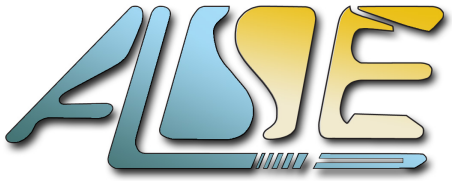
Date : Jan 2025
Version : 2025.01b



Advanced Logic Synthesis for Electronics
A.L.S.E. - https://alse-fr.com

- ALSE -
Implementing Signal Processing in Lattice FPGAs
-
Application Note #1
- sysDSP Inference -

To contact ALSE, visit the
ALSE website (FPGA.fr)
or write to : info@alse-fr.com

Table of Contents

I - Document history

Version	Date	Description
v2025.01a	Jan 2025	Initial Version
v2025.01b	Jan 2025	Iteration

II - Archive

This document is part of an archive which you can download using the link on our web site.
This archive contains :

sDSP/docs/*	.pdf : this document
sDSP/src	Source files (VHDL & SystemVerilog + SDC constraints)
sDSP/sim	VHDL test benches & Simulation scripts
sDSP/Radiant	Lattice Radiant project folder
sDSP/Radiant/sDSP	Radiant Project

III - Introduction

Lattice FPGAs are typically offering many advantages among which :
- low **cost**
- low **power** consumption
- simple and **efficient** architectures (which allowed open source tools to be developed)
- robust **silicon**
- availability under a lot of **variants** and packages.
- and the arrival of **new mid-range** families (Avant-).

However, there were in the past a number of historically weak points :
- the design **tools** (some limitations, not enough intuitive nor user-friendly)
- the mediocre quality of the **documentation**
- the limited choice of FPGA **kits**
- the Lattice **IPs**
- the limited market shares (with smaller **user base** and ecosystem as a consequence).

In the past, designing Lattice FPGAs was more difficult, demanded more know-how and experience, and required more time and efforts than for the more popular FPGAs. Leading to higher development cost (NREs). This explains why the choice of Lattice FPGAs has often been related with high volume applications, where the increased development cost was compensated by the lower cost of the FPGAs.

As of end-2024, **the situation has positively evolved** on several fronts, with the new "**Radiant**" IDE replacing "Diamond". **Radiant** is now a rather complete and modern tool, with very active development under way. Note that Lattice has also developed tools for building systems and SoC application ("Propel"), but this is not relevant to this Application note.

However, mastering Radiant does requires some efforts. The good news is there is a really **excellent and free Training video** (2h40) available to quickly understand and master Radiant ("Developing with Radiant Fundamentals") which you can find in the Lattice Insights (Education) web site.

Moreover, this Application note, the first of a series, will demonstrate through a real-world example, that a modern and efficient design methodology is applicable to the Lattice FPGAs, even if it doesn't transpire if you rely solely on the documentation !

Experimenting

At no cost, you can test the contents of this ApNote using Lattice Radiant and QuestaSim Lattice Edition. If you don't have these tools already installed, contact ALSE and we can share our 6 pages document that shows how to install and license the latest Lattice tools, then how use them to create your first project.

The archive of this ApNote contains the source code, the test benches and a Radiant project file (.rdf).

IV - The Design Challenge

We decided to build a specialized IP to implement a very large FIR filter of up to 256 non-symmetric 16-bits coefficients, with 16 bits input, while using a very small amount of resources. This is possible when the sample rate is limited (like 2 Msamples/s in our case).

As you certainly know, a FIR filter is a very simple equation : the sum of the D(n) last samples each multiplied by a K(n) coefficient. With a processing clock at 600 MHz, the above filter can be implemented using a single MAC (Multiply-Accumulate) block and a single memory block for the samples. At 150 MHz, this must be done by 4 x MACs, and a much more complex design involving careful arrangement and use of several internal memory blocks.

In the intended configuration, the output width (in the absence of saturation or truncation) is 32+8 = 40 bits.

When you read the Lattice documentation (of the Nexus families), you discover that the “DSP” blocks are quite versatile and capable, and can be configured as 18x18 multipliers followed by a 54 bits accumulator without necessitating any external logic. This is nicely matching our needs !
A Pre-Adder is also available, but we won’t need it.

Unfortunately, you won’t find anything really useful in the documentation telling you how to do this ! You will be encouraged to create a macro function with confusing parameters, but that’s not the way to go.

As we will see in this Application Note, with a solid experience and a bit of experimentation, you can use traditional and efficient inference methods and take advantage of the chosen low-cost Lattice FPGA to achieve excellent processing performance in a very small fraction of the FPGA.

V - The Silicon

We chose to use a modern and affordable family with many advantages : the Certus-NX (28nm SOI technology). But its architecture is common to many Lattice FPGA families, so this article will apply directly to **all the Lattice “Nexus” FPGAs**, which includes the CrossLink™-NX, Certus™-NX, CertusPro™-NX, and MachXO5™-NX FPGAs.

VI - The Design Tools

These families are supported by the new generation tool : **Radiant**, on which Lattice efforts are focused. Its predecessor, Diamond, has become a legacy tool and must still be used for the ECP families.

We’ve seen Radiant make progress on a lot of fronts, and one of them (since version 3.0) is the generalized and efficient support of SDC Timing Constraints, one of many key points.

As mentioned in the Introduction, if you are familiar with FPGA development and have used other tools, you can quickly master Radiant with the free training video that we recommend.

VII - The documents

We found :

- The “sysDSP User Guide for Nexus Platform” Technical Note. (June 2024)
- The “DSP Arithmetic Modules - Lattice Radiant Software” (Dec 2023)
- “Lattice Radiant Software 2024.2 User Guide” (20 Dec 2024)
  (beware : a lot of obsolete versions of this User Guide show up in the Lattice website)

As you will see, the documents available about the DSP blocks aren’t too useful when it comes to using them efficiently, but they contain information about their internal architecture.

Architecture of a DSP block

The diagram below is not particularly clear nor very helpful :

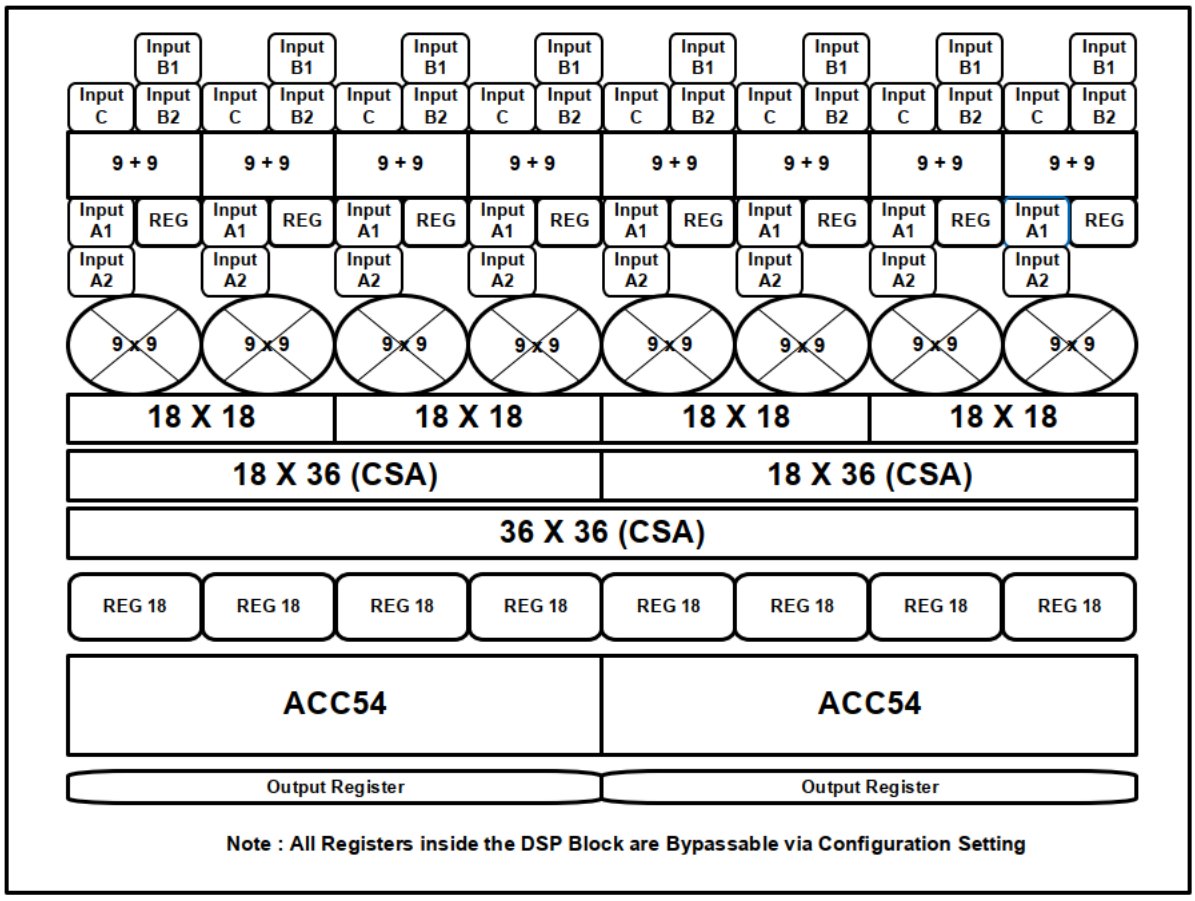


Figure 2.1. DSP Block Diagram Overview

Summary : there are pre-Adders, followed by Multipliers followed by Accumulators, with by-passable registers inserted between the stages. The operations can be configured as 9, 18 or 36 bits vectors.

There is another figure in the second documentation that is close to our use case :

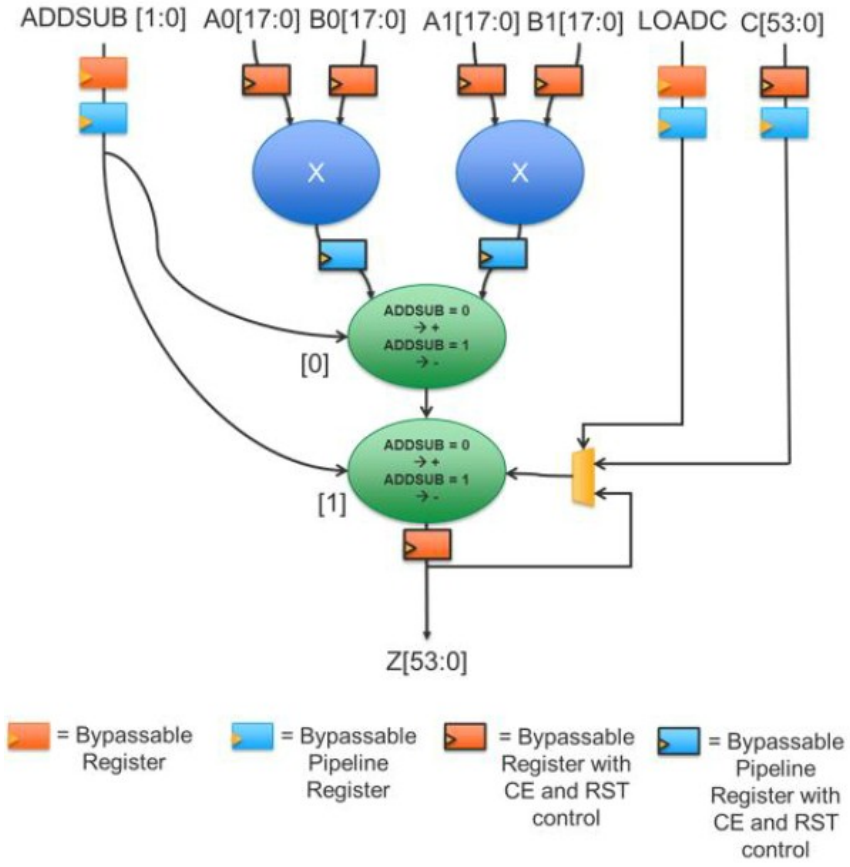


Figure 4.14. 18 × 18 Wide Multiplier and Adder/Subtractor Arithmetic Function

In this figure, it appears that the accumulator seems loadable (but our tests of inference failed to enable this feature -to load zero synchronously-).

But this is enough to hope that we could :

- Enter two 18 bits operands into the first bank of pipeline registers
- Perform a signed multiplication to obtain a signed 36 bits vector
- Enter the product in the second bank of registers
- Add the 36 bits result in the 54 bits accumulator
- Output the Accumulator through the final registers bank.
- Note that we also need the capability to clear the accumulator at any time.

VIII - Inference by HDL Code

We have coded two versions : one in VHDL and the other in SystemVerilog.

We have verified that both VHDL and SystemVerilog versions produce the exact same result after synthesis, either by Synplify Pro or with Lattice Synthesis Engine (“LSE”).

The first pipeline stage is necessary since both inputs will come from memory blocks (and maybe after traversing some switching logic), and it’s more efficient to use the sysDSP registers which are “free” as opposed to “fabric” registers.

We tried with no pipeline registers between the Multiplier and the Accumulator, but the Fmax was significantly lower and insufficient to guarantee an operation at at least 160 MHz with a slow speed grade.

And the last pipeline stage is also welcome to relieves the P&R timing constraints on the MAC output.

So, we have activated the three pipeline stages available in the sysDSP block, with a resulting *latency* of 3. But indeed, the block is capable of performing one calculation (MAC) per each clock cycle.

We have a streaming input interface with a Data-Valid for the Ain & Bin inputs : “DAVin”, and a “DAVout” output which follows three clock cycles later.

VHDL ‘93

VHDL Entity :

```
-- sDSP.vhd
--------------------------------------------------------------------------------
-- Title      : Lattice sysDSP inference - VHDL source code
-- Tool       : Lattice Radiant 2024.1.1
-- Project    : XXXX FIR
--------------------------------------------------------------------------------
-- File       : sDSP.vhd
-- Author     : Bert Cuzeau - ALSE
-- Contact    : info@alse-fr.com
-- Created    : 2024-12-28
-- Standard   : VHDL'93/02
--------------------------------------------------------------------------------
-- Description and Notes :
--   * 18x18 signed multiplier into a [53:0] Accumulator
--   * Infers a sysDSP block in MULTADDSUB configuration
--   * Synchronous clear of Accum isn't inferred correctly by LSE/SynplifyPro,
-     must be Asynch
--   * Latency = 3
--   * Synthesizes as 3 Reg + 2 LUT4s + 1 sysDSP block
--   * runs at 200+ MHz on LFD2NX-28 CABGA256 - 8_High-Performance_1.0V
--   * Comes with advanced self-testing test bench
--------------------------------------------------------------------------------

Library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;

-- ------------------------------------
    Entity sDSP is
-- ------------------------------------
  port( Clk     : in  std_logic;
        Rst     : in  std_logic;
        Clr     : in  std_logic;  -- (Asynch) clear for Accum
        Ain, Bin: in  std_logic_vector(17 downto 0);
        DAVin   : in  std_logic;
        Dout    : out std_logic_vector(53 downto 0);
        DAVout  : out std_logic
      );
begin
end entity sDSP;
```

VHDL Architecture :

```
-- -------------------------------------
    Architecture RTL of sDSP is
-- -------------------------------------
  subtype Accum_t is signed(Dout'range);
  subtype Din_t   is signed(Ain'range);
  subtype Mult_t  is signed (2*Ain'length-1 downto 0);

  signal DAVin_r, DAVin_rr, DAVin_rrr : std_logic;
  signal Ain_r   : Din_t;
  signal Bin_r   : Din_t;
  signal Accum   : Accum_t;
  signal Mult    : Mult_t;

-----\
Begin -- Architecture
-----/

-- DAV pipeline to match latency
  DAVin_r   <= '0' when Rst='1' else DAVin    when rising_edge(Clk);
  DAVin_rr  <= '0' when Rst='1' else DAVin_r  when rising_edge(Clk);
  DAVin_rrr <= '0' when Rst='1' else DAVin_rr when rising_edge(Clk);

process (Clk,Rst)
  begin
    if Rst = '1' then
      Ain_r   <= (others => '0');
      Bin_r   <= (others => '0');
      Mult    <= (others => '0');
    elsif rising_edge(Clk) then
      Ain_r   <= signed (Ain);
      Bin_r   <= signed (Bin);
      Mult    <= Ain_r * Bin_r;
    end if;
  end process;

process (Clk,Rst,Clr)
-- Synchronous Clear doesn't work for inference, have to use asynch clr
  begin
    if Rst = '1' or Clr='1'
    then
      Accum   <= (others => '0');
    elsif rising_edge(Clk) then
      if DAVin_rr='1' then  -- _rr for Latency = 3
        Accum <= Accum + Mult;
      end if;
      -- if Clr='1' then Accum <= (others => '0'); end if; -- not supported
    end if;
  end process;

DAVout <= DAVin_rrr; -- _rrr for Latency = 3
Dout   <= std_logic_vector(Accum);

end architecture RTL;
```

SystemVerilog P1800-2017

```
//------------------------------------------------------------------------------
// Title      : Lattice sysDSP inference - SystemVerilog source code
// Tool       : Lattice Radiant 2024.1.1
// Project    : XXXX FIR
//------------------------------------------------------------------------------
// File       : sDSP.sv
// Author     : Bert Cuzeau - ALSE
// Contact    : info@alse-fr.com
// Created    : 2024-12-28
// Standard   : SystemVerilog P1800-2017
//------------------------------------------------------------------------------
// Description and Notes :
//  * 18x18 signed multiplier into a [53:0] Accumulator
//  * Infers a sysDSP block in MULTADDSUB configuration
//  * Synchronous clear of Accum isn't inferred correctly by
//    LSE nor SynplifyPro, must be Asynch
//  * Latency = 3
//  * Synthesizes as 3 Reg + 2 LUT4s + 1 sysDSP block
//  * runs at 200+ MHz on LFD2NX-28 CABGA256 - 9_High-Performance_1.0V
//  * Comes with advanced self-testing test bench
//------------------------------------------------------------------------------
package DSPtypes;
  typedef logic signed [17:0] sData_t;
  typedef logic signed [2*$bits(sData_t)-1:0] Mult_t;
  typedef logic signed [53:0] Accum_t;
endpackage

// ------------------------------------
    module sDSP
// ------------------------------------
      ( input  logic             Clk,
        input  logic             Rst,  // Asynch reset
        input  logic             Clr,  // actually Asynch clear of Accumulator
        input  DSPtypes::sData_t Ain,
        input  DSPtypes::sData_t Bin,
        input  logic             DAVin,
        output DSPtypes::Accum_t Dout,
        output logic             DAVout   );

import DSPtypes::*;

logic DAVin_r, DAVin_rr, DAVin_rrr;
sData_t Ain_r, Bin_r;
Accum_t Accum;
Mult_t  Mult;


// DAV pipeline to match latency
always_ff @(posedge Rst, posedge Clk) begin
  if (Rst) begin
    DAVin_r   <= 1'b0;
    DAVin_rr  <= 1'b0;
    DAVin_rrr <= 1'b0;
  end else begin
    DAVin_r   <= DAVin;
    DAVin_rr  <= DAVin_r;
    DAVin_rrr <= DAVin_rr;
    end
end

// Register Data inputs, and Mult output
always_ff @(posedge Clk, posedge Rst) begin
    if (Rst) begin
      Ain_r   <= '0;
      Bin_r   <= '0;
      Mult    <= '0;
    end else begin
      if (DAVin) begin // to potentially reduce slightly the power consumption
        Ain_r   <= $signed (Ain);
        Bin_r   <= $signed (Bin);
      end
      Mult    <= Ain_r * Bin_r;
    end
end
```

```
logic  aCLR;
assign aClr= Rst | Clr;

// Accumulate
always_ff @(posedge Clk, posedge aClr) begin
  if (aClr) begin
    Accum <= '0;
  end else begin
    if (DAVin_rr) begin
      Accum <= Accum + Mult;
    end
  end
end

// module outputs
assign Dout   = Accum;
assign DAVout = DAVin_rrr; // _rrr for Latency = 3

endmodule : sDSP
```

IX - Instantiation using Radiant

Even if inference has our preference when it is possible (as demonstrated in the previous sections), the other way to insert an FPGA hardware feature in your design is to **instantiate** it.

If you try to insert the a DSP *primitive*, you're doomed :-)
The best way seems to use "IP Catalog", but there are many choices available which aren't doing the job, or almost but not completely, and you have the issue of not being sure of the behavior of the block you instantiate.

We tried hard to find a way to generate exactly what we wanted : 18x18 multiplication followed by 54 bits accumulator with an enable for the input data !
The closest we came with was using the "pmi_mac" template found in the "PMI Templates" section suing the "Source Template" mode. Unfortunately there was no "Enable" input controlling the accumulator.

So in this application, the best way to generate the intended function is to use inference through the code shown in this ApNote.

Note that in a lot of other cases (like PLLs or Asynch FIFOs), there is no way to use inference !
So you'll have to use instantiation, using either the "IP Catalog" or the "Source Template" modes.

X - Verification

It's really useful to create a **self-testing test bench** with a decent coverage :

- During the development to debug the code (and pipeline levels) : **RTL simulation**.
- To verify by post-synthesis simulation that the inferred sysDSP block behaves exactly like the RTL code : **post-synthesis simulation**.
- For many reasons, it's **not** useful to perform post-layout timing simulation.
  This is explained in details in our Design Training Course.

Note that Radiant can help you in the creation of the testbench skeleton based on a selected entity (or module). Many tools can also do this like Emacs (for a VHDL entity) or Teros-HDL.

We decided to write this testbench in **plain VHDL**, since SystemVerilog offers easier/better verification capabilities (like queues and randomization). The challenge is higher in VHDL.

In VHDL, we have created a **simple behavioral FIFO** to handle automatically the MAC block latency using a **shared protected variable**. This simple code is very useful and not trivial to many VHDL users. It could be enhanced by adding a generic type and by using pointers (access) for dynamic storage… but this will be for another time :-) and Google will help you locate a free library implementing generic VHDL Queues (very similar to the SystemVerilog queues).

We use intensively **pseudo-random generation** for the input Data and for DAVin recurrence.

Note that a library like OSVVM can be useful especially when it comes to Randomization and Functional Coverage, but this example demonstrates that using plain VHDL is often sufficient to implement good quality verification.

Simulation of the RTL code

Both for the VHDL and the SystemVerilog code, we use QuestaSim Lattice Edition which comes with Radiant and support both languages.

You can use another commercial simulator that supports VHDL and Verilog.

Self-Testing VHDL Test bench

Please open and understand the testbench.

We just reproduce here two snippets of the code.

Behavioral Fifo (“Queue”) :

```
-- Behavioral FIFO to handle fixed or variable latency
type Fifo_t is array (0 to Fifo_depth) of Accum_t;

type AccFifo_t is protected
  procedure Push (A : Accum_t);
  impure function Pop return Accum_t;
end protected AccFifo_t;

type AccFifo_t is protected body
  variable Fifo : Fifo_t;
  variable W,R  : natural;
  variable vA   : Accum_t;
  procedure Push (A : Accum_t) is begin
    Fifo(W):= A; W:=(W+1) mod Fifo_depth;
    assert W/=R report "Fifo overflow" severity failure;
  end procedure Push;
  impure function Pop return Accum_t is begin
    assert W/=R report "Reading from empty Fifo" severity failure;
    vA := Fifo(R);
    R:=(R+1) mod Fifo_depth;
    return vA;
  end function Pop;
end protected body AccFifo_t;

shared variable AccFifo : AccFifo_t;
```

Pushing into and popping from this behavioral Fifo consumes no time (and no delta).

For the circular buffer, we have used a fixed size array (which just needs to be deeper than the longest latency), and a determined (non-generic) data type (Accum_t), so the code is very simple. (but we’ll see it causes trouble to the simulation wizard).

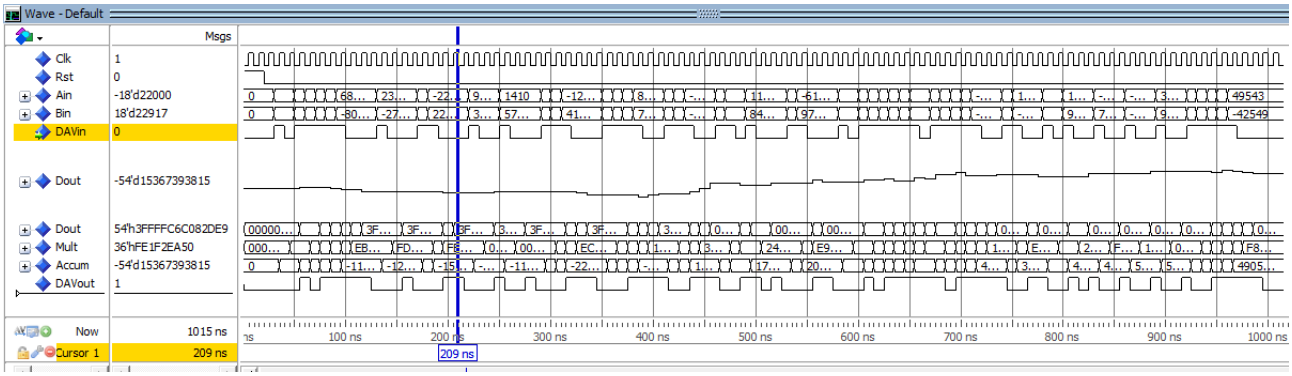
Stimulus generation :

```
Clk <= '0' when Done else not Clk after Period/2;
Rst <= '1', '0' after 2 * Period;

process -- Generate random stimuli with random DAVin
  variable S1,S2 : positive;
  variable R : real;
begin
  DAVin <= '0';
  wait until Rst='0';
  wait until Clk='0';
  for I in 1 to 50 loop
    uniform (S1,S2,R);  Ain <= R2V(R);
    uniform (S1,S2,R);  Bin <= R2V(R);
    DAVin <= '1';
    wait until Clk='0';
    DAVin <= '0';
    uniform (S1,S2,R); R := R*5.0;
    for I in 1 to integer(R-2.0) loop
      wait until Clk='0';
    end loop;
  end loop;
  -- let's make sure the queue is empty :
  for I in 1 to 4 loop wait until Clk='0'; end loop;
  Done <= true;
  report "Test is successful if no error above";
  wait;
end process;
```

Note that the testbench also records the MAC output in a text file (we’ll see why later).

RTL Simulation results




```
#
# === Simulation starting now ===
# ** Note: Test is successful if no error above
#    Time: 1010 ns  Iteration: 0  Instance: /sdsp_tb
```

We notice that the Data Valid input has some randomness to test the correct behavior of the pipeline.

The simulation is self testing, but also records the output values :

```
00000001417995
00000032C2B689
3FFFFFBD71F39B
3FFFFECF200B15
3FFFFEA2A63843
3FFFFD5871AEC9
3FFFFD3108EA69
3FFFFC8A154399
3FFFFC6C082DE9
3FFFFD43471445
3FFFFD481E87D7
3FFFFD4455BECD
3FFFFC0D2D1689
3FFFFAD0D59751
3FFFFAE2866ECB
3FFFF8D02156AD
3FFFFA586E96F9
3FFFFBCE2B47A5
3FFFFD29665395
3FFFFFBB57601E
00000387B30F8A
0000020B804DC0
etc ...
```

We'll see later that this file will be useful.

XI - Unitary Synthesis

It's time to verify how our working behavioral code is implemented by Radiant.

But first, we'll need a decent SDC constraints file.

Timing Constraints

This block is very simple : single clock domain, inputs and outputs are registered and are to be used internally in the designs, so they can be false_path'ed.

- Definition of the clock (at 160 MHz)
- All I/Os can be ignored in the timing analysis

Here is the resulting SDC file

```
# Very simple SDC Constraints file for sDSP project
# ALSE - B. Cuzeau - Jan 2025
create_clock -period 6.25 [get_ports Clk]
set_false_path -from [get_ports Rst]
set_false_path -from [get_ports Clr]
set_false_path -from [get_ports Ain*]
set_false_path -from [get_ports Bin*]
set_false_path -from [get_ports DAVin]
set_false_path -to   [get_ports Dout*]
set_false_path -to   [get_ports DAVout]
```

This file will be used by Synthesis (either LSE or SynplifyPro), by P&R, and by the STA.

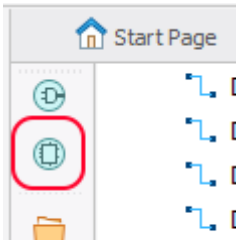
Radiant project

You can open the project provided, or create a new project :

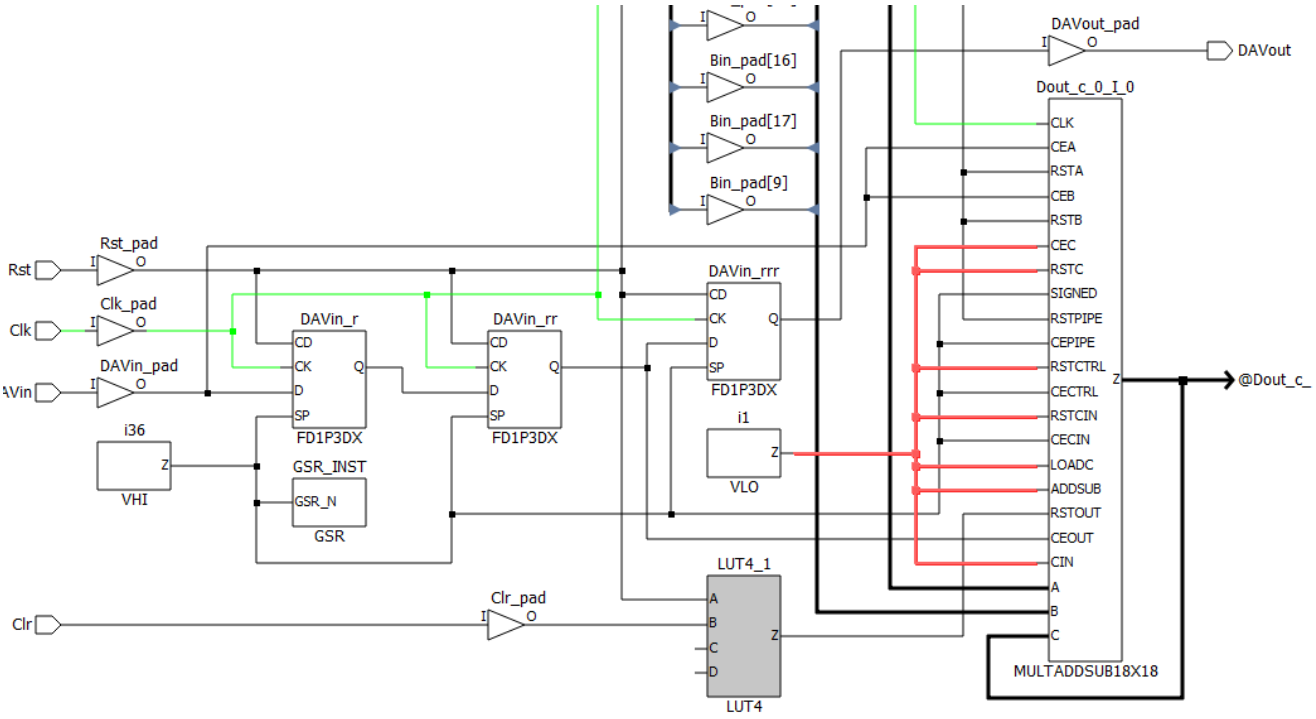
- Select the FPGA : LFD2NX-28-9BG256C
- Add the HDL source code (either sv or vhd).
- Add the above SDC constraints file
- Make sure you generate the Post-Synthesis simulation model, the timing reports and optionally the Gate-level simulation model.
- Run Synthesis and Map.

Check the Implementation result

- Launch the Netlist Analyzer
- Select the Technology View (see on the right) →
- Analyze the implemented logic



Technology View Netlist



As we can see, three Flip-Flops have been inferred for the DAV pipeline -as expected- and a sysDSP block in MULTADDSUB18x18 mode, which is exactly what we wanted and consumes no logic at all.

Timing Results

The post map timing reports indicates no error.

This is confirmed by the post-route timing check :

```
Setup at Speed Grade 9_High-Performance_1.0V Corner at 85 Degrees
   Timing Errors: 0 endpoints;  Total Negative Slack: 0.000 ns

Setup at Speed Grade 9_High-Performance_1.0V Corner at 0 Degrees
   Timing Errors: 0 endpoints;  Total Negative Slack: 0.000 ns

Hold at Speed Grade m Corner at 0 Degrees
   Timing Errors: 0 endpoints;  Total Negative Slack: 0.000 ns
```

As we can see, the MAC block can run very comfortably at the intended 160 MHz in the selected FPGA (the positive slack is reported at 2.739 ns, indicating it would run at > 200 MHz !).

Post synthesis simulation

Post Synthesis simulation is important when using (instantiating) vendor blocks to ensure they perform as you want. In this case, we could do without if we 100% trust the synthesizer to infer the correct block with correct parameters. Better safe than sorry, let simulate !

Here we hit other problems. There are tools to perform post implementation simulations (with the Simulation Wizard) but we did hit issues, and the documentation is insufficient as soon as anything goes wrong. For example, it's not trivial to understand that you should only launch the simulation wizard once, then you must open the created simulation script.

One annoyance is that all the generated simulation files are in Verilog, there is no option to generate VHDL models. It is possible to conduct mixed language simulation with the simulator that comes with Radiant, so it's not a show stopper but it is an additional challenge to deal with.

And unfortunately, the simulation wizard process is based on a custom parser ("hdlparser") to analyze the test bench : which is a very bad idea !!! In our case, this parser crashed by rejecting reasonably advanced (but absolutely legal and VHDL 93) features. It also chokes on direct instantiation.

We tried to generate a simplified VHDL test bench based on side-by-side simulation of RTL and post-synthesis models, and we gave up the not useful idea of post-layout timing simulation (the tool had trouble trying to back-annotate the vo netlist with the sdf, for reasons we didn't investigate). But it became complex to instantiate the side by side the RTL model and the post-synthesis model with the generated simulation script.

So we ended up with a simplified testbench that records the MAC output values in a text file (MACoutput.txt), which can be compared with the RTL model outputs (rtlMACoutput.txt).

XII - Conclusion

This document explains how to design efficiently Signal Processing blocks using the sysDSP hard IP available in many Lattice FPGAs while retaining a standard HDL methodology.

It demonstrates some advanced techniques to code efficient VHDL test benches.

It shows our experience during this simple development and how the issues can be worked around.

It demonstrates that the Lattice FPGAs are attractive and that the Lattice design tools have made significant progress. And we definitely recommend the 2 hours 40 minutes free video training about Radiant Fundamentals. Don't use Radiant before taking this small course.

And if you want to test the tools, you can request our Radiant Installation and Discovery document !

For any type question about this Application Note, please contact us by **E-mail**.

Bertrand CUZEAU
Chief Technology Officer A.L.S.E

Advanced Logic Synthesis for Electronics
A.L.S.E. - https://www.alse-fr.com