Figure 1: Base architecture for FIR filter of order 8 with 10 bits signals

After checking all the results given by C model, the hardware architecture for the filter is implemented, following a standard FIR architecture. A Control Unit is included in the project to control the flow of the filter, mostly forecasting its use in the optimized architecture.

In VHDL description of filter:

- A bank of registers is instantiated; these registers have a synchronous reset, to avoid glitch-related problems.
- Signals are instantiated of SIGNED type, to easily handle sums and multiplications. In *numeric_std* library, the output of a multiplication between two numbers of length N is represented with 2N bits. However, since numbers are in Q1.9 format, only 2N-1 least significants bit can be used to correctly represent the result. Therefore, the most significant bit can be eliminated and then the result can be right shifted to return to Q1.9 representation (N-1 least significant bits truncated);
- As explained in previous section, one single guard bit is necessary to avoid overflows. Adding this bit is implemented practically by not removing the most significant bit after multiplication.
- Output samples, before output register, must then be right shifted of one position (least significant bit truncated) to return the result with 10 bits. In figure, the numbers of bits used are represented in blue.
- Input and output registers are introduced to synchronize the filter with external world; they introduce a delay of two clock cycles.

In Figure 2 is shown the connection between the filter and its control unit.

In Figure 3 is reported the ASM chart of the control unit. In IDLE state, input register is update with new values at each clock cycle, while registers of the shift array keep old values. If input data is valid (VIN=1), then output register and registers of the array are updated (DATA_CYCLE1). In the next clock cycle, data is valid in output and VOUT si asserted. After DATA_CYCLE1, the control unit moves to LAST_DATA1 if there are no other data to process and then returns in idle or to DATA_CYCLE2 if there are new values.

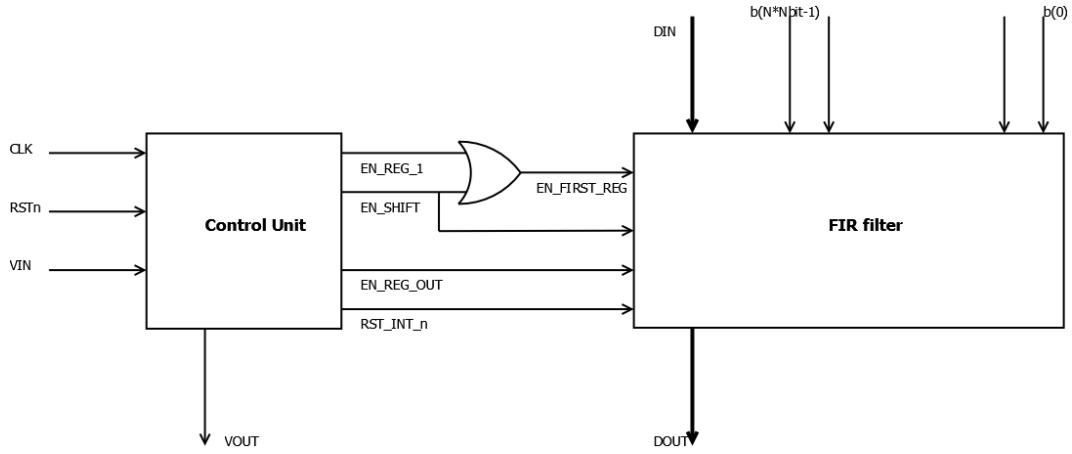

Figure 2: Control Unit and FIR Filter

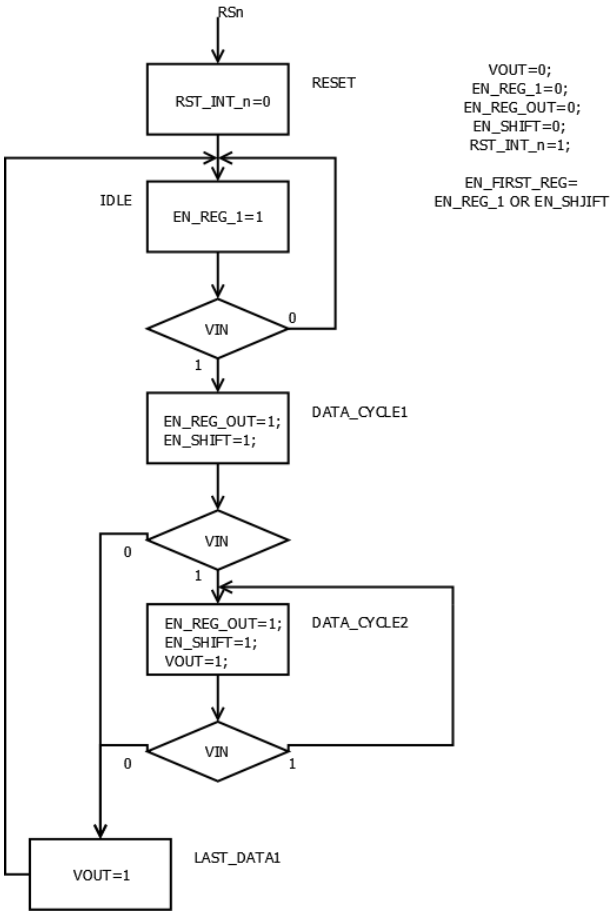

Figure 3: ASM chart of FIR filter control unit

3.2 Physical Implementation

3.2.1 Synthesis

After checking the correctness of the simulation, the architecture is synthesized with the 70nm cells present in the Nangate Open Cell Library. Initially, target period is set to 0ns to find the maximum

achievable frequency. Then many other synthesis are repeated increasing target period until finding a slack time equal to zero (Figure 4). In Table 1 maximum frequency is reported with corresponding cell area.

```
---------------------------------------------------------------------------
data required time                                                    2.85
data arrival time                                                    -2.85
---------------------------------------------------------------------------
slack (MET)                                                           0.00
```

Figure 4: Slack time equal to 0 met

		Area
T_{min}	2.96ns	$5157.739746\mu m^2$
f_{max}	337.84MHz	
$4 \cdot T_{min}$	11.84ns	$4889.080078\mu m^2$
$f_{max}/4$	84.46MHz	

Table 1: Maximum frequency achievable and corresponding cell area; maximum frequency divided by four and corresponding cell area

Frequency is then set at $f_{max}/4$; again, obtained area of the cell is reported in Table 1. As expected, the area of the cell with target frequency f_{max} is larger, as the synthesizer needs to do more optimizations to respect the imposed limit.

After synthesis, it is possible to obtain an estimation of power consumption of the cell.

```
  Cell Internal Power  = 278.0258 uW   (55%)
  Net Switching Power  = 229.0841 uW   (45%)
                         ---------
Total Dynamic Power    = 507.1099 uW  (100%)

Cell Leakage Power     = 113.0839 uW
```

Figure 5: Obtained power report at $f = f_{max}/4$

3.2.2 Place and Route

The filter is placed and routed with Cadence SOC Innovus with $f = f_{max}/4$ and the resulting area is $4879.5\mu m^2$ (Figure 6).

The behavior of the filter is then tested including the delay file (*.sdf* file) generated by Cadence Innovus: power report obtained is shown in Figure 7 (power units: mW).

3.3 Simulation

To the testbench model provided in course material, a section is added to test also the start-stop behavior depending on the input signal VIN. The signal generator is modified too, to better fit the specific design, but the final result is the same.

The differences between provided signal generator and the modified version consist in how data are processed. In the modified signal generator there is a process which runs through all the data to

```
Gate area 0.7980 um^2

Level 0 Module FIR_filter

Gates=       6114
Cells=       3592
Area=         4879.5 um^2
```

Figure 6: Obtained area after place & route

```
Total Power
-------------------------------------------------------------------------------------------
Total Internal Power:        0.68929791        48.7703%
Total Switching Power:       0.61138734        43.2579%
Total Leakage Power:         0.11267000         7.9718%
Total Power:                 1.41335525
-------------------------------------------------------------------------------------------
```

Figure 7: Obtained power report at $f = f_{max}/4$, after place and route

be sent as input to the Unit Under Test and saves them, while the other one sends the data following the standard pattern and ending the simulation once it is done. In data_maker_new instead there is a process reading and sending all the data to the UUT at run-time, while the other process is used to correctly end the simulation.

The only encountered problem was related to matching the Verilog connections in the top test-bench, because Verilog does not allow custom types; thus, all the custom connections for coefficients (at the beginning written as an array of std_logic_vector) and input/output had to be converted to single long std_logic_vector and multiplexed inside the components.

Three simulations are performed:

1. Behavioral simulation
2. Post synthesis simulation
3. Post place and route simulation

All produced results are compared with those generated by C model. The correctness of simulations can be checked from Figure 8.

In Figure 9 it is possible to see that the latency is of one clock cycle, while the throughput is 1 data/cycle.

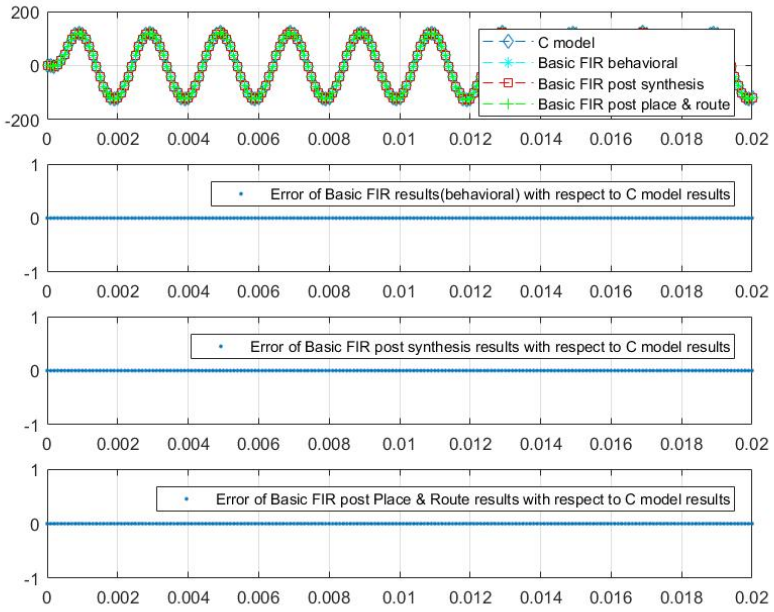

Figure 8: Top: The results from C model and from the three simulations are plotted together, to show their equality; Bottom: the three bottom plots show the error between simulation results and C model results

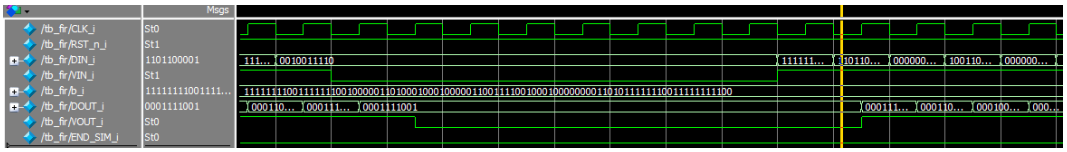

Figure 9: Some simulation signals from behavioral simulation, showing the start-stop behavior and the latency.

4 Optimization of designed Filter

4.1 Design

FIR filter is optimized with unfolding and pipelining, to improve both throughput and maximum frequency at the expense of higher area, caused by replication of the architecture and insertion of registers to split critical paths.

The system is unfolded with order 3; firstly, the architecture is replicated, then these relations are applied:

$$j = \mathrm{mod}\left(\frac{i+w}{P}\right) \tag{3}$$

$$w_i = \left\lfloor \frac{i+w}{P} \right\rfloor \tag{4}$$

where i is the starting copy of the graph (in this case 0, 1 or 2), j is the arrival copy, w is the number of registers present on the original arc and P is the unfolding order, in particular 3. Following these expressions, for each arc connecting two nodes (which are the operators plus inputs and outputs)a new arc for each copy (i from 0 to 2) is generated. The final structure can be seen in Figure 10.

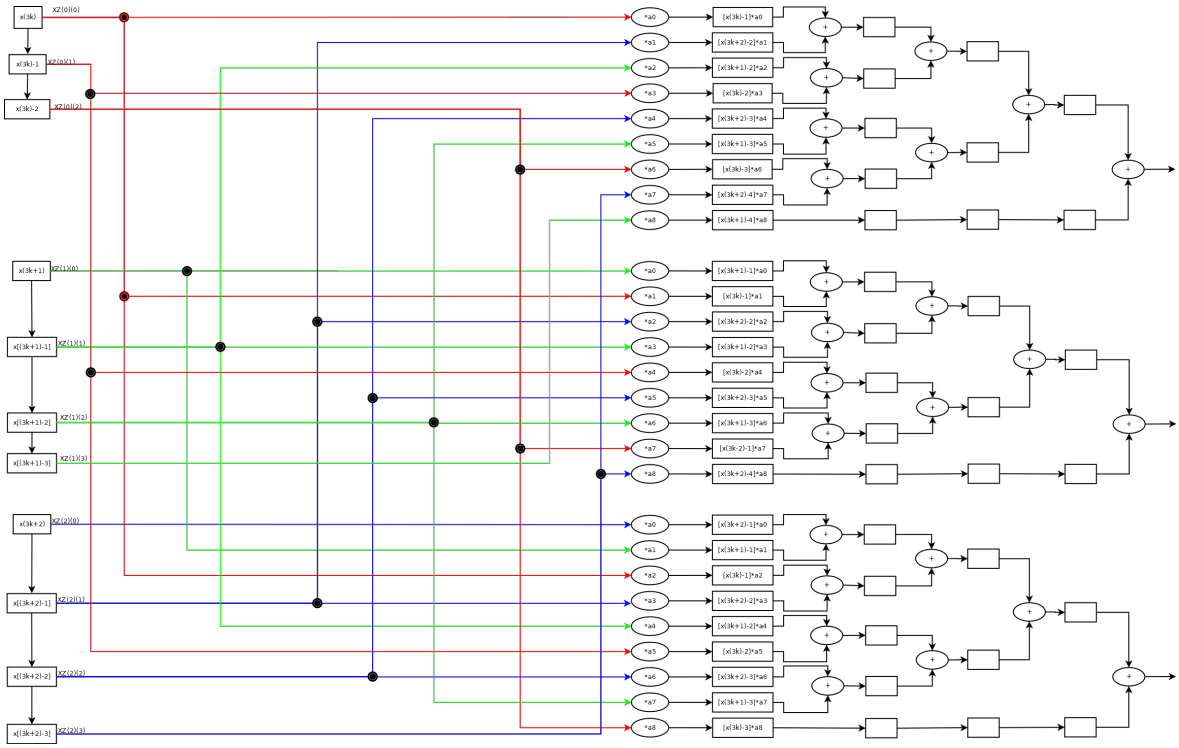

Figure 10: Final scheme of FIR filter after unfolding

After properly connecting the whole architecture, pipeline registers can be added. Adding registers to the standard FIR graph is not very useful: a possibility would be splitting all the cascaded sums in different clock cycles but the result would be only an increased area consumption, without improving performances, that are mostly limited by the multipliers. In fact, it is found out that the delay of 1 multiplier is roughly equal to that of 7/8 adders.

Thus a slightly different architecture is tested, a tree structure with all the multiplications in parallel in the first cycle, then four additions in parallel, later only two and then two final stages

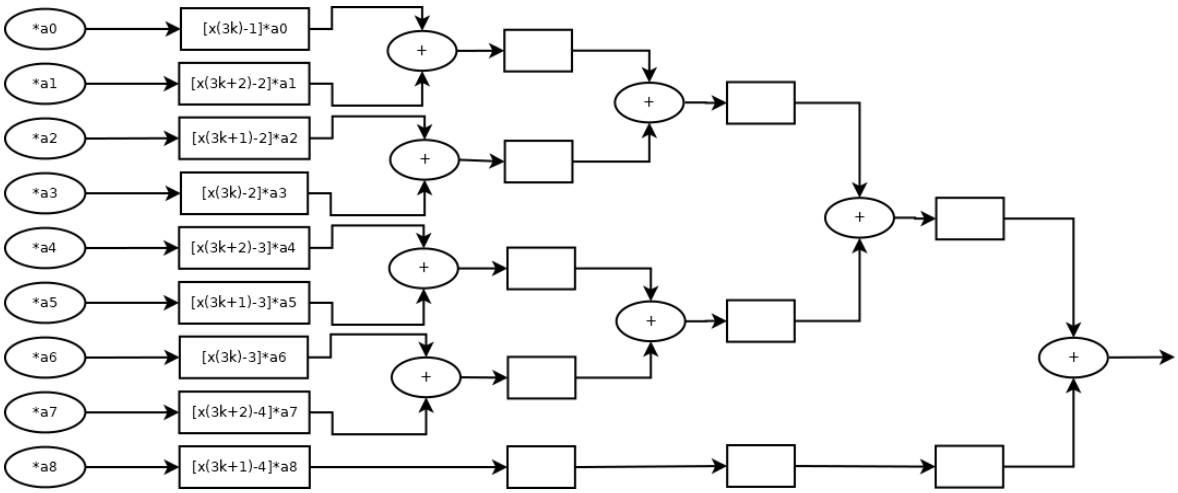

Figure 11: Final scheme of FIR filter after unfolding - zoom on computation elements

with one sum each, the last one needed to add the ninth term. The stages are in-between each two operands, to maximize performances at cost of more latency and slightly more area. However, the improvement should be bounded by the presence of multipliers also in this case.

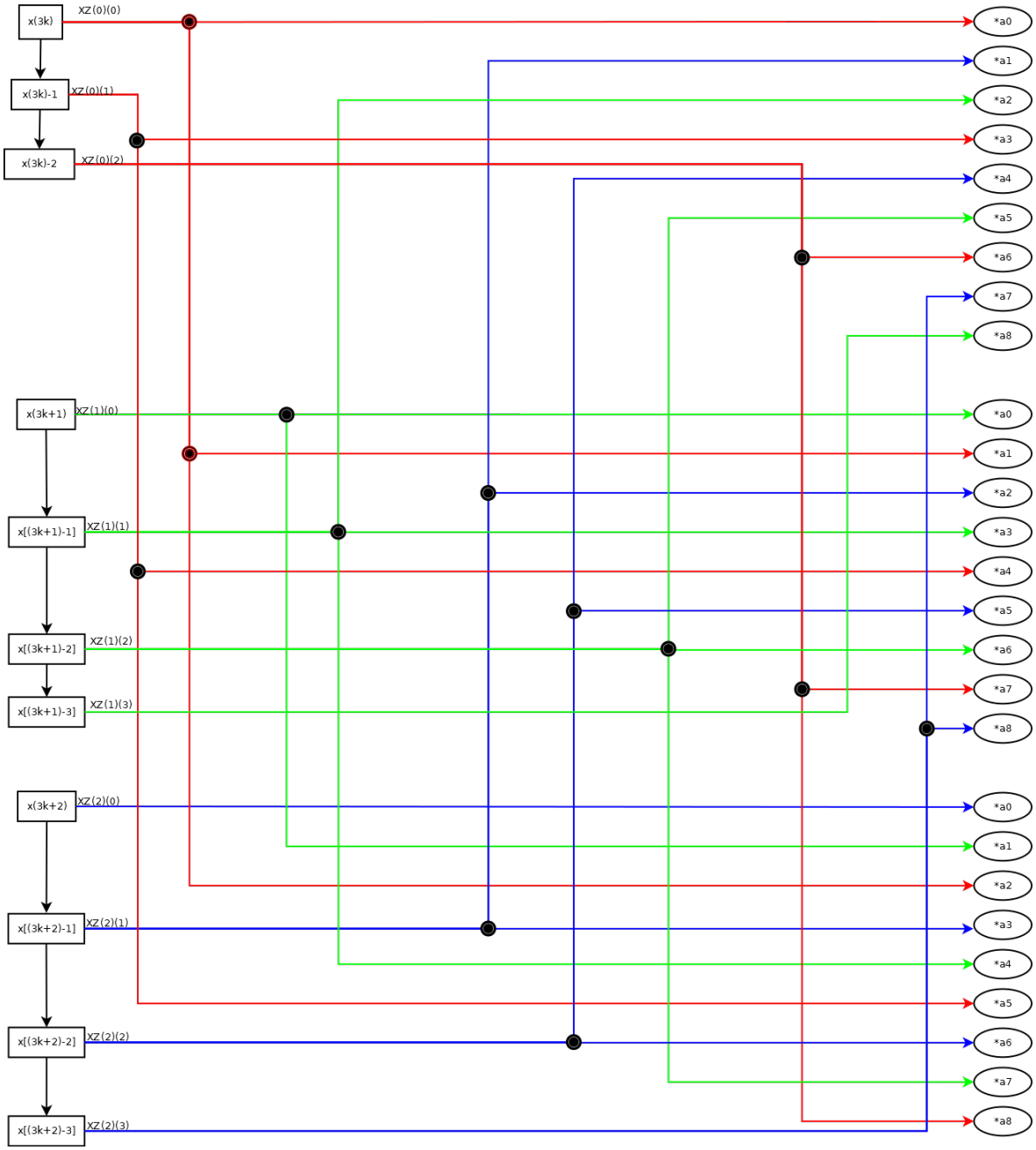

Figure 12: Final scheme of FIR filter after unfolding - zoom on interconnections

4.2 Physical Implementation

4.2.1 Synthesis

After checking the correctness of the behavioral simulation, the architecture is synthesized with the same technology used for unoptimized design (70nm cells present in the Nangate Open Cell Library). Again, target period is set to 0ns to find the maximum achievable frequency and then increased until finding a slack time equal to zero. In Table 1 maximum frequency is reported with corresponding cell area.

		Area
T_{min} f_{max}	2ns 500MHz	$18444.173828\mu m^2$
$4 \cdot T_{min}$ $f_{max}/4$	8ns 125MHz	$18244.408203\mu m^2$

Table 2: Maximum frequency achievable and corresponding cell area; maximum frequency divided by four and corresponding cell area

Frequency is then set at $f_{max}/4$; again, obtained area of the cell is reported in Table 2. As for unoptimized design, a lower target frequency implies a smaller area.

The after synthesis power estimation is here reported:

```
  Cell Internal Power  =   1.7219 mW   (58%)
  Net Switching Power  =   1.2622 mW   (42%)
                         ---------
Total Dynamic Power    =   2.9841 mW  (100%)

Cell Leakage Power     = 421.1755 uW
```

Figure 13: Obtained power report at $f = f_{max}/4$

4.2.2 Place And Route

The filter is placed and routed with Cadence SOC Innovus with $f = f_{max}/4$ and the resulting area is $18043.0\mu m^2$ (Figure 14).

```
Gate area 0.7980 um^2

Level 0 Module FIR_filter

Gates=     22610
Cells=   12591
Area=   18043.0 um^2
```

Figure 14: Obtained power report at $f = f_{max}/4$

The behavior of the filter is then tested including the delay file (*.sdf* file) generated by Cadence Innovus: power report obtained is shown in Figure 15 (power units: mW).

```
Total Power
--------------------------------------------------------------------
Total Internal Power:        3.39771481        51.4597%
Total Switching Power:       2.79317273        42.3036%
Total Leakage Power:         0.41178826         6.2367%
Total Power:                 6.60267580
--------------------------------------------------------------------
```

Figure 15: Obtained power report at $f = f_{max}/4$, after place and route

4.3 Simulation

The simulation for the new architecture is very similar to the previous one, but three inputs and three outputs at a time are used, in the correct order.

Three simulations are performed: behavioral, after synthesis and after place and route. As for the basic filter, the results are compared with those produced by the C model and the equality is checked (Figure 16).

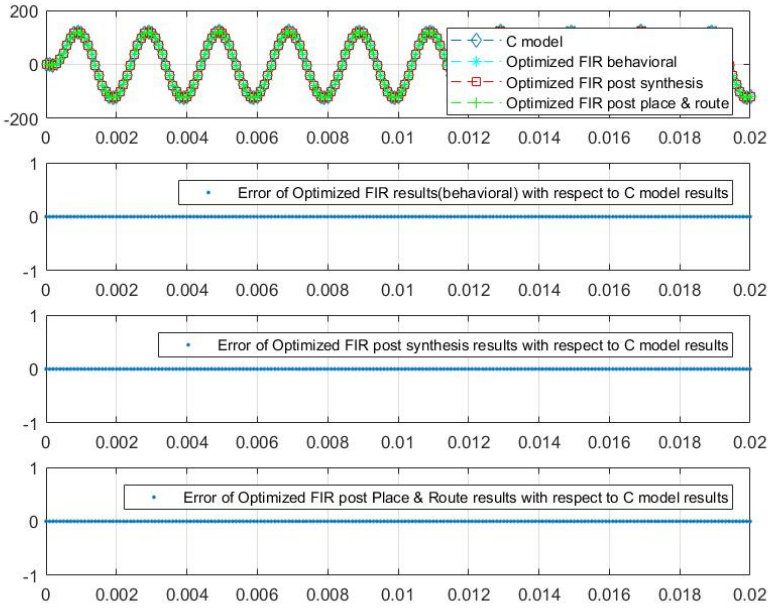

Figure 16: Top: The results from C model and from the three simulations are plotted together, to show their equality; Bottom: the three bottom plots show the error between simulation results and C model results

5 Comparison of optimized and un-optimized designs

In Table 3 is reported a comparison between the Basic FIR and its successive optimization. The unfolding of level three leads to a throughput three times higher and the pipelining allows to increase the maximum clock frequency. However, the critical path of the design is in the multipliers.

To unfold the system, the original architecture is triplicated and this explains the increase of area and power. The increase of dynamic power is associated also to the higher frequency.

	Basic FIR	**Optimized FIR**
T_{min}	2.96 ns	2 ns
f_{max}	337.84MHz	500MHz
$4\,T_{min}$	11.84 ns	8 ns
$f_{max}/4$	84.46MHz	125MHz
Throughput	1	3
Synthesis		
Area	4889.080078 μm^2	18244.408203 μm^2
Internal Power	278.0258 μW	1.7219 mW
Switching Power	229.0841 μW	1.2622 mW
Total Dynamic Power	507.1099 μW	2.9841 mW
Leakage Power	113.0839 μW	421.1755 μW
Total Power	620.1938 μW	3.4053 mW
Place and Route		
Area	4879.5 μm^2	18043.0 μm^2
Internal Power	689.29791 μW	3.39771 mW
Switching Power	611.38734 μW	2.79317 mW
Leakage Power	112.67000 μW	0.41179 mW
Total Power	1.41336 mW	6.60268 mW

Table 3: Comparison between Basic FIR and Optimized FIR

Appendices

Models

These are the models of the filter in MATLAB and C languages.

MATLAB code

./code/models/my_fir_filter.m

```
fs=10000 %% sampling frequency
f1=500;  %% first sinewave freq (in band)
f2=4500; %% second sinnewave freq (out band)

N=8; %% filter order
nb=10; %% number of bits

T=1/500; %% maximum period
tt=0:1/fs:10*T; %% time samples

x1=sin(2*pi*f1*tt); %% first sinewave
x2=sin(2*pi*f2*tt); %% second sinewave

x=(x1+x2)/2; %% input signal

[bi, bq]=myfir_design(N, nb); %% filter design

y=filter(bq, 1, x); %% apply filter

%% plots
figure
plot(tt,x1,'--d');
hold on
plot(tt,x2,'r--s');
plot(tt,x, 'g--+');
plot(tt, y, 'c--o');

legend('x1', 'x2', 'x', 'y')

%% quantize input and output
xq=floor(x*2^(nb));
idx=find(xq==2^(nb));
xq(idx)=2^(nb)-1;

yq=floor(y*2^(nb));
idy=find(yq==2^(nb));
yq(idy)=2^(nb)-1;

%% save input and output
fp=fopen('samples.txt','w');
fprintf(fp,'%d\n', xq);
fclose(fp);

fp=fopen('resultsm.txt', 'w');
fprintf(fp, '%d\n', yq);
fclose(fp);
```

C code

./code/models/my_fir.c

```c
#include<stdio.h>
#include<stdlib.h>

#define NT 9 /// number of coeffs
#define NB 10 /// number of bits

const int b[NT]={-4, -7, 26, 136, 207, 136, 26, -7, -4}; /// b array
//const int a[NT-1]={-147, 52}; /// a array

/// Perform fixed point filtering assming direct form I
///\param x is the new input sample
///\return the new output sample
int myfilter(int x)
{
  static int sx[NT]; /// x shift register
  static int sy[NT-1]; /// y shift register
  static int first_run = 0; /// for cleaning shift registers
  int i; /// index
  int y; /// output sample

  /// clean the buffers
  if (first_run == 0)
  {
    first_run = 1;
    for (i=0; i<NT; i++)
      sx[i] = 0;
    for (i=0; i<NT-1; i++)
      sy[i] = 0;
  }

  /// shift and insert new sample in x shift register
  for (i=NT-1; i>0; i--)
    sx[i] = sx[i-1];
  sx[0] = x;

  /// make the convolution
  /// Moving average part
  y = 0;
  for (i=0; i<NT; i++)
    y += (sx[i]*b[i]) >> (NB-1) ;
  /// Auto regressive part
  //for (i=0; i<NT-1; i++)
  //  y -= (sy[i]*a[i]) >> (NB-1);

  /// update the y shift register
  for (i=NT-2; i>0; i--)
    sy[i] = sy[i-1];
  sy[0] = y;

  return y;
}

int main (int argc, char **argv)
{
  FILE *fp_in;
  FILE *fp_out;

  int x;
  int y;

  /// check the command line
```

```
62    if (argc != 3)
63    {
64      printf("Use: %s <input_file> <output_file>\n", argv[0]);
65      exit(1);
66    }
67
68    /// open files
69    fp_in = fopen(argv[1], "r");
70    if (fp_in == NULL)
71    {
72      printf("Error: cannot open %s\n");
73      exit(2);
74    }
75    fp_out = fopen(argv[2], "w");
76
77    /// get samples and apply filter
78    fscanf(fp_in, "%d", &x);
79    do{
80      y = myfilter(x);
81      fprintf(fp_out,"%d\n", y);
82      fscanf(fp_in, "%d", &x);
83    } while (!feof(fp_in));
84
85    fclose(fp_in);
86    fclose(fp_out);
87
88    return 0;
89
90  }
```

VHDL code - non optimized FIR filter

This is the code of the unoptimized filter; codes of the testbench and of basic elements (eg. registers) are not included.

Package util

./code/fir_unoptimized/util.vhd

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.numeric_std.all;
4  use ieee.math_real.all;
5
6  PACKAGE util IS
7
8  CONSTANT Nbit: INTEGER := 10;
9  CONSTANT N: INTEGER := 9;
10 --CONSTANT Nbit_result: INTEGER := Nbit+integer(ceil(log2(real(N))));
11 CONSTANT Nbit_result: INTEGER := Nbit;
12
13 CONSTANT FINAL_DELAY : integer := N + 5;
14
15 CONSTANT T: time := 20 ns; -- Clock period
16 CONSTANT start_time: time := 101 ns; -- Start time of simulation
17
18 TYPE LIST_N IS ARRAY (0 to N-1) OF SIGNED(Nbit-1 downto 0);
19 TYPE LIST_mult IS ARRAY (0 to N-1) OF SIGNED(Nbit+Nbit-1 downto 0);
20 TYPE LIST_mult_resize IS ARRAY (0 to N-1) OF SIGNED(Nbit_result downto 0);
21 TYPE LIST_sum IS ARRAY (0 to N-2) OF SIGNED(Nbit_result downto 0);
```

```
22
23 END util;
```

FIR filter code

./code/fir_unoptimized/FIR_filter.vhd

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.numeric_std.all;
4
5  library std;
6  USE work.util.all;
7
8  ENTITY FIR_filter IS
9
10 PORT(    CLK:     in std_logic;
11         RST_n:   in std_logic;
12
13         VIN:     in std_logic;
14         VOUT:    out std_logic;
15
16         DIN:     in signed (Nbit-1 downto 0);
17         DOUT:    out signed (Nbit-1 downto 0);
18
19         b:       in std_logic_vector(N * Nbit - 1 downto 0));
20
21 END ENTITY FIR_filter;
22
23 ARCHITECTURE behavior OF FIR_filter IS
24
25 COMPONENT regn IS
26 GENERIC (N: INTEGER := 16);
27 PORT (D : IN SIGNED (N-1 DOWNTO 0);      -- input
28      Clock, Resetn, EN : IN STD_LOGIC;   -- clock, reset, enable
29      Q : OUT SIGNED (N-1 DOWNTO 0));     -- output
30 END COMPONENT regn;
31
32 component dff IS
33 PORT (D : IN STD_LOGIC;      -- input
34      Clock, Resetn, EN : IN STD_LOGIC;   -- clock, reset, enable
35      Q : OUT STD_LOGIC);      -- output
36 END component;
37
38 SIGNAL xz: LIST_N;
39 SIGNAL mult: LIST_mult;
40 SIGNAL mult_resize: LIST_mult_resize;
41 SIGNAL sum: LIST_sum;
42
43 SIGNAL VIN_retard : std_logic;
44
45 TYPE state IS (RESET,IDLE,DATA_CYCLE1,DATA_CYCLE2,LAST_DATA1);
46 SIGNAL present_state : state;
47 SIGNAL EN_REG_1,EN_REG_OUT,EN_SHIFT,RST_INT_n,EN_FIRST_REG: STD_LOGIC;
48
49 BEGIN
50 ---- DATAPATH------------------------------------------------------
51
52 in_reg: regn     generic map (N => Nbit)
53                  port map (D => DIN, Clock => CLK, Resetn => RST_INT_n, EN =>
       EN_FIRST_REG , Q => xz(0));
```

```
54 out_reg: regn    generic map (N => Nbit)
55                  port map (D => sum(N-2)(Nbit_result downto Nbit_result-Nbit+1), Clock
       => CLK, Resetn => RST_INT_n, EN => EN_REG_OUT, Q => DOUT);
56
57 shift_reg: for i in 0 to N-2 generate
58     reg_i: regn generic map (N => Nbit)
59                  port map (D => xz(i), Q => xz(i+1), Clock => CLK, Resetn => RST_INT_n,
        EN => EN_SHIFT);
60 end generate shift_reg;
61
62 multipliers: for i in 0 to N-1 generate
63     mult(i) <= signed(b((i + 1) * Nbit - 1 downto i * Nbit)) * xz(i);
64     mult_resize(i)(Nbit_result downto 0) <= mult(i)(Nbit+Nbit-1 downto Nbit-1);
65
66 adders: for i in 0 to N-2 generate
67     i_0:    if (i=0) generate sum(i)<=mult_resize(i)+mult_resize(i+1); end generate;
68     i_etc:  if (i>0) generate sum(i)<=sum(i-1)+mult_resize(i+1); end generate;
69 end generate adders;
70
71 ---------CONTROL UNIT-----------------------------------------------
72 state_process: PROCESS (CLK, RST_n, VIN)
73 BEGIN
74 IF (RST_n='0') THEN present_state<=RESET;
75 ELSIF (CLK'EVENT AND CLK='1') THEN
76     CASE (present_state) IS
77         -- reset
78         WHEN RESET => present_state<= IDLE;
79         WHEN IDLE => IF (VIN='1') THEN present_state <= DATA_CYCLE1;
80                      ELSE present_state <= IDLE;
81                      END IF;
82         WHEN DATA_CYCLE1 => IF (VIN='0') THEN present_state <= LAST_DATA1;
83                      ELSE present_state<=DATA_CYCLE2;
84                      END IF;
85         WHEN DATA_CYCLE2 => IF (VIN='0') THEN present_state <= LAST_DATA1;
86                      ELSE present_state<=DATA_CYCLE2;
87                      END IF;
88         WHEN LAST_DATA1 =>  present_state <= IDLE;
89
90       END CASE;
91 END IF;
92 END PROCESS state_process;
93
94 output_process: PROCESS (present_state)
95 BEGIN
96 VOUT<='0';
97 EN_REG_1<='0';
98 EN_REG_OUT<='0';
99 EN_SHIFT<='0';
100 RST_INT_n<='1';
101     CASE (present_state) IS
102         -- reset
103         WHEN RESET => RST_INT_n<='0';
104         WHEN IDLE => EN_REG_1<='1';
105         WHEN DATA_CYCLE1 => EN_REG_OUT<='1';
106                             EN_SHIFT<='1';
107         WHEN DATA_CYCLE2 => EN_REG_OUT<='1';
108                             EN_SHIFT<='1';
109                             VOUT<='1';
110         WHEN LAST_DATA1 =>  VOUT<='1';
111       END CASE;
112 END PROCESS output_process;
113
```

```
114 EN_FIRST_REG<=(EN_REG_1 or EN_SHIFT);
115
116 END ARCHITECTURE behavior;
```

VHDL code - optimized FIR filter

This is the code of the optimized filter, without the files that are in common with the standard filter.

Package util

./code/fir_optimized/util.vhd

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.numeric_std.all;
4  use ieee.math_real.all;
5
6  PACKAGE util IS
7
8  CONSTANT Nbit: INTEGER := 10;
9  CONSTANT N: INTEGER := 9;-- number of coefficients
10 CONSTANT P: INTEGER := 3; --- unfolding factor
11 CONSTANT W1: INTEGER := 2; --- MAX NUMBER OF INTERMEDIATE REGISTER FOR DIN(P*K)
12 CONSTANT W2: INTEGER := 3; --- MAX NUMBER OF INTERMEDIATE REGISTER FOR DIN(P*K+1)
13 CONSTANT W3: INTEGER := 3; --- MAX NUMBER OF INTERMEDIATE REGISTER FOR DIN(P*K+2)
14
15 CONSTANT Nbit_result: INTEGER := Nbit;
16
17 CONSTANT FINAL_DELAY : integer := N + 5;
18
19 CONSTANT T: time := 20 ns; -- Clock period
20 CONSTANT start_time: time := 101 ns; --Start time of simulation
21
22 TYPE LIST_N IS ARRAY (0 to W3) OF SIGNED(Nbit-1 downto 0);
23 TYPE LIST_mult IS ARRAY (0 to N-1) OF SIGNED(Nbit+Nbit-1 downto 0);
24 TYPE LIST_mult_resize IS ARRAY (0 to N-1) OF SIGNED(Nbit downto 0);
25
26 TYPE LIST_sum_1 IS ARRAY (0 to (N/2)-1) OF SIGNED((Nbit+1)-1 downto 0);
27 TYPE LIST_sum_2 IS ARRAY (0 to ((((N*P)/2)-1)/2)+1) OF SIGNED(Nbit downto 0);
28 ----array used in folding structure-----------------------------------------
29 TYPE input_format_type          IS ARRAY (0 to P-1) OF SIGNED(Nbit-1 downto 0);
30 TYPE OUT_PIPES_TYPE             IS ARRAY (0 TO P-1) OF LIST_N;
31 TYPE mult_array_TYPE            IS ARRAY (0 TO P-1) OF LIST_mult;
32 TYPE mult_resize_array_TYPE     IS ARRAY (0 TO P-1) OF LIST_mult_resize;
33
34 END util;
```

FIR filter code

./code/fir_optimized/FIR_filter.vhd

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.numeric_std.all;
4
5  library std;
6  USE work.util.all;
7
8  ENTITY FIR_filter IS
```

```
9
10 PORT(    CLK:     in std_logic;
11          RST_n:   in std_logic;
12
13          VIN:     in std_logic;
14          VOUT:    out std_logic;
15
16          -- we have Nbit * P types for Verilog, which does not support custom types
17          DIN:     in std_logic_vector(Nbit * P - 1 downto 0);
18          DOUT:    out std_logic_vector(Nbit * P -1 downto 0);
19
20          b:       in std_logic_vector(N * Nbit - 1 downto 0));
21
22 END ENTITY FIR_filter;
23
24 ARCHITECTURE behavior OF FIR_filter IS
25
26 COMPONENT regn IS
27 GENERIC (N: INTEGER := 16);
28 PORT (D : IN SIGNED (N-1 DOWNTO 0);      -- input
29      Clock, Resetn, EN : IN STD_LOGIC;  -- clock, reset, enable
30      Q : OUT SIGNED (N-1 DOWNTO 0));     -- output
31 END COMPONENT regn;
32
33 component dff IS
34 PORT (D : IN STD_LOGIC;      -- input
35      Clock, Resetn, EN : IN STD_LOGIC;   -- clock, reset, enable
36      Q : OUT STD_LOGIC);      -- output
37 END component;
38
39 -- From util:
40 -- TYPE LIST_N IS ARRAY (0 to W3) OF SIGNED(Nbit-1 downto 0);
41 -- TYPE LIST_mult IS ARRAY (0 to N-1) OF SIGNED(Nbit+Nbit-1 downto 0);
42 -- TYPE LIST_mult_resize IS ARRAY (0 to N-1) OF SIGNED(Nbit downto 0);
43 -- TYPE OUT_PIPES_TYPE           IS ARRAY (0 TO P-1) OF LIST_N;
44 -- TYPE mult_array_TYPE              IS ARRAY (0 TO P-1) OF LIST_mult;
45 -- TYPE mult_resize_array_TYPE  IS ARRAY (0 TO P-1) OF LIST_mult_resize;
46
47 SIGNAL xz: OUT_PIPES_TYPE;
48 SIGNAL mult: mult_array_TYPE;
49 SIGNAL mult_out_pipe: mult_resize_array_TYPE;
50
51 SIGNAL mult_resize: LIST_mult_resize;
52 SIGNAL sum_1_in,sum_2_in,sum_3in,sum_1_out,sum_2_out,sum_3out: LIST_sum_1;
53 SIGNAL PIPE_REG_MULT_2_8: SIGNED(Nbit+1-1 downto 0);
54
55 TYPE sum1_reg_type is array (0 to 4-1) of SIGNED((Nbit+1)-1 downto 0);
56 TYPE sum1_type is array (0 to P-1) of sum1_reg_type;
57 SIGNAL sum1_reg_in,sum1_reg_out : sum1_type;
58
59 TYPE sum2_reg_type is array (0 to 2-1) of SIGNED((Nbit+2)-1 downto 0);
60 TYPE sum2_type is array (0 to P-1) of sum2_reg_type;
61 SIGNAL sum2_reg_in,sum2_reg_out : sum2_type;
62
63 TYPE sum3_type is array (0 to P-1) of SIGNED((Nbit+3)-1 downto 0);
64 SIGNAL sum3_reg_in,sum3_reg_out : sum3_type;
65 SIGNAL REG8_EXTENDED : SIGNED(Nbit+3-1 DOWNTO 0);
66
67 TYPE sum_final_type is array (0 to P-1) of SIGNED((Nbit+4)-1 downto 0);
68 SIGNAL sum_final : sum_final_type;
69
70 TYPE reg_8_reg_type is array (0 to 4-1) of SIGNED((Nbit+1)-1 downto 0);
```

```
71 TYPE reg_8_type is array (0 to P-1) of  reg_8_reg_type;
72 SIGNAL reg_8_value : reg_8_type;
73 SIGNAL reg_8_value_not_aggregate: signed(Nbit-1 downto 0);
74
75 TYPE pipelined_type is array (0 to 4) of std_logic;
76 SIGNAL en_shift_p,vout_p : pipelined_type;
77
78 SIGNAL VIN_retard : std_logic;
79
80 TYPE state IS (RESET,IDLE,DATA_CYCLE1,DATA_CYCLE2,LAST_DATA1);
81 SIGNAL present_state : state;
82 SIGNAL EN_REG_1,EN_REG_OUT,EN_SHIFT,RST_INT_n,EN_FIRST_REG,VOUT1 : STD_LOGIC;
83
84 SIGNAL DOUT1,DOUT2,DOUT3 :SIGNED(Nbit-1 DOWNTO 0 );
85
86 BEGIN
87 ---- DATAPATH------------------------------------------------------
88 EN_FIRST_REG<=(EN_REG_1 or EN_SHIFT);
89
90 -- 3 INPUT REGISTERS
91 in_reg_3k: regn      generic map (N => Nbit)
92                 port map (D => signed(DIN(3 * Nbit - 1 downto 2 * Nbit)), Clock => CLK
    , Resetn => RST_INT_n, EN => EN_FIRST_REG , Q => xz(0)(0));
93 in_reg_3k_plus_1: regn  generic map (N => Nbit)
94                 port map (D => signed(DIN(2 * Nbit - 1 downto Nbit)), Clock => CLK,
    Resetn => RST_INT_n, EN => EN_FIRST_REG , Q => xz(1)(0));
95 in_reg_3k_plus_2: regn  generic map (N => Nbit)
96                 port map (D => signed(DIN(Nbit -1 downto 0)), Clock => CLK, Resetn =>
    RST_INT_n, EN => EN_FIRST_REG , Q => xz(2)(0));
97
98 -- 3 OUTPUT REGISTERS
99 out_reg_1: regn generic map (N => Nbit)
100                port map (D => sum_final(0)((Nbit+1)-1 downto (Nbit+1)-1-Nbit+1),
    Clock => CLK, Resetn => RST_INT_n, EN => en_shift_p(4), Q => DOUT1(Nbit -1 downto
    0));
101 out_reg_2: regn generic map (N => Nbit)
102                port map (D => sum_final(1)((Nbit+1)-1 downto (Nbit+1)-1-Nbit+1),
    Clock => CLK, Resetn => RST_INT_n, EN => en_shift_p(4), Q => DOUT2(Nbit - 1 downto
     0));
103 out_reg_3: regn generic map (N => Nbit)
104                port map (D => sum_final(2)((Nbit+1)-1 downto (Nbit+1)-1-Nbit+1),
    Clock => CLK, Resetn => RST_INT_n, EN => en_shift_p(4), Q => DOUT3(Nbit - 1 downto
     0));
105 -- 3 output data are aligned in a single vector
106 DOUT( Nbit - 1  downto 0)       <=std_logic_vector(DOUT3);
107 DOUT(2 * Nbit-1 downto Nbit)    <=std_logic_vector(DOUT2);
108 DOUT(3 * Nbit-1 downto 2*Nbit)  <=std_logic_vector(DOUT1);
109
110 -- registers for xn[3k]
111 shift_reg_3k: for i in 0 to W1-1 generate
112     reg_i: regn generic map (N => Nbit)
113                 port map (D => xz(0)(i), Q => xz(0)(i+1), Clock => CLK, Resetn =>
    RST_INT_n, EN => EN_SHIFT);
114 end generate shift_reg_3k;
115
116 -- registers for xn[3k+1]
117 shift_reg_3k_plus_1: for i in 0 to W2-1 generate
118     reg_i: regn generic map (N => Nbit)
119                 port map (D => xz(1)(i), Q => xz(1)(i+1), Clock => CLK, Resetn =>
    RST_INT_n, EN => EN_SHIFT);
120 end generate shift_reg_3k_plus_1;
121
```

```
122 -- registers for xn[3k+2]
123 shift_reg_3k_plus_2: for i in 0 to W3-1 generate
124     reg_i: regn generic map (N => Nbit)
125                 port map (D => xz(2)(i), Q => xz(2)(i+1), Clock => CLK, Resetn =>
    RST_INT_n, EN => EN_SHIFT);
126 end generate shift_reg_3k_plus_2;
127
128 -- All the nine multiplications are performed in parallel in the three
129 -- replicas of the original datapath
130 mult(0)(0)<= signed(b((0 + 1) * Nbit - 1 downto 0 * Nbit)) * xz(0)(0);
131 mult(0)(1)<= signed(b((1 + 1) * Nbit - 1 downto 1 * Nbit)) * xz(2)(1);
132 mult(0)(2)<= signed(b((2 + 1) * Nbit - 1 downto 2 * Nbit)) * xz(1)(1);
133 mult(0)(3)<= signed(b((3 + 1) * Nbit - 1 downto 3 * Nbit)) * xz(0)(1);
134 mult(0)(4)<= signed(b((4 + 1) * Nbit - 1 downto 4 * Nbit)) * xz(2)(2);
135 mult(0)(5)<= signed(b((5 + 1) * Nbit - 1 downto 5 * Nbit)) * xz(1)(2);
136 mult(0)(6)<= signed(b((6 + 1) * Nbit - 1 downto 6 * Nbit)) * xz(0)(2);
137 mult(0)(7)<= signed(b((7 + 1) * Nbit - 1 downto 7 * Nbit)) * xz(2)(3);
138 mult(0)(8)<= signed(b((8 + 1) * Nbit - 1 downto 8 * Nbit)) * xz(1)(3);
139
140 mult(1)(0)<= signed(b((0 + 1) * Nbit - 1 downto 0 * Nbit)) * xz(1)(0);
141 mult(1)(1)<= signed(b((1 + 1) * Nbit - 1 downto 1 * Nbit)) * xz(0)(0);
142 mult(1)(2)<= signed(b((2 + 1) * Nbit - 1 downto 2 * Nbit)) * xz(2)(1);
143 mult(1)(3)<= signed(b((3 + 1) * Nbit - 1 downto 3 * Nbit)) * xz(1)(1);
144 mult(1)(4)<= signed(b((4 + 1) * Nbit - 1 downto 4 * Nbit)) * xz(0)(1);
145 mult(1)(5)<= signed(b((5 + 1) * Nbit - 1 downto 5 * Nbit)) * xz(2)(2);
146 mult(1)(6)<= signed(b((6 + 1) * Nbit - 1 downto 6 * Nbit)) * xz(1)(2);
147 mult(1)(7)<= signed(b((7 + 1) * Nbit - 1 downto 7 * Nbit)) * xz(0)(2);
148 mult(1)(8)<= signed(b((8 + 1) * Nbit - 1 downto 8 * Nbit)) * xz(2)(3);
149
150 mult(2)(0)<= signed(b((0 + 1) * Nbit - 1 downto 0 * Nbit)) * xz(2)(0);
151 mult(2)(1)<= signed(b((1 + 1) * Nbit - 1 downto 1 * Nbit)) * xz(1)(0);
152 mult(2)(2)<= signed(b((2 + 1) * Nbit - 1 downto 2 * Nbit)) * xz(0)(0);
153 mult(2)(3)<= signed(b((3 + 1) * Nbit - 1 downto 3 * Nbit)) * xz(2)(1);
154 mult(2)(4)<= signed(b((4 + 1) * Nbit - 1 downto 4 * Nbit)) * xz(1)(1);
155 mult(2)(5)<= signed(b((5 + 1) * Nbit - 1 downto 5 * Nbit)) * xz(0)(1);
156 mult(2)(6)<= signed(b((6 + 1) * Nbit - 1 downto 6 * Nbit)) * xz(2)(2);
157 mult(2)(7)<= signed(b((7 + 1) * Nbit - 1 downto 7 * Nbit)) * xz(1)(2);
158 mult(2)(8)<= signed(b((8 + 1) * Nbit - 1 downto 8 * Nbit)) * xz(0)(2);
159
160 -- registers at the output of multipliers
161 mult_reg: for i in 0 to P-1 generate
162             sec:for k in 0 to N-1 generate
163                 mult_reg_i: regn generic map (N => Nbit+1)
164                                         port map (D => mult(i)(k)(Nbit+Nbit-1 downto Nbit
    -1) , Q => mult_out_pipe(i)(k), Clock => CLK, Resetn => RST_INT_n, EN =>
    en_shift_p(0));
165             end generate;
166         end generate mult_reg;
167
168 ---- first layer of adders ---------------------------------------------------
169 adders_1_1: process(mult_out_pipe)
170             variable temp1,temp2,temp3: integer;
171             begin
172                 for i in 0 to P-1 loop
173                     for k in 0 to 3 loop
174                         temp1:=to_integer(mult_out_pipe(i)(2*k+1));
175                         temp2:=to_integer(mult_out_pipe(i)(2*k));
176                         temp3:=temp1+temp2;
177                         sum1_reg_in(i)(k)<=to_signed(temp3,Nbit+1);
178                     end loop;
179                 end loop;
180             end process;
```

```
181
182 ----second layer of adders ------------------------------------------------------------
183 adders_2: process(sum1_reg_out)
184             variable temp1,temp2,temp3: integer;
185             begin
186                 for i in 0 to P-1 loop
187                     for k in 0 to 1 loop
188                         temp1:=to_integer(sum1_reg_out(i)(2*k+1));
189                         temp2:=to_integer(sum1_reg_out(i)(2*k));
190                         temp3:=temp1+temp2;
191                         sum2_reg_in(i)(k)<=to_signed(temp3,Nbit+2);
192                     end loop;
193                 end loop;
194             end process;
195
196 --- third layer of adders-------------------------------------------------------------
197 adders_3: process(sum2_reg_out)
198             variable temp1,temp2,temp3: integer;
199             begin
200                 for i in 0 to P-1 loop
201                     temp1:=to_integer(sum2_reg_out(i)(0));
202                     temp2:=to_integer(sum2_reg_out(i)(1));
203                 temp3:=temp1+temp2;
204                 sum3_reg_in(i)<=to_signed(temp3,Nbit+3);
205
206                 end loop;
207             end process;
208
209 ---fourth layer of adders---------------------------------------------------------------
210
211 adders_4:process(reg_8_value,sum3_reg_out)
212             variable temp1,temp2,temp3: integer;
213             begin
214                 for i in 0 to P-1 loop
215                     temp1:=to_integer(reg_8_value(i)(3));
216                     temp2:=to_integer(sum3_reg_out(i));
217                 temp3:=temp1+temp2;
218                 sum_final(i)<=to_signed(temp3,Nbit+4);
219
220                 end loop;
221             end process;
222
223
224
225 ------------reg_8 shift register-----------------
226 shift_reg8: for i in 0 to P-1 generate
227                 reg_8_value(i)(0)<=mult_out_pipe(i)(8);
228             sec:for k in 0 to 2 generate
229                 shift_reg_8: regn generic map (N => Nbit+1)
230                 port map (D => reg_8_value(i)(k), Q => reg_8_value(i)(k+1), Clock =>
    CLK, Resetn => RST_INT_n, EN => en_shift_p(k+1));
231             end generate;
232         end generate shift_reg8;
233
234 -------registers between adders -------------------
235 registri_vari: for i in 0 to P-1 generate
236             sum1_cycle:for k in 0 to 3 generate
237                 sum1_reg: regn generic map (N => Nbit+1)
238                 port map (D => sum1_reg_in(i)(k), Q => sum1_reg_out(i)(k), Clock =>
    CLK, Resetn => RST_INT_n, EN => en_shift_p(1));
239             end generate;
240             sum2_cycle:for k in 0 to 1 generate
```

```
241                 sum2_reg: regn generic map (N => Nbit+2)
242                 port map (D => sum2_reg_in(i)(k), Q => sum2_reg_out(i)(k), Clock =>
    CLK, Resetn => RST_INT_n, EN => en_shift_p(2));
243             end generate;
244
245             sum3_reg: regn generic map (N => Nbit+3)
246                 port map (D => sum3_reg_in(i), Q => sum3_reg_out(i), Clock => CLK,
    Resetn => RST_INT_n, EN => en_shift_p(3));
247
248         end generate registri_vari;
249
250 ------PIPE OF CONTROL SIGNALS-----------------------------------------
251 -- registers to shift enable signal
252 pipe_registers_en_shift: for i in 0 to 3 generate
253     reg_i: dff  port map (D => en_shift_p(i), Q => en_shift_p(i+1), Clock => CLK,
    Resetn => RST_INT_n, EN => '1');
254 end generate pipe_registers_en_shift;
255
256  -- registers to shift vout signal
257 pipe_registers_vout: for i in 0 to 3 generate
258     reg_i: dff port map (D => vout_p(i), Q => vout_p(i+1), Clock => CLK, Resetn =>
    RST_INT_n, EN => '1');
259 end generate pipe_registers_vout;
260
261 en_shift_p(0)<=EN_SHIFT;
262 vout_p(0)<=VOUT1;
263 VOUT<=vout_p(4);
264
265
266 ---------CONTROL UNIT-----------------------------------------------------
267 state_process: PROCESS (CLK, RST_n, VIN)
268 BEGIN
269 IF (RST_n='0') THEN present_state<=RESET;
270 ELSIF (CLK'EVENT AND CLK='1') THEN
271     CASE (present_state) IS
272         -- reset
273         WHEN RESET => present_state<= IDLE;
274         WHEN IDLE => IF (VIN='1') THEN present_state <= DATA_CYCLE1;
275                      ELSE present_state <= IDLE;
276                      END IF;
277         WHEN DATA_CYCLE1 => IF (VIN='0') THEN present_state <= LAST_DATA1;
278                      ELSE present_state<=DATA_CYCLE2;
279                      END IF;
280         WHEN DATA_CYCLE2 => IF (VIN='0') THEN present_state <= LAST_DATA1;
281                      ELSE present_state<=DATA_CYCLE2;
282                      END IF;
283         WHEN LAST_DATA1 =>  present_state <= IDLE;
284
285       END CASE;
286 END IF;
287 END PROCESS state_process;
288
289 output_process: PROCESS (present_state)
290 BEGIN
291 VOUT1<='0';
292 EN_REG_1<='0';
293 EN_REG_OUT<='0';
294 EN_SHIFT<='0';
295 RST_INT_n<='1';
296     CASE (present_state) IS
297         -- reset
298         WHEN RESET => RST_INT_n<='0';
```

```
299             WHEN IDLE => EN_REG_1<='1';
300             WHEN DATA_CYCLE1 => EN_REG_OUT<='1';
301                                 EN_SHIFT<='1';
302             WHEN DATA_CYCLE2 => EN_REG_OUT<='1';
303                                 EN_SHIFT<='1';
304                                 VOUT1<='1';
305             WHEN LAST_DATA1 =>  VOUT1<='1';
306           END CASE;
307 END PROCESS output_process;
308
309 END ARCHITECTURE behavior;
```