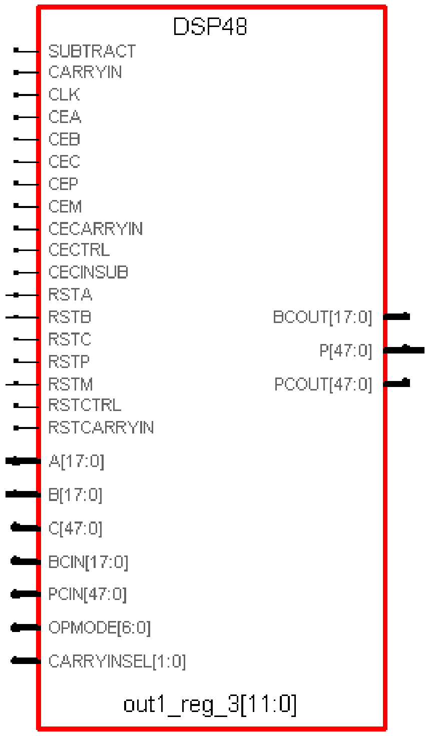
Using Virtex4 DSP48 Components with the Synplify Pro® Software

Synplicity®

Simply Better Results

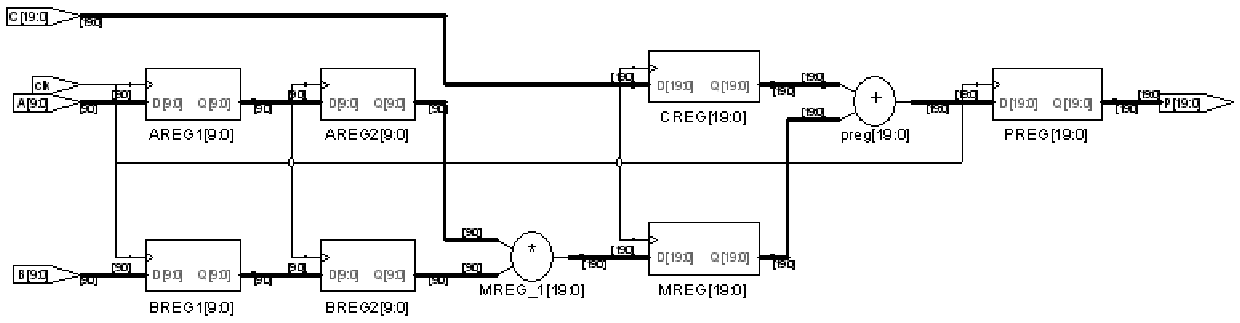
The Virtex4 FPGA architecture from Xilinx includes a new DSP-oriented component called the DSP48. This is a dedicated component in the Virtex4 architecture that is specifically engineered to help with DSP-related operations. This application note gives you a general understanding of the DSP48 component, and shows you how to infer it with the Synplify Pro synthesis software. For a complete list of all the DSP48 options and the way in which they are arranged in the Virtex4 FPGA, refer to the *XtremeDSP Design Considerations User Guide* or the *Virtex4 Library Guide*, both available from www.xilinx.com.

The following figure shows the DSP48 component:

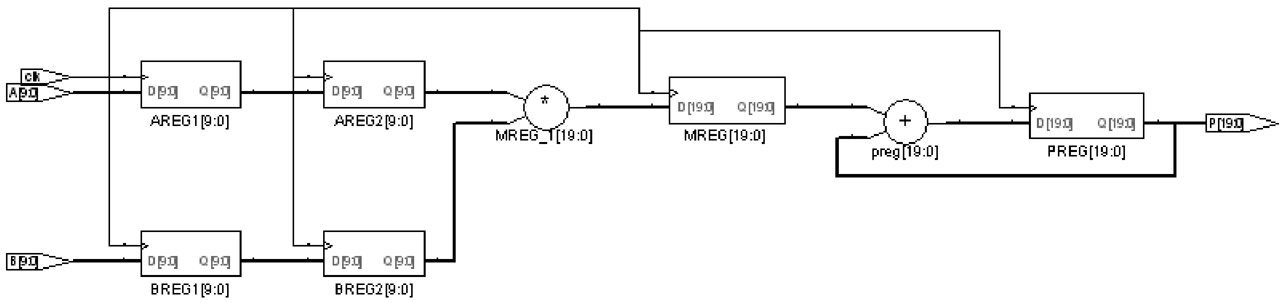


What is the DSP48?

At its most basic, the DSP48 is a multiplier that is followed by an adder with several optional registers on the ports and between the multiplier and adder. The following figure shows a basic representation of the DSP48 component.



The multiplier takes two 18-bit signed signals and multiplies them into a 36-bit result. This is then sign extended to 48 bits and can be fed into the adder or routed directly to the outputs of the DSP48. The adder, which can be configured either as an adder or subtractor, can accept the sign-extended output of the multiplier and 48-bit C input to the DSP48. In addition, the adder can also accept itself as an input, to form an accumulator.



Along with the multiplier and adder are several registers. All these registers are optional and their usage is controlled by attributes set with the DSP48 when it is inferred. The A and B inputs that go to the multiplier can have up to two registers, known as the AREG and BREG registers. The C input that goes to one of the operands of the adder can have up to one register, which is known as the CREG. There can be up to one register between the multiplier and adder; this is called MREG. Finally, there can be up to one register after the adder, and this is called the PREG.

Each of these registers has enables and resets, which can be different from each other. The only exception to this is if there are two AREG or two BREG registers, they must share the same enable and reset. AREG and BREG can have different enables, but if there are two AREGs for example, both of them must have the same enable and reset. Also note that all the resets are synchronous. Finally, there is only one clock pin on the DSP48, so all of the registers must use the same clock.

The Synplify Pro tool recognizes the architecture of these registers and uses that knowledge to infer the DSP48 block. If you want registers to go inside the DSP48 block, it is very important that your Verilog or VHDL register description match what can go into a DSP48. If you describe asynchronous resets for example, the DSP48 will not include that register inside it.

One more basic control of the DSP48 is the OPMODE input to the DSP48. This is a 7-bit input to the DSP48 that controls the functionality of the DSP. Different combinations of these 7 bits control the different muxes inside the DSP48 and thus control how the DSP48 works. The synthesis tool automatically sets this when it infers the DSP48 components. For a complete list of all the combinations of the OPMODE input, please consult the DSP48 component description in the *Virtex4 Library Guide* available at www.xilinx.com.

Inferring DSP48 Components

When the Synplify Pro tool detects structures that can be mapped into a DSP48, it writes out a DSP48 in the EDIF netlist.The following sections show examples of structures that the synthesis tool implemented in a DSP48. The first section will show general structures, and the second section will show coding styles that will use specific parts of the DSP48 architecture.

The syn_dspstyle Attribute

For more control over the inference of the DSP48 component, use the new syn_dspstyle attribute. You can apply the attribute to operators (adders and multipliers), registers, and modules/architectures. This attribute can take 2 values: logic or dsp48. When it is set to logic, the software does not infer DSP48s; if it is set to dsp48, the tool infers DSP48s.

In the Synplify Pro 8.0 software, adders and counters are not automatically put in the DSP48 components, to ensure that the DSP48 components are only used for the most important structures in the design. In subsequent versions of the tool, more structures will automatically go into the DSP48. In order to get adders and counters into DSP48s for Synplify Pro 8.0, you must use the syn_dspstyle attribute.

VHDL	`attribute syn_dspstyle : string;`
	`attribute syn_dspstyle of adder_sig : signal is “dsp48”;`

Verilog	`wire [9:0] adder_sig /* synthesis syn_dspstyle = “dsp48” */;`

You can also use the syn_dspstyle attribute to map structures to logic that would by default go into the DSP48. For example, a multiplier name mult_sig, that drives a register named mult_reg would go into the DSP48. If you want to leave the mult in the DSP48 and take out the register, you must add the syn_dspstyle attribute to the register:

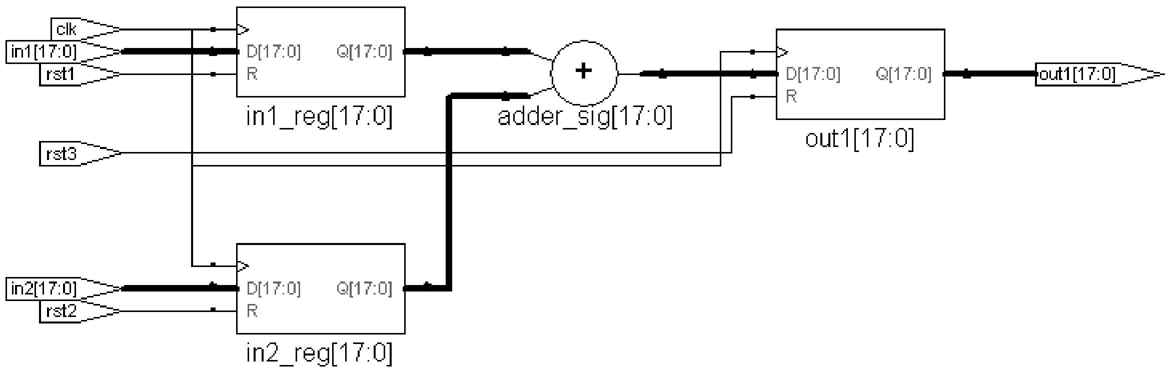
VHDL	`attribute syn_dspstyle : string;`
	`attribute syn_dspstyle of mult_reg : signal is “logic”;`

Verilog	`wire [19:0] mult_reg /* synthesis syn_dspstyle = “logic” */;`

Example 1: Adder

This is an 18x18 signed adder.The inputs and output each have a register with a different synchronous reset signal. They all fit into one DSP48 block.

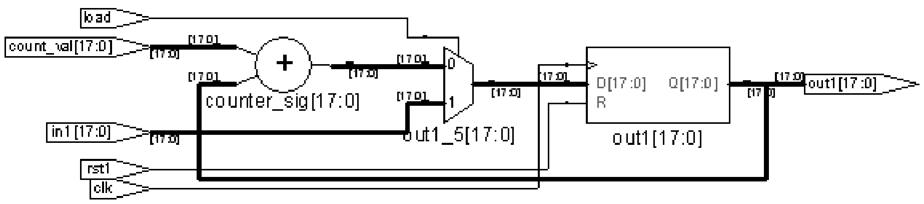
```
module adder_example(in1, in2, clk, rst1,rst2, rst3, out1);
input signed [17:0] in1, in2;
input rst1,rst2,rst3;
input clk;
output signed [17:0] out1;
reg signed [17:0] out1;
reg signed [17:0] in1_reg, in2_reg;
wire signed [17:0] adder_sig /* synthesis syn_dspstyle = "dsp48" */;

always@(posedge clk)
if (rst1)
   in1_reg <= 18'b0;
else
   in1_reg <= in1;

always@(posedge clk)
if (rst2)
   in2_reg <= 18'b0;
else
   in2_reg <= in2;

always@(posedge clk)
if (rst3)
   out1 <= 18'b0;
else
   out1 <= adder_sig;

assign adder_sig = in1_reg + in2_reg;
endmodule
```



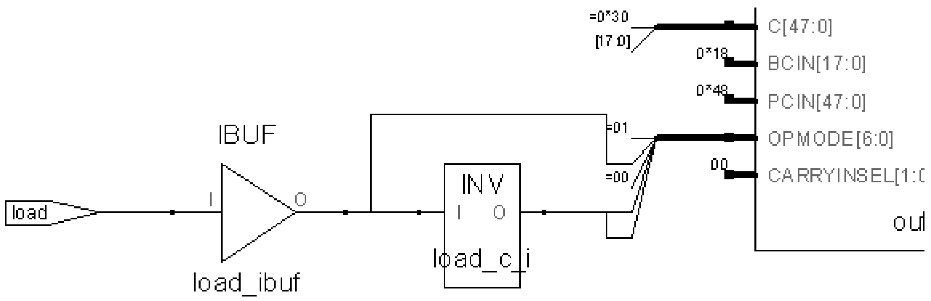
Example 2: Counter

This design is a loadable counter with a synchronous reset and a variable count increment. Like the adder, all of this fits in one DSP48.

```
module loadable_counter (in1,rst1, load,count_val, clk, out1);
input signed [17:0] in1, count_val;
input load;
input rst1;
input clk;
output signed [17:0] out1;
reg signed [17:0] out1;

wire signed [17:0] counter_sig /* synthesis syn_dspstyle = "dsp48" */;

always@(posedge clk)
   if (rst1)
      out1 <= 18'b0;
   else if (load)
      out1 <= in1;
   else
      out1 <= counter_sig;

assign counter_sig = out1 + count_val;

endmodule
```



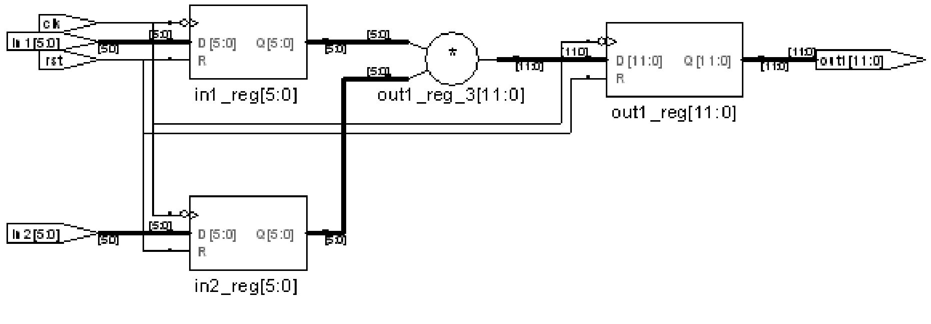
One other interesting thing to note about this example is the OPMODE reg input to the DSP48. In the figure below, notice that it is not static. Depending on the load signal, this design can either be a counter, or can pass the in1 value straight through. This load input can then control OPMODE to tell the DSP48 how it should be behaving. In the following figure, note that the INV component is absorbed into the DSP48 by the Xilinx Place and Route software. This means that you do not need to use a LUT for that path.



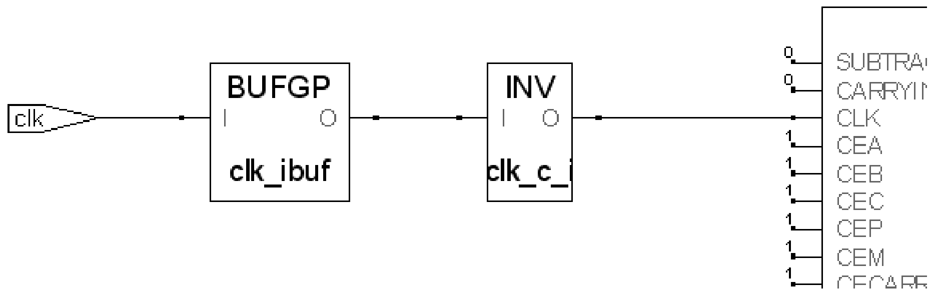
Example 3: Multiplier

This design is a simple 6x6 multiplier that has registers with the same reset on the inputs and the output. One difference here is that the clocks are all active low.

```
module mult_example(in1, in2, clk, rst, out1);
input [5:0] in1, in2;
input clk;
input rst;
output [11:0] out1;

wire [11:0] out1;

reg [5:0] in1_reg, in2_reg;
reg [11:0] out1_reg;

always@(negedge clk)
begin
   if (rst)
      begin
            in1_reg <= 6'b0;
            in2_reg <= 6'b0;
            out1_reg <= 12'b0;
      end
   else
      begin
            in1_reg <= in1;
            in2_reg <= in2;
            out1_reg <= in1_reg * in2_reg;
      end
end

assign out1 = out1_reg;

endmodule
```



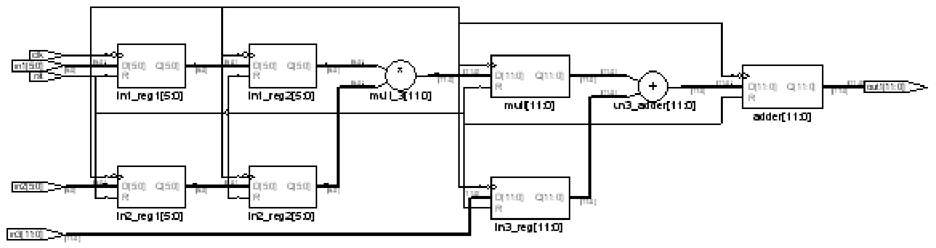
The DSP48 does not have an inverted clock port, so the Synplify Pro tool inverts the clock before the CLK input. The Xilinx Place and Route software absorbs that INV component into the DSP48, so that you do not need a LUT for that path. The following figure illustrates:



Example 4: Mult_add

A mult_add takes the multiplier and makes it drive an adder. In the following example, we are going to make use of all the registers that are in the DSP48. This whole design will fit into one DSP48.

```
module mult_add(in1, in2, in3, clk, rst, out1);
input [5:0] in1, in2;
input [11:0] in3;
input clk;
input rst;
output [11:0] out1;

wire [11:0] out1;

reg [5:0] in1_reg1, in1_reg2, in2_reg1, in2_reg2;
reg [11:0] in3_reg;
reg [11:0] mult;
reg [11:0] adder;

always@(negedge clk)
begin
   if (rst)
      begin
            in1_reg1 <= 6'b0;
            in2_reg1 <= 6'b0;
            in1_reg2 <= 6'b0;
            in2_reg2 <= 6'b0;
            in3_reg <= 12'b0;
            mult <= 12'b0;
            adder <= 12'b0;
      end
   else
       begin
            in1_reg1 <= in1;
            in2_reg1 <= in2;
            in1_reg2 <= in1_reg1;
            in2_reg2 <= in2_reg1;
            in3_reg <= in3;
            mult <= in1_reg2 * in2_reg2;
            adder <= mult + in3_reg;
      end
   end

assign out1 = adder;

endmodule
```



Example 5: Mult_subtract

There are two approaches to implementing a multiply and subtract:

- Take the result of the multiplier and subtract a value from that.
- Take a value, and subtract the result of multiplier from that.

The approach you use is important in DSP48 architecture because there is a subtract input that performs a subtraction when it is active. This input uses the equation below. If you code the VHDL or Verilog to use the same equation, the Synplify Pro tool can map directly to the DSP48:

```
   P <= C (input) - mult
```

However if you code your Verilog or VHDL code using the equation below, the Synplify Pro tool will have to add some extra logic to get it to work.

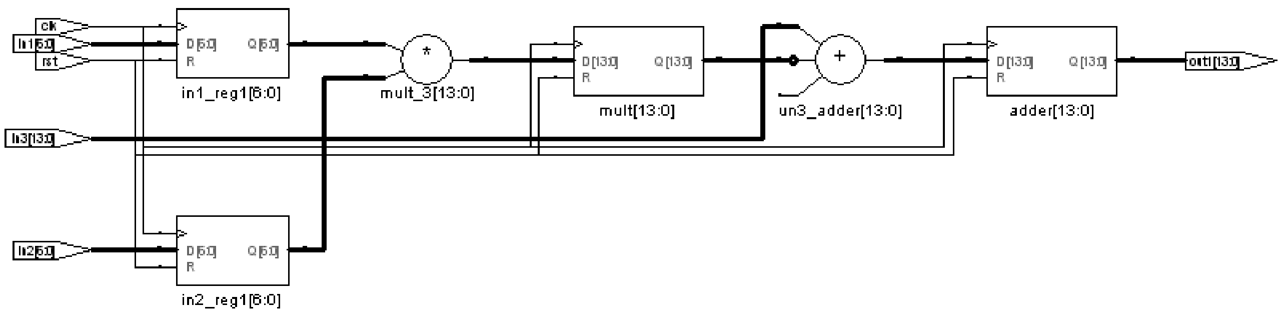
```
   P <= mult - C (input)
```

The following sections show examples of the two approaches.

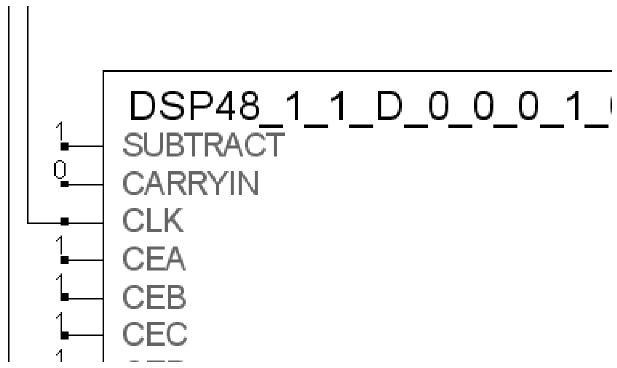
Mult_sub (Example 1)

```
module mult_sub(in1, in2, in3, clk, rst, out1);
input [6:0] in1, in2;
input [13:0] in3;
input clk;
input rst;
output [13:0] out1;

wire [13:0] out1;

reg [6:0] in1_reg1, in2_reg1;
reg [13:0] mult;
reg [13:0] subtractor;
```

```
always@(posedge clk)
begin
   if (rst)
      begin
            in1_reg1 <= 7'b0;
            in2_reg1 <= 7'b0;
            mult <= 14'b0;
            subtractor <= 14'b0;
      end
   else
      begin
            in1_reg1 <= in1;
            in2_reg1 <= in2;
            mult <= in1_reg1 * in2_reg1;
            subtractor <= in3 - mult;
      end
end

assign out1 = subtractor;

endmodule
```

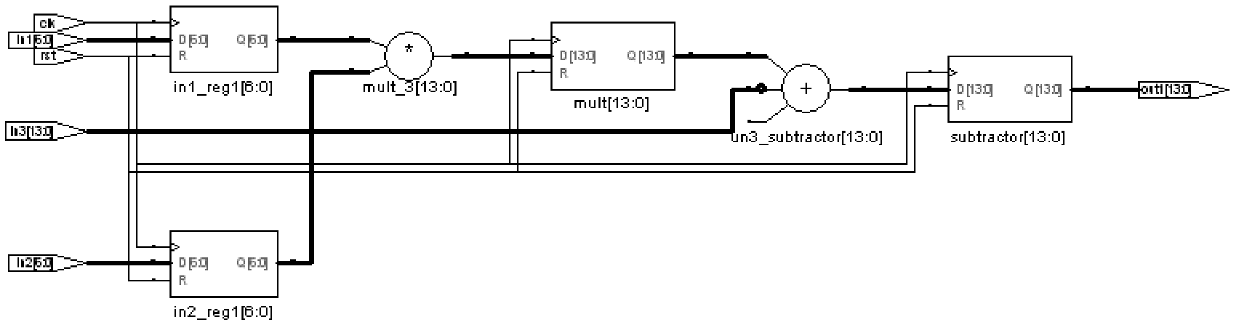


In the following example, since the equation is set as in3 – mult, this will fit directly into the DSP48. So all you have to do is to set the subtract input to 1.

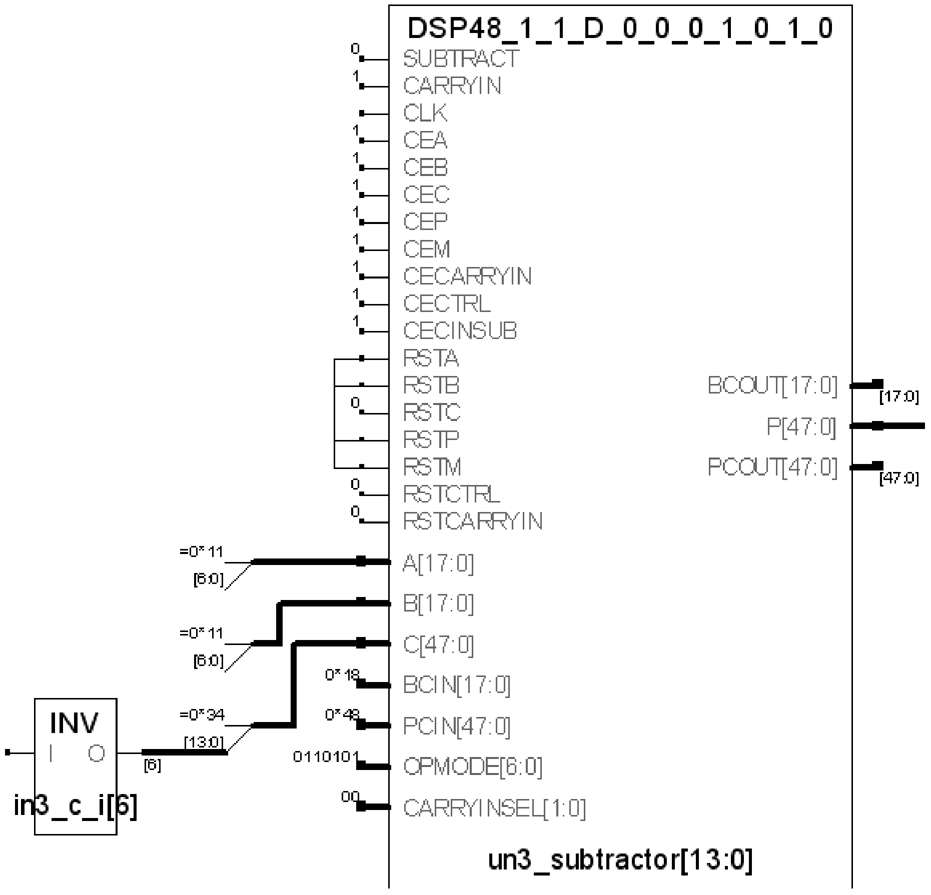


Mult_sub (Example 2)

In this case, instead of Q <= Cin – mult, it is written as Q <= mult –c. Since this is not possible as is in the DSP48, something else needs to be done. In this case, the Synplify Pro tool inverts the Cin, adds that to multiply, and adds 1 to it. In other words, it performs a two's complement addition that is the same as a subtraction.



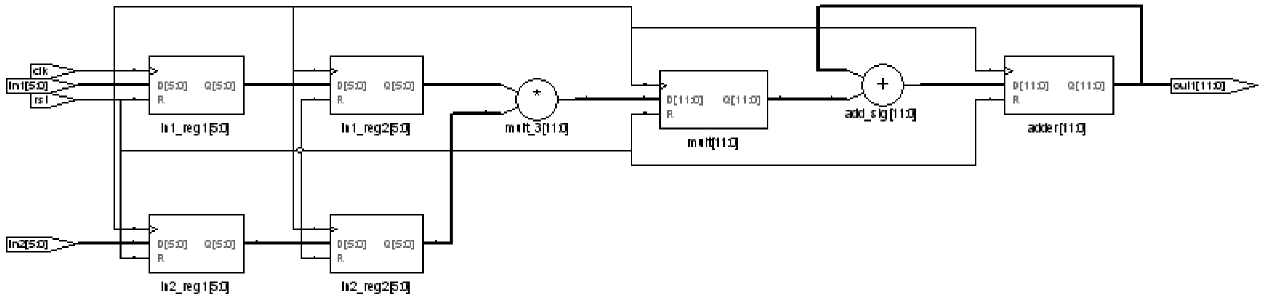
Since there is no inversion on the data inputs of the DSP48, the Synplify Pro tool performs it outside the DSP48.



Example 6: Mult-accumulate

A multiply accumulate is just like an adder, with the exception that the adder's operands are the output of the multiplier and the output of the adder itself.

```
module mult_accum(in1, in2, clk, rst, out1);
input [5:0] in1, in2;
input clk;
input rst;
output [11:0] out1;
wire [11:0] out1;

reg [5:0] in1_reg1, in1_reg2, in2_reg1, in2_reg2;
reg [11:0] mult;
reg [11:0] adder;
wire [11:0] add_sig;

always@(posedge clk)
begin
   if (rst)
      begin
            in1_reg1 <= 6'b0;
            in2_reg1 <= 6'b0;
            in1_reg2 <= 6'b0;
            in2_reg2 <= 6'b0;
            mult <= 12'b0;
            adder <= 12'b0;
      end
   else
      begin
            in1_reg1 <= in1;
            in2_reg1 <= in2;
            in1_reg2 <= in1_reg1;
            in2_reg2 <= in2_reg1;
            mult <= in1_reg2 * in2_reg2;
            adder <= add_sig;
      end
end

assign add_sig = mult + adder;
assign out1 = adder;
endmodule
```



This whole design fits inside one DSP48 component.

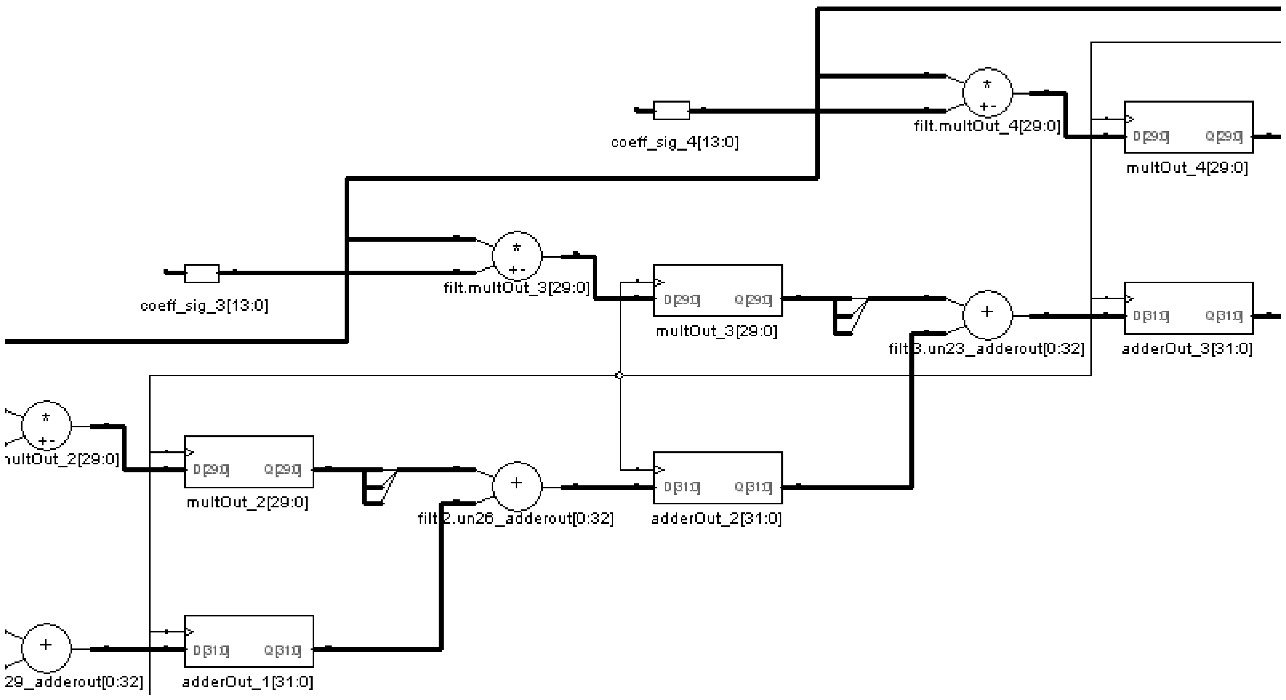
Example 7: Filter

Filters are more complicated structures, but they are built out of the components that have been implemented above. The main thing to remember when designing filters is that you must keep them as mult_adds (multiplier followed by an adder) if you want the Synplify Pro tool to map them to DSP48s. This is because mult_adds match the structure of a DSP48. When the Synplify Pro tool detects mult adds in the design, it automatically places them in the DSP48 architecture. This is a sample filter:

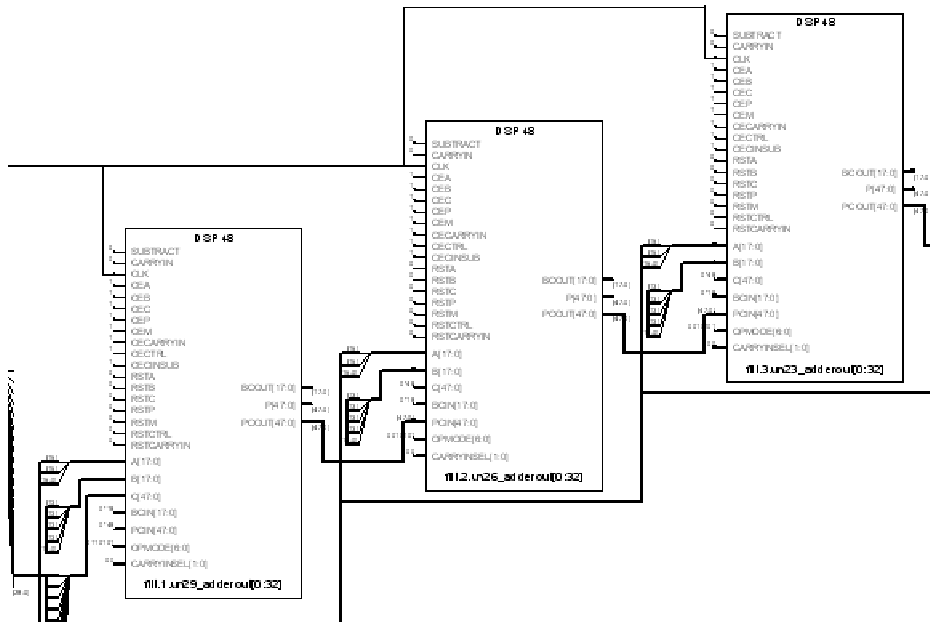
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity filter_new is
generic(
   tap_length: integer := 8;
   width: integer := 16;
   coeff_width: integer := 14;
   out_width: integer := 32
);
port (
   clk:  in std_logic;
   inp:  in signed(width-1 downto 0);
   outp: out signed(out_width-1 downto 0)
);
end filter_new;

architecture behav of filter_new is
type bus16 is array (0 to tap_length-1) of signed(width-1 downto 0);
type bus32 is array (0 to tap_length-1) of signed(out_width-1 downto 0);
type bus14 is array (0 to tap_length-1) of signed(coeff_width-1 downto 0);

signal in_flop  :  signed(width-1 downto 0);
signal in_flop_a :  signed(width-1 downto 0);
signal out_flop : signed(out_width-1 downto 0);
signal adderOut  : bus32;
attribute syn_preserve : boolean;
attribute syn_preserve of adderOut : signal is true;
signal multOut : bus32;
constant coeff: bus14 := (to_signed(-3,coeff_width),
   to_signed(-10,coeff_width),
   to_signed(-20,coeff_width),
   to_signed(-30,coeff_width),
   to_signed(30,coeff_width),
   to_signed(20,coeff_width),
   to_signed(10,coeff_width),
   to_signed(3,coeff_width));
signal coeff_sig : bus14;
attribute syn_keep : boolean;
attribute syn_keep of coeff_sig : signal is true;
```

```
begin

coeff_sig <= coeff;

filt: process(clk)
begin
   if(rising_edge(clk)) then
      in_flop_a <= inp;
      in_flop <= in_flop_a;
      multOut(0) <= resize(in_flop * coeff_sig(0), out_width);
      adderOut(0) <= multOut(0);
      for i in tap_length-1 downto 1 loop
            multOut(i)  <= resize(in_flop * coeff_sig(i), out_width);
            adderOut(i) <= adderOut(i-1) + multOut(i);
      end loop;
      out_flop <= adderOut(tap_length-1);
   end if;
end process;
outp <= out_flop;
end behav;
```

Notice that this is in the form of registers, multipliers and adders.



The figure below shows how the Synplify Pro software implemented the DSP48s.



Using syn_preserve to Infer Mult-accumulates

Notice that these DSP48s use the PCOUT/PCIN pins in order to use the chain features of the DSP48. In order to make use of the PCOUT/PCIN pins, the Synplify Pro software automatically sign extends the size of the adder so that it uses all 48 bits. One other thing to note about this design, is that if the adders are not the same size, the Synplify Pro 8.0 tool does not support the use of PCOUT/PCIN pins. In this example, the code says the following:

```
for i in tap_length-1 downto 1 loop
      multOut(i)  <= resize(in_flop * coeff_sig(i), out_width);
      adderOut(i) <= adderOut(i-1) + multOut(i);
end loop;
```

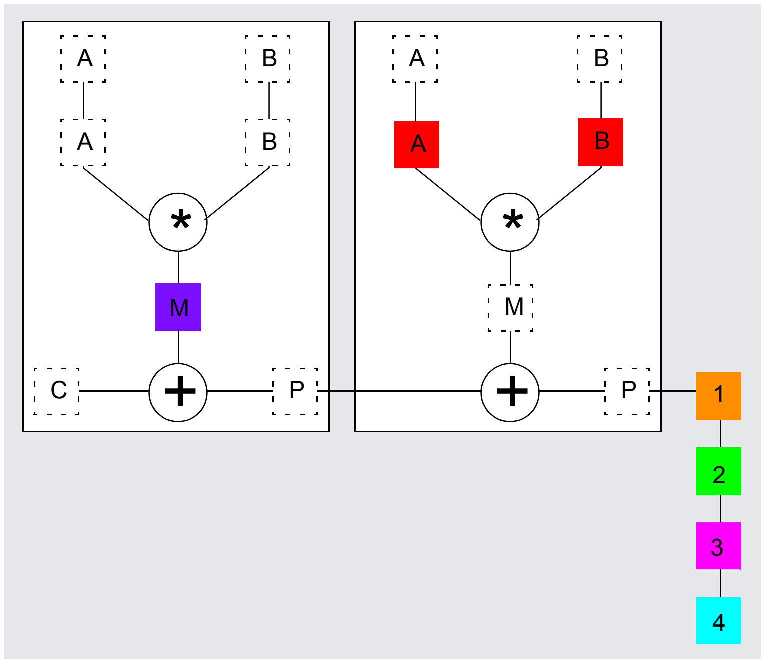
This means that even though the adderOut registers are all declared as the same size (8 x 16 array), the Synplify Pro tool prunes out the unused ones because not all the array elements are used. This causes the DSP48 not to be implemented. In the example above, this problem was fixed by using the syn_preserve attribute to keep the unused registers. In the Synplify Pro 8.1 release, this will be done automatically, so you will not need the syn_preserve attribute.

Using syn_keep to Infer Mult-accumulates

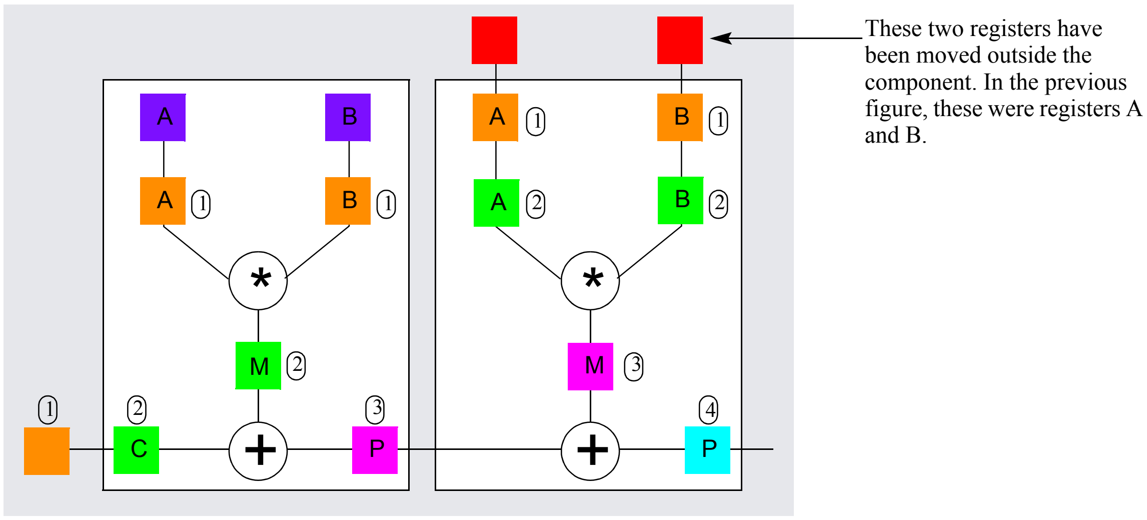
In filters, one of the operands of the multiplier is usually a constant. Historically, when it found a multiplier with a constant on one operand, the Synplify Pro tool propagated the constant on that multiplier to optimize the design for area, and removed unneeded logic. In the case of Virtex4 with DSP48s, the extra logic is not a problem because it all goes into the DSP48. However, the Synplify Pro 8.0 tool still propagates the constant, and this prevents the multiplier with a constant operand from being inferred as a DSP48. This will be fixed in Synplify Pro 8.1 software. In the mean time, the way to fix this is to make a signal that gets the constant, and then place a syn_keep on that signal, as shown in the previous code.

Pipelining Registers to Maximize Use of DSP48s

In addition to inferring many types of structures for DSP48 components, the Synplify Pro tool also pipelines registers around logic in order to more effectively utilize the registers inside the DSP48. It does this by taking registers that are outside the mult_adds, and moving them inside the mult_adds so that they match the DSP48 structure. The following figure shows the first mult_add has an MREG, the second has one AREG and one BREG, but there are several registers after the second mult_add. If the registers are not moved, the first and second DSP48 would only use 1 and 2 registers respectively.



The Synplify Pro mapper uses pipelining to move the four registers into the DSP48s, so that the design looks like the following figure. The circled numbers represent the four registers that were previously outside the DSP48s (see previous figure) and show how they have been packed into the components. Now, the only registers that are left over are the two on the operands of the second mult_add and the one on the C input of the first mult_add.


A
B
A 1
B 1
*
M 2
1
2
3
C
+
P
A 1
B 1
A 2
B 2
*
M 3
4
+
P
These two registers have been moved outside the component. In the previous figure, these were registers A and B.

Conclusion

The Xilinx mapper within the Synplify Pro tool has been specifically optimized to make use of the new DSP48 component within the Virtex4 architecture. The DSP48 component is essential if you want to create fast Virtex4 designs that utilize DSP functions.

The examples provided in this application note are meant to show you the types of coding styles you can use to create the design you want. However, they are not meant to be the only way to code for this architecture. The most important thing to remember is if you want to infer and implement DSP48s, your code structure must resemble the DSP48 itself. If the source code has structures that are in the DSP48 (like multipliers, adders, counters, and mult_adds), and if DSP48 rules are followed (synchronous registers), the Synplify Pro tool can automatically implement the DSP48 structures you want.

Synplicity®

Simply Better Results

Synplicity, Inc.
600 W. California Ave, Sunnyvale, CA 94085 USA
Phone: (U.S.) +1 408 215-6000, Fax: (U.S.) +1 408 222-0268
www.synplicity.com