# Writing RTL Code for Virtex-4 DSP48 Blocks with XST 8.1i

Writing RTL code for your DSP applications is easy and efficient.

by Edgard Garcia
Xilinx Consultant/Designer
Multi Video Designs.
*edgard.garcia@mvd-fpga.com*

The Xilinx® Virtex™-4 family introduced a new high-performance concept for fast and complex DSP algorithm implementation. The XtremeDSP™ Design Considerations User Guide, available on the Xilinx website (*www.xilinx.com/bvdocs/userguides/ug073.pdf*), describes how you can take advantage of the DSP48 architecture and includes several examples.

When you have to develop a real DSP application, you can of course instantiate each DSP48 block and assign their respective attribute values to obtain the correct behavior. But did you know you can also infer most of the useful DSP48 configurations by writing very simple RTL code?

Developing DSP algorithms in VHDL (or Verilog) is a nice way to maintain designs over a long period of time, but the synthesis results must meet your performance requirements. In this article, I will show you how to write RTL code to take full advantage of Virtex-4 DSP48 blocks.

## DSP48 Architecture

The Virtex-4 DSP48 architecture is extensively described in the XtremeDSP User Guide. Let's start, however, with an overview of some very important aspects of DSP48 blocks:

- DSP48 blocks have two18-bit inputs to feed the multiplier. If you want to work with unsigned data, 17 bits is the maximum width of the multiplier inputs. Don't forget to expand the unsigned data/coefficients by concatenating one or more '0' to the most significant bit (MSB). Similarly, if using the adder/subtracter, its inputs and output will have to be 48 bits or less for signed arithmetic and 47 bits or less for unsigned.

For the examples described in this article, we will use signed data. You will have to use the IEEE.STD_LOGIC_SIGNED package.

- Another important parameter for describing DSP behavior for Virtex-4 DSP48 blocks is that all DSP48 internal registers have a synchronous reset (using asynchronous reset will prevent the synthesis tool from using the DSP48 internal registers). The reset functionality has priority, regardless of OpCode or other control inputs.

- It is important to note that the last stage of the adder/subtracter can be driven dynamically to take a 48-bit input (from the output stage feedback or from the DSP48 C or Pcin input) and to add or subtract another 48- or 36-bit input (originating for most common cases from the multiplier output).

## Basic Examples

**1. Multiplier_accumulator.** This commonly used function is our first example, useful for FIR filters and other DSP functions. Here is the source code:

```
library IEEE;            use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;                          — Signed arithmetic is used

entity MULT_ACC is
   Port ( CK : in std_logic;
        RST : in std_logic;                             — Synchronous reset
        Ain, Bin : in std_logic_vector(17 downto 0);    — A and B inputs of the multiplier
        S : out std_logic_vector(47 downto 0)       );  — Accumulator output
end MULT_ACC;

architecture Behavioral of MULT_ACC is

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          ACC <= (others => '0');
                          else                        ACC <= ACC + (AIN * BIN);
                          end if;
             end if;
end process;

S <= ACC;

end Behavioral;
```

This example will be synthesized into a single DSP48 block – no other logic resource is necessary. The performance is about 180-200 MHz, depending on placement and routing.

**2. Fully pipelined Multiplier_accumulator.** If you need more performance and less dependency on place and route tools, you can still improve the performance of the Multiplier_accumulator. The DSP48 blocks have internal input registers (zero, one, or two stages for A and B inputs), as well as one selectable multiplier output register. The following RTL code uses one level of registers at the A and B inputs, as well as the multiplier output register:

```
library IEEE;            use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;                        — Signed arithmetic is used

entity MULT_ACC is
   Port ( CK : in std_logic;
        RST : in std_logic;                           — Synchronous reset
        Ain, Bin : in std_logic_vector(17 downto 0);  — A and B inputs of the multiplier
        S : out std_logic_vector(47 downto 0)       ); — Accumulator output
end MULT_ACC;

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;
```

This example will be synthesized by using just a single DSP block. You can take advantage of the internal registers to greatly improve performance to more than 400 MHz for the slowest Virtex-4 speed grade, independent of the implementation (place and route) tools.

**3. Fully pipelined Loadable_Multiplier_accumulator.** You can improve the design further by using a loadable multiplier accumulator. For more details, please refer to the class material of the Xilinx course, "DSP Implementation Techniques for Xilinx FPGAs" (*www.xilinx.com/support/training/abstracts/dsp-implementation.htm*). Let's modify the previous code for the load functionality:

```
library IEEE;            use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;                    — Signed arithmetic is used

entity MULT_ACC_LD is
   Port ( CK : in std_logic;
        RST : in std_logic;                            — Synchronous reset
        Ain, Bin : in std_logic_vector(17 downto 0);   — A and B inputs of the multiplier
        LOAD : in std_logic;                           — Active high LOAD command
        S : out std_logic_vector(47 downto 0)       ); — Accumulator output
end MULT_ACC_LD;

architecture Behavioral of MULT_ACC_LD 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

— 48 bit "ZERO" constant used for MULTR sign extension to 48 bits
constant ZERO : std_logic_vector(47 downto 0) := (others => '0');

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;
                                       if LOAD = '1'  then
                                                ACC <= ZERO + MULTR;  — OpCode = x05
                                       else
                                                ACC <= ACC + MULTR;   — OpCode = x25
                                       end if;
                          end if;
             end if;
end process;

S <= ACC;

end Behavioral;
```

**4. Multiplier_accumulator_or_adder.** This is another useful version of the multiplier accumulator. It is useful for multiplications of data buses of more than 18 bits (see Figure 1-18 in the XtremeDSP User Guide). Here is the RTL code:

```
library IEEE;                use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;                  — Signed arithmetic is used

entity MULT_ACC_ADD is
    Port ( CK : in std_logic;
           RST : in std_logic;
           SEL : in std_logic;
           A_in, 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;

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 (Notes 2, 3)
                       end if;

                end if;
     end if;
end process;

S <= Pout;

end Behavioral;
```

Note that the synthesis results are not currently as optimized as we could expect with XST 8.1. Some combinatorial logic will be used to implement the multiplexer between C_in and Pout, while the same function was available inside the DSP48 block. The performance is still 220 MHz for the -10 speed grade, and 270+ MHz for -12. However, Synplify Pro 8.2 provides the ideal implementation with the same RTL code.

Note1 : Adding ZERO to Pout is equivalent to the previously described load function.

Note 2 : You can also use the 17-bit right shift on Pout by changing this line as follows (at this time, this feature is supported only by Synplicity Synplify Pro 8.2):

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

Note 3 : If for any reason you do not want to use the output register of the multiplier, you can write:

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

instead of declaring a combinatorial multiplier output. The resulting RTL code is also more compact.

**5. Symmetric rounding.** Another simple but useful example is a multiplier with symmetric rounding (see Table 1-9 in the XtremeDSP User Guide). Assuming that you want to round the result of the multiplication Ain x Bin to 20 bits, the following RTL code will be synthesized in just one DSP48 block and one slice:

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

entity ROUNDING is
    Port ( CK : in std_logic;
           RST : in std_logic;
           Ain, 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;
```

This example will also work at 400 MHz for the Virtex-4 -10 speed grade device and 500 MHz for the -12 speed grade device. Only one LUT and its associated slice flip-flop is used, as the second flip-flop is pushed inside the DSP48 block for carry input.

All of these examples can be used in a wide range of applications. You can see that they are very efficiently synthesized, and all of the logic is mapped into the DSP48 blocks. The performance for each of these DSP functions is independent of the place and route tools.

To make it easier for synthesis tools to recognize the DSP48 structure, it is important to write the code in a simple way, giving your tools the best option to pack your desired functions into each DSP48 block. For this reason, each code has been written in a single process.

The more simple and compact your RTL code, the more efficient the synthesis result. Of course, depending on your synthesis tool, other alternatives can also give you excellent results, but they will be more dependent on the synthesis tools.

## Higher Complexity Designs

What happens when you need more complex DSP functions? You can use a similar approach for many complex DSP algorithm implementations by describing each block separately to ensure optimal synthesis results.

You will find many other examples, most of them directly related to those explained in their algorithmic and schematic form, in the XtremeDSP User Guide.

## Conclusion

This article is excerpted from the application note, "Virtex-4 DSP48 Inference," which is available at *www.mvd-fpga.com/en/publi_V4_DSP48.html*.

The application note includes additional examples, such as:

- Single DSP slice 35 x 18 multiplier (Figure 1-18 in the XtremeDSP User Guide)
- Single DSP slice 35 x 35 multiplier (Figure 1-19 in the XtremeDSP User Guide)
- Fully pipelined complex 18 x 18 multiplier (Figure 1-22 in the XtremeDSP User Guide)
- High-speed FIR filter (Figure 1-17 in the XtremeDSP User Guide)

The application note also describes many of the important features of DSP48 blocks supported by XST 8.1i and Synplify-Pro 8.2. MAP reports, Timing Analyzer reports, and a detailed view of the FPGA Editor show the efficiency of the synthesis and implementation tools. You can also see how the cascade chain between adjacent DSP48 slices is used to improve both performance and power consumption. Almost all of these widely used configurations provide the best implementation results – in terms of resources used as well as performance. However, some remaining limitations are also described. We expect these few points to be resolved in future releases.

For more information, see the XtremeDSP Design Considerations User Guide at *www.xilinx.com/bvdocs/userguides/ug073.pdf.* The methodology is clearly explained and implementation results analyzed in detail, with ISE™ software tools like Timing Analyzer and FPGA Editor.

*Multi Video Designs (MVD) is a training and design center specializing in FPGA designs, PowerPC™ processors, RTOS for embedded/real-time applications, and high-speed buses like PCI Express and RapidIO. MVD as an Approved Training Partner and a member of the Xilinx XPERTS program, with offices in France, Spain, and South America.*

## XST Support for DSP48 Inference

XST, the synthesis engine included with the Xilinx® ISE™ toolset, contains extensive support for inference of DSP48 macros. A number of macro functions are recognized and mapped to these dedicated resources, including adders, subtracters, multipliers, and accumulators, as well as combinations like multiply-add and multiply-accumulate (MAC). Register stages can be absorbed into the DSP48 blocks, and direct connect resources are used to cascade large or multiple functions.

Macro implementation on DSP48 blocks is controlled by the USE_DSP48 constraint with a default value of auto. In auto mode, XST attempts to implement all aforementioned macros except adders or subtracters on DSP48 resources. To push adders or subtracters into a DSP48, set the USE_DSP48 constraint value to yes.

XST performs automatic resource control in auto mode for all macros except adders and subtracters. In this mode you can control the number of available DSP48 resources for synthesis using the DSP_UTILIZATION_RATIO constraint, specifying either a percentage or absolute number. By default, XST tries to utilize, as much as possible, all available DSP48 resources within a given device.

With the 8.1i release of ISE software, XST has introduced further enhancements to its DSP support. XST can now infer loadable accumulators and MACs, which are critical for filter applications. XST can recognize chains of complex filters or multipliers – even across hierarchical boundaries – and will use dedicated fast connections to build these DSP48 chains. The Register Balancing optimization feature will consider the registers with DSP48 blocks when optimizing clock frequencies. Consult the XST User Guide (*http://toolbox.xilinx.com/docsan/xilinx7/books/docs/xst/xst.pdf*) for details about coding styles, and watch the synthesis reports for specific implementation results for your Virtex™-4 designs.

*– David Dye*
*Senior Technical Marketing Engineer*
*Xilinx, Inc.*