Copyright : Bertrand Cuzeau - ALSE
Version : 2024.02d
Contact : ALSE – http://www.alse-fr.com
E-mail : info@alse-fr.com

Advanced **L**ogic **S**ynthesis for **E**lectronics
www.ALSE-FR.com

Floating Points in FPGAs

Application Note

Copyright and Usage Notice

Introduction

This Application Note is an extract from our Digital Design course, when we consider the various Data Types that are used in a Digital Design : Integers, Binary Vectors, Signed and Unsigned Vectors, Enumerations, Arrays, Records…

We decided to add a chapter dedicated to Reals because some High-End and now even mid-Range FPGAs do include some dedicated hardware for floating point operations. However, the documentation and the support for this is not very friendly, so this chapter includes a step-by-step example.

And we thought it would be useful to publish this chapter as an Application Note to encourage users in upgrading their know-how about Reals and FPGAs.

Motivation

In which case would floating point numbers be really advantageous (over integers or fixed point numbers) ? Contrarily to what many people incorrectly believe, it's **not** at all a matter of **precision** ! Sometimes, calculation made using real numbers are said to be "infinite precision", but the term is misleading. When using double precision reals with 80 bits internal precision, it's true that it handles numbers with " lots of digits", but it's not the issue. Cryptography has to deal with huge integers and they are never represented as floating point numbers.

No, the real advantage of real numbers (even in single precision) is their **<u>dynamic range</u>** ! They can represent extremely small or extremely large values : around +/- 3×10^{38} for 32 bits FPs.

Some mathematical algorithms may generate intermediate values requiring a very high dynamic (like Matrix processing involving divisions). It is **not** at all **frequent** in signal processing.

Floating Points (reals)

There is a shortcut, often taken, including in older versions of our Digital Design Training course, stating that "Reals (floating point numbers) are not synthesizable in FPGAs".

In the absolute, this is indeed not true because FPUs (Floating Point Units) exist since a very long time ago. A famous example of these FPUs is Intel's x87 co-processors family (a clone of which was redesigned by a French Avionics company around 1995). Eventually, the FPUs are made of gates and Flip-Flops, so they *could* be synthesized in FPGAs.

Synthesizing the Floating Point *vector* is trivial : it's only a vector, with a specific and smart way to represent a real number. We'll see this in details later.

What is complex is to synthesize the *arithmetic operators* for processing floating point vectors.

Since around 2004, thanks to the excellent work of David Bishop, a set of *synthesizable* VHDL libraries were developed and proposed for adoption by IEEE to add **fixed point** and **floating points** types and the overloading of arithmetic operators for these types.
These libraries are now an integral part of IEEE VHDL 2008 and 2017, and can be used directly with modern synthesis tools (that have a serious support of VHDL 2008 or higher) for most FPGAs.

Another possibility is to use the FP IPs that the FPGA vendor may propose.

Does it mean that your next FPGA design could be using floating point numbers as the main type to represent numbers ? It's probably **<u>very</u>** **<u>un</u>**likely ! Even if your FPGA has some support for Floating Point vectors.

You can safely consider that, except in extremely specific and rare cases, you should continue to design the same way as in the previous 30+ years : making efforts to convert physical values into binary vectors (representing integers or -occasionally- fixed point numbers).

Because it is more than 100 times more efficient to do so in legacy FPGAs !
As we will see in the next paragraph.

Let's start with a first approach (IEEE float_pkg) that can be used in any FPGA, even those which offer no native support for floating point.

Floating Point Example Code

We define a very simple entity which **adds** two 32-bits IEEE single precision floating point (FP) vectors, and outputs both a 32 bits floating point and an integer vector as results.
Adding numbers seems to be a trivial operation… (it's not when it comes to FP).

Source code

```
-- fpadd.vhd
-- -----------------------------------------------------------------------
--  Floating point example (Addition)
--  Works with Quartus Pro on Cyclone 10GX (enabling VHDL 2017 support)
--  Does *not* use any vendor IP, just IEEE library
-- -----------------------------------------------------------------------
-- B. Cuzeau - ALSE - info@alse-fr.com
-- uses pipelines on I/Os to get optimal Fmax
-- and more pipelines to enable Physical Synthesis
--

Library IEEE;
    use IEEE.STD_LOGIC_1164.all;
    use IEEE.numeric_std.all;
    use IEEE.float_pkg.all;

-- ------------------------
    Entity fpadd is
-- ------------------------
  port (Clk  : std_logic;
        A, B : in  float32;
        Q    : out float32 := (others=>'0');
        S32  : out signed (31 downto 0) := (others=>'0')
     );
end entity fpadd;

-- -----------------------------
    Architecture RTL of fpadd is
-- -----------------------------
  signal Ar,Br,Qi,Qii : float32;
Begin
  Ar  <= A       when rising_edge(Clk);
  Br  <= B       when rising_edge(Clk);
  Qii <= Ar + Br when rising_edge(Clk);
  Qi  <= Qii     when rising_edge(Clk);
  Q   <= Qi      when rising_edge(Clk);
--S32 <= signed(to_slv(A)) + signed(to_slv(B)) when rising_edge(Clk);
end architecture RTL;
```

Only the code in **bold** is specific to the floating point library.

S32 is a simple binary adder which outputs A + B (as signed vectors).
This optional output is to compare its complexity with the FP adder.

Test bench

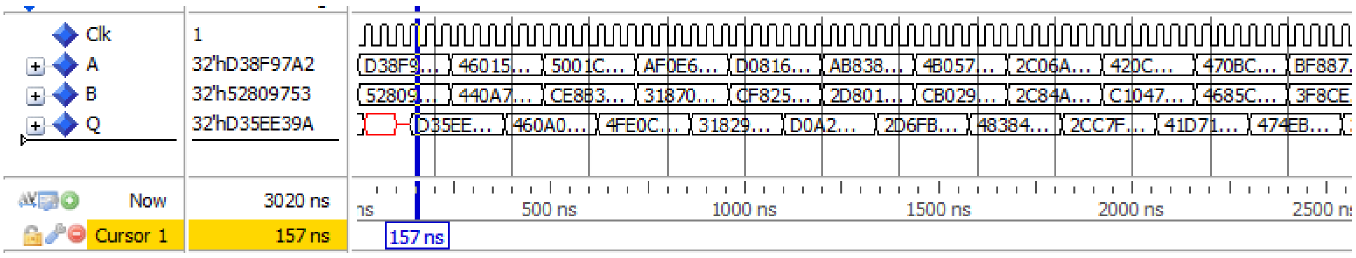
We reproduce the self-verifying testbench which generates some random values with constraints (to have A & B within a reasonable distance). SystemVerilog would have been much easier but it's doable in VHDL without too much pain.

```
-- ========================================
--  Testbench for Floating Point Adder
-- ========================================
-- Bert Cuzeau - ALSE - info@alse-fr.com
-- Must be compiled as VHDL 2008
-- synopsys translate_off
    use STD.textio.all;
Library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;
    use IEEE.math_real.all;
    use IEEE.float_pkg.all;

Entity fpadd_TB is end;
```

```
Architecture TEST of fpadd_TB is
  signal Clk : std_logic := '0';
  signal A,B,Q : float32;
  signal S32 : signed(31 downto 0);
Begin
Clk <= '0' when now > 3000 ns else not Clk after 20 ns;
UUT : Entity work.fpadd port map (Clk=>Clk, A=>A, B=>B, Q=>Q,S32=>S32);
process  -- Simulus + Checker
  variable S1,S2 : positive;
  variable Sg : std_logic;
  variable Exp, Mant : integer;
  variable R  : real;
  variable L  : Line;
begin
  uniform (S1,S2,R);  -- exponent -40/+40
  Exp := integer((R * 80.0) - 40.0) + 127;-- exponent 2**-40/+40

  uniform (S1,S2,R);
  Mant := integer(R*2.0**20);
  uniform (S1,S2,R);
  Sg := '1' when R>0.5 else '0';
  A <= to_float(Sg & std_logic_vector(to_unsigned(Exp,8)) &
       std_logic_vector(to_unsigned(Mant,23)));

  uniform (S1,S2,R);
  Exp := Exp + integer(R*12.0 - 6.0);
  uniform (S1,S2,R);
  Mant := integer(R*2.0**20);
  uniform (S1,S2,R);
  Sg := '1' when R>0.5 else '0';
  B <=to_float(Sg & std_logic_vector(to_unsigned(Exp,8)) &
std_logic_vector(to_unsigned(Mant,23)));

  for i in 1 to 5 loop wait until Clk='0'; end loop;

  write(L,now,left,10); write(L,ht);
  write(L,to_real(A));  write(L,ht&" + ");
  write(L,to_real(B));  write(L,ht&" = ");
  write(L,to_real(Q));
  writeline(output,L);
  assert Q=A+B report "Error in Floating Point Addition result";
  wait until Clk='0';

end process;
end architecture TEST;
-- synopsys translate_on
```

The output (transcript) is :

```
# Loading work.fpadd(rtl)#1
# 200 ns     -1.233449e+12     + 2.761473e+11      = -9.573013e+11
# 440 ns     8.278705e+03      + 5.539173e+02      = 8.832622e+03
# 680 ns     8.709889e+09      + -1.167671e+09     = 7.542218e+09
# 920 ns     -1.295461e-10     + 3.930730e-09      = 3.801184e-09
# 1160 ns    -1.736930e+10     + -4.374267e+09     = -2.174357e+10
# 1400 ns    -9.347047e-13     + 1.456148e-11      = 1.362677e-11
# 1640 ns    8.745866e+06      + -8.557191e+06     = 1.886750e+05
# 1880 ns    1.913174e-12      + 3.770081e-12      = 5.683255e-12
# 2120 ns    3.516499e+01      + -8.277925e+00     = 2.688706e+01
# 2360 ns    3.578680e+04      + 1.712799e+04      = 5.291479e+04
# 2600 ns    -1.066163e+00     + 1.100730e+00      = 3.456724e-02
# 2840 ns    -8.771853e-03     + -4.064315e-03     = -1.283617e-02
```


Clk
1
A
32'hD38F97A2
B
32'h52809753
Q
32'hD35EE39A
D38F9... 46015... 5001C... AF0E6... D0816... AB838... 4B057... 2C06A... 420C... 470BC... BF887.
52809... 440A7... CE8B3... 31870... CF825... 2D801... CB029... 2C84A... C1047... 4685C... 3F8CE
D35EE... 460A0... 4FE0C... 31829... D0A2... 2D6FB... 48384... 2CC7F... 41D71... 474EB...
Now
3020 ns
Cursor 1
157 ns
500 ns
1000 ns
1500 ns
2000 ns
2500 ns
157 ns

Summary

As we can see, the code is extremely simple and straightforward.

- ✔ The source code of all IEEE VHDL libraries is now open source.
  You can also visit David Bishop's Git where you can find details and adaptations he made for particular tools : https://github.com/FPHDL/fphdl. In particular, you may read his excellent User Guide : `Float_ug.pdf`.
- ✔ It works nicely in simulation : ModelSim and QuestaSim, like other simulators can directly display the vectors as real numbers (simply select *radix = float32*).
- ✔ It works "nicely" for synthesis too and inside all kinds of FPGAs, even for low cost families.

Let's note however that when it comes to Quartus, the "Standard" version which addresses the FPGAs prior to 20 nm (up the the V families) unfortunately doesn't provide an extensive support of VHDL 2008 (and no VHDL 2017). So it will prevent you from using the **`float_pkg`** library.
That's why we used Quartus Pro, which does support this library and has a free version for the Cyclone 10 GX family (20 nm).

There is another way (than float_pkg), with better performances, to implement Floating point operators and functions : using the IPs proposed by the FPGA vendor.
With Altera/Intel the list is impressive, and is applicable to old and/or low cost FPGAs supported by Quartus Standard. But indeed this is a solution that is not standard nor portable to other vendors.
If you want to look at this solution, you can read the Intel Floating-Point IP Cores User Guide.

So, if the IEEE float_pkg library can be synthesized into any FPGA, and if your FPGA is offering a large set of Floating Point IPs, *what is the catch ?*
See below...

Implementation Results

The synthesis and timing reports (for 10CX220YF780E5) are eloquent.

```
Registered 32 bits Integer Adder : 16.5 ALMs (33 ALUTs) - FMax 512 MHz
Float32 Adder : 594 ALMS, 96 regs, FMax 74 MHz (778 ALUTs) | Single pipeline
Float32 Adder : 671 ALMS,309 regs, FMax 82 MHz (891 ALUTs) | + 2 pipeline stages
```

As compared to a binary adder, the FP adder is **~37 times bigger** and about **7 times slower**, an overall disadvantage of more than 200x.

Note : the timings are enhanced (82 MHz vs 74) by using more pipelining (and 309 regs vs 96) with aggressive physical synthesis (register retiming). We could compare with the implementation proposed by Intel offering higher Fmax.

But **in any case**, the penalty for using floating points vectors is huge !

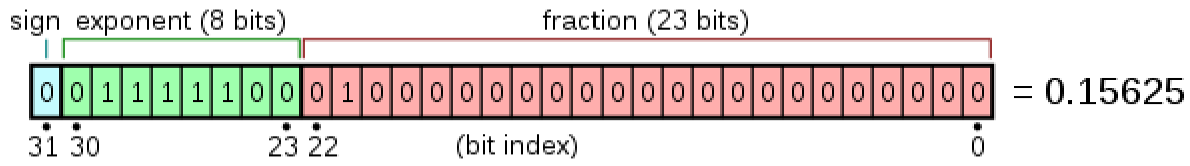
But we will not stop here !

Let's now take a look at the embedded DSP block available in the FPGA we have chosen, and how to use its "Native Floating Point" support. For this, we are designing a FP **Multiplier**.

Floating Point using advanced DSP blocks

Starting with the 20 nm families (Arria10 and Cyclone 10 GX) and beyond (Stratix 10 and Agilex 7), the embedded DSP block now offers a "*Native Floating Point*" mode which can multiply two single precision (32 bits) floating point vectors (IEEE 754).

The format is (from Wikipedia) :



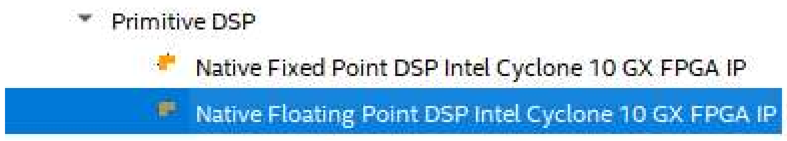
and the real value is :

$$\text{value} = (-1)^{\text{sign}} \times 2^{(E-127)} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

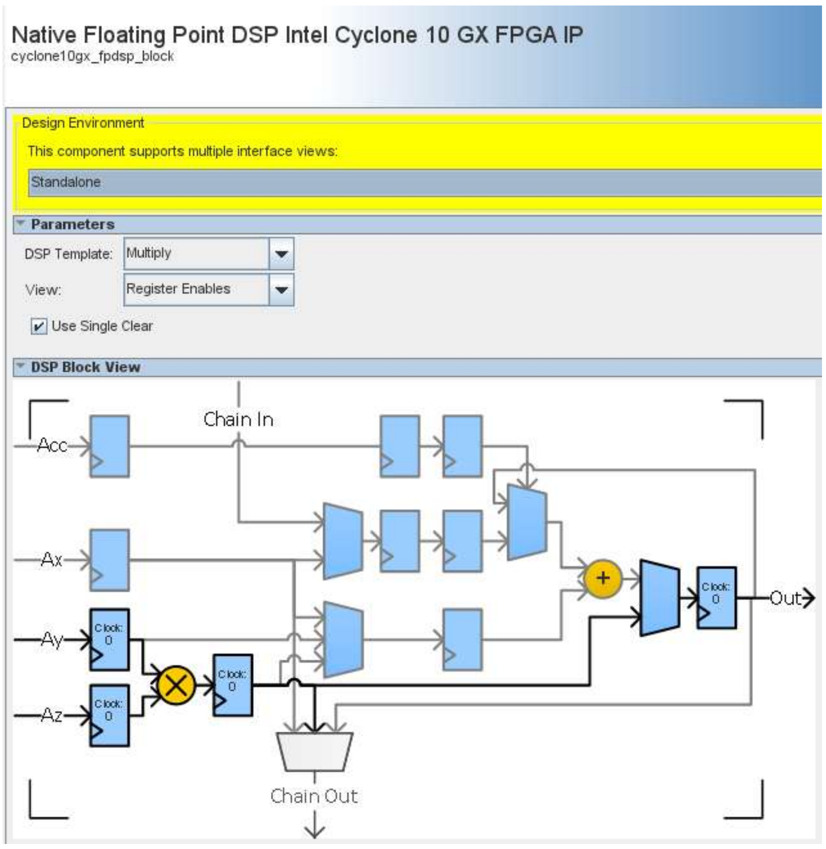
In the float_pkg package, this type is **float32**.

Let try to use the DSP block for multiplying two float32.
Bad News : **no inference** is possible, we have to **instantiate** the macro-function.

So, in Quartus Pro, we open the **IP catalog** and we select :


Primitive DSP
Native Fixed Point DSP Intel Cyclone 10 GX FPGA IP
Native Floating Point DSP Intel Cyclone 10 GX FPGA IP

which we set up simply as below :



Native Floating Point DSP Intel Cyclone 10 GX FPGA IP
cyclone10gx_fpdsp_block

Design Environment
This component supports multiple interface views:
Standalone

Parameters
DSP Template: Multiply
View: Register Enables
[✓] Use Single Clear

DSP Block View

We notice in the above diagram that we will have a latency of 3 clock cycles.

Design

When we generate the “IP” we get an entity FP_mult (wrapper) which we can instantiate in our design.

Here is the code for this simple project :

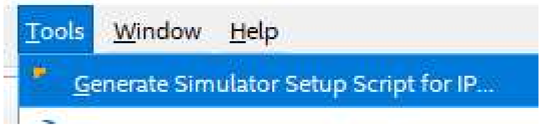
```
-- fpmult.vhd
-- ----------------------------------------------------------
--  Cyclone 10 gx Floating Point Mult Example (DSP based)
-- ----------------------------------------------------------
-- Author : Bert CUZEAU - cuzeau@alse-fr.com
-- Date   : Feb 2024
--
Library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;
    use IEEE.float_pkg.all;

-- ---------------------------------------------------------------------------
    Entity fpmult is
-- ---------------------------------------------------------------------------
  port (  Clk  : in  std_logic;
          A, B : in  std_logic_vector(31 downto 0);
          P, Q : out std_logic_vector(31 downto 0)   );
end entity fpmult;

-- -------------------------------------------------------------------
    Architecture RTL of fpmult is
-- -------------------------------------------------------------------
attribute chip_pin : string;
attribute chip_pin of Clk : signal is "J23";

component FP_mult
  port (
    clk0   : in  std_logic                     := '0';              --   clk0.clk
    ena    : in  std_logic                     := '0';              --    ena.ena
    aclr0  : in  std_logic                     := '0';              --  aclr0.reset
    result : out std_logic_vector(31 downto 0);                     -- result.result
    ay     : in  std_logic_vector(31 downto 0) := (others => '0');  --     ay.ay
    az     : in  std_logic_vector(31 downto 0) := (others => '0') );--     az.az
end component FP_mult;

signal Ar, Br, Pr : std_logic_vector(31 downto 0);

Begin

Ar <= A when rising_edge(Clk);
Br <= B when rising_edge(Clk);

-- Instantiate the FP multiplier
FPM: FP_mult port map (
    clk0   => Clk,
    ena    => '1',
    aclr0  => '0',
    result => Pr,
    ay     => Ar,
    az     => Br    );
P <= Pr when rising_edge(Clk);

-- Doesn't infer a DSP : infers logic based on float_pkg
-- comment out this line below to only synthesize the DSP block above.
Q <= to_slv(to_float(Ar) * to_float(Br)) when rising_edge(Clk);
-- use this if you remove the line above :
-- Q <= (others=>'0');

end architecture RTL;
```

Note : we pipeline the inputs and output to avoid I/O constraints and to fairly estimate the Fmax performance of this multiplier.

The code at the bottom is an attempt* to infer the multiplier, but this code (like in the previous section with the FP Adder) does synthesize and work, but it doesn’t infer a DSP block, only standard logic, and necessarily a lot of them (ALMs). And indeed, the resulting logic is slow (poor Fmax).

* : this inference attempt is doomed anyway since the DSP infers a latency of 3 clock cycles.

Simulation



What we are showing here is an overkill in this very simple example, but the usual method to prepare for simulation is to use Quartus for generating the compilation commands to reach the proper IP files generated by the Wizard.

This action does create a **`msim_setup.tcl`** script which you can open and inspect. This script will take care of automatically compiling all that is needed to simulate the generated IPs. At the beginning of this script, as comments, you will see a template to create your simulation script, and that is what we are doing now.

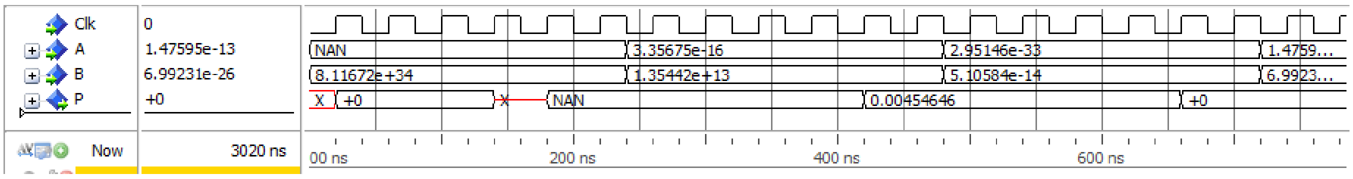
Based on the provided template, we create the **`simu.do`** simulation script file to call the setup script, compile your own files, and perform the simulation :

```
# Simulation script

# Base directory for simulation
set QSYS_SIMDIR ../sim

# Source the generated IP simulation script
source $QSYS_SIMDIR/mentor/msim_setup.tcl

# set USER_DEFINED_COMPILE_OPTIONS <compilation options>
# set USER_DEFINED_VHDL_COMPILE_OPTIONS <compilation options for VHDL>
# set USER_DEFINED_VERILOG_COMPILE_OPTIONS <compilation options for Verilog>

# Compile the simulation library + Compile Quartus-generated IP simulation files
dev_com
com

# compile all design files and testbench files
vcom ../../../src/fpmult.vhd

set TOP_LEVEL_NAME fpmult_TB
set USER_DEFINED_ELAB_OPTIONS "-voptargs=+acc"

# Launch the simulation
elab

# Populate the waveform and Run the simulation
do wfp.do
run -a
wave zoomfull
```

Launch **QuestaSim Intel Edition**.

`File > Change directory >` select the simulation directory (where you have **`simu.do`**).

```
do simu.do
```

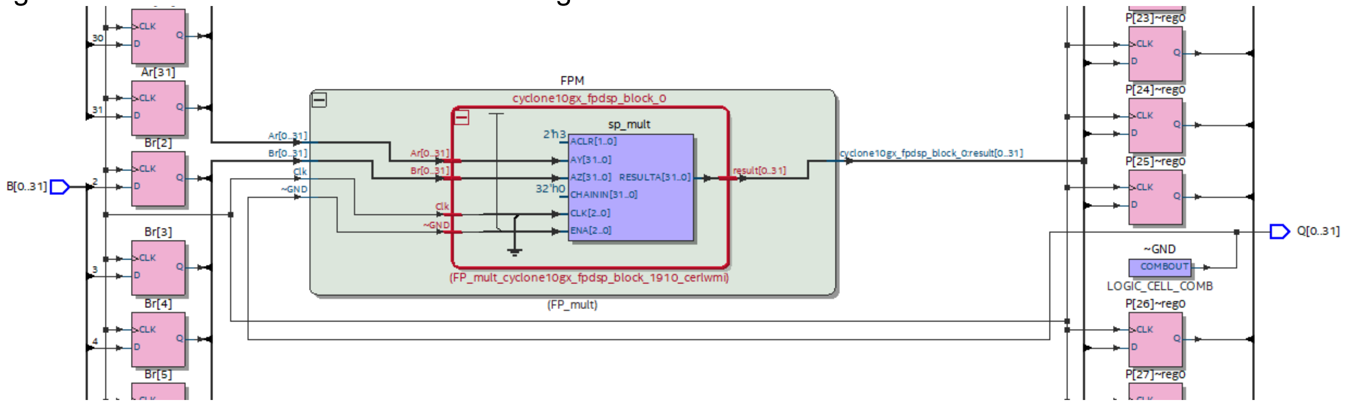


```
 ** Error: Error in floating point multiply
#    Time: 200 ns  Iteration: 0  Instance: /fpmult_tb
# 440 ns        3.356746e-16  x 1.354425e+13   = 4.546460e-03
# 680 ns        2.951459e-33  x 5.105841e-14   = 0.000000e+00
# 920 ns        1.475946e-13  x 6.992314e-26   = 0.000000e+00
# ** Error: Error in floating point multiply
#    Time: 920 ns  Iteration: 0  Instance: /fpmult_tb
# 1160 ns       1.809377e-07  x 3.776970e-13   = 6.833963e-20
# 1400 ns       4.754167e-15  x 4.236727e-19   = 2.014210e-33
# 1640 ns       9.486771e+24  x 1.015332e-21   = 9.632225e+03
# 1880 ns       7.374053e+07  x 4.887282e+02   = 3.603908e+10
# 2120 ns       3.904103e-08  x 1.566372e+31   = 6.115276e+23
# 2360 ns       8.330347e-25  x 6.041107e+23   = 5.032451e-01
# 2600 ns       4.412106e+32  x 5.676503e+25   = 1.000000e+308
# 2840 ns       5.969615e+29  x 1.338291e-27   = 7.989082e+02
```

It's a self testing testbench which generates random positive 32 bits vectors, and you may see some disagreement for overflow or underflow inputs or results (NAN "not a number" or +INF "infinite").

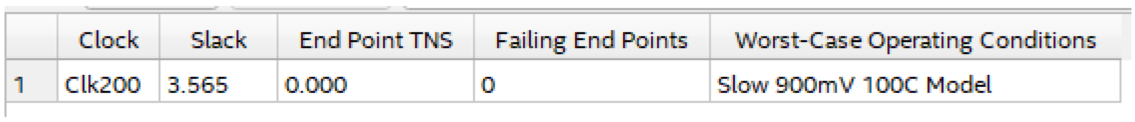
Synthesis

After running a full compilation in Quartus, open the Technology Map Viewer : you will see that no logic at all was used for the floating point multiplication : only a DSP block. Indeed, you will find the pipeline registers that have been inserted in the design.



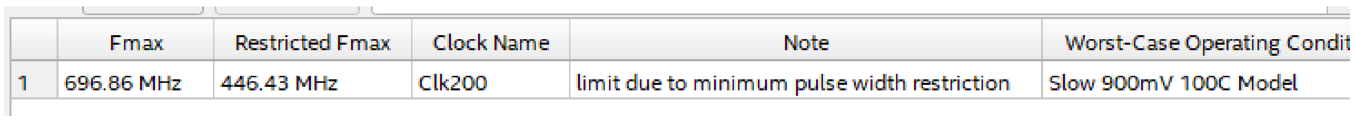
Timing results

With a 200 MHz clock, we see a very comfortable margin :



	Clock	Slack	End Point TNS	Failing End Points	Worst-Case Operating Conditions
1	Clk200	3.565	0.000	0	Slow 900mV 100C Model

In fact, this FPGA could run this design at 446 MHz :



	Fmax	Restricted Fmax	Clock Name	Note	Worst-Case Operating Condit
1	696.86 MHz	446.43 MHz	Clk200	limit due to minimum pulse width restriction	Slow 900mV 100C Model

Conclusion

If you use the high-end or mid-range Intel FPGAs (and necessarily Quartus Pro), it becomes feasible to use single-precision floating point vectors. In this context, you'll also notice that some IPs will propose using Floating Points (like the FFTs).

However, it still remains easier and much more productive to use fixed point or integer arithmetic whenever possible.

Bertrand CUZEAU

Founder and Chief Technology Officer A.L.S.E

E-mail = info@alse-fr.com

Table of Contents