Inferring Microsemi SmartFusion2, IGLOO2 and RTG4 MACC Blocks

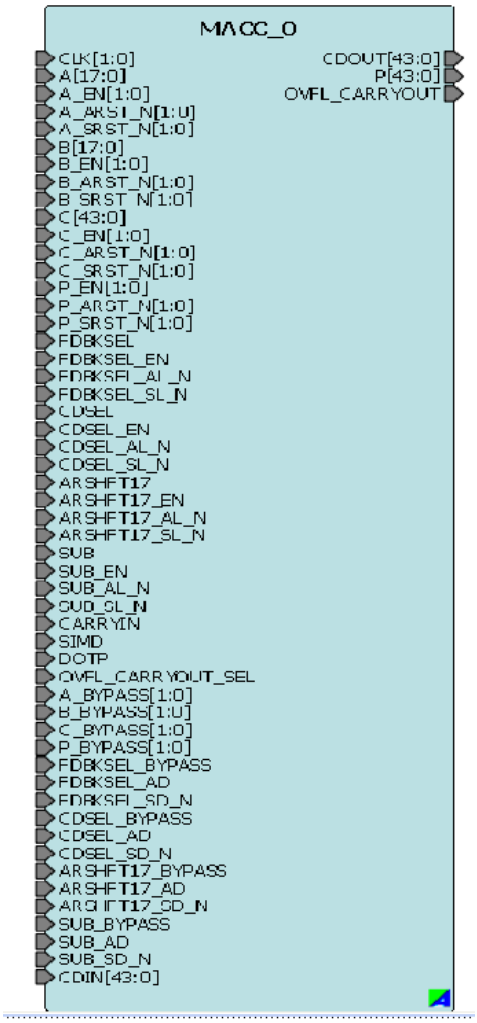
Synopsys® Application Note, May 2016

The Synopsys® Synplify Pro® synthesis tool automatically infers and implements Microsemi® SmartFusion2, IGLOO2, and RTG4 MACC blocks. The SmartFusion2, IGLOO2, and RTG4 architecture includes dedicated MACC block components, which are 18x18-bit signed multiply-accumulate blocks. The blocks can perform DSP-related operations like multiplication followed by addition, multiplication followed by subtraction, and multiplication with accumulate. This application note provides a general description of the Microsemi SmartFusion2, IGLOO2, and RTG4 MACC block component and shows you how to infer and implement it with the Synplify Pro software.

The following topics describe the details:

The SmartFusion2, IGLOO2, and RTG4 MACC Blocks

The SmartFusion2, IGLOO2, and RTG4 device supports 18x18-bit signed multiply-accumulate MACC blocks. The multiplier takes two 18-bit signed signals and multiplies them for a 36-bit result. The result is then extended to 44 bits. In addition to multiplication followed by addition or subtraction, the blocks can also accumulate the current multiplication product with a previous result, a constant, a dynamic value, or a result from another MACC block. The following figure shows the 18x18-bit MACC block.



All signals of the MACC block, except CDIN and CDOUT, have optional registers. All registers must use the same clock. Each of the registers has enables and resets that can differ from each other. For a complete list of all the block options and their configurations, refer to the Microsemi documentation.

Inferring MACC Blocks for SmartFusion2, IGLOO2, and RTG4

Starting with the F-2011.09M-SP1 version of the Synplify Pro tool, you can now infer MACC block components. You can write your RTL so that the synthesis tool recognizes the structures and maps them to MACC components. The Synplify Pro tool extracts the following logic structures from the hardware description and maps them to MACC blocks: mults (Multiplier), multAdds (multiplier followed by an adder), multSubs (multiplier followed by a subtractor), and multAccs (multiplier-accumulator structures)

The Synplify Pro tool supports the inference of both signed and unsigned multipliers. There are some design criteria that influence inference:

- The Microsemi MACC blocks support multipliers up to a maximum of 18x18 bits for signed multipliers and 17x17 bits for unsigned multipliers. The synthesis tool splits multipliers that exceed these limits between multiple MACC blocks, as described in *Inferring MACC Blocks for Wide Multipliers*, on page 21.
- The Synplify Pro synthesis tool supports the inference of multiple MACC block components across different hierarchies. The multipliers, input registers, output registers, and subtractors/adders are packed into the same MACC block, even if they are in different hierarchies.
- The synthesis tool packs registers at the inputs and outputs of mults, multAdds, multSubs, and multAccs into MACC blocks.

  By default, the tool maps all multiplier inputs with a width of 3 or greater to MACC blocks. If the input width is smaller, it is mapped to logic. You can change this default behavior with the syn_multstyle attribute (see *Controlling Inference with the syn_multstyle Attribute*, on page 4).
- The tool packs registers at inputs and outputs of mults, multAdds, multSubs, and multAccs into MACC blocks, as long as all the registers use the same clock.

  – If the registers have different clocks, the clock that drives the output register gets priority, and all registers driven by that clock are packed into the block.

  – If the outputs are unregistered and the inputs are registered with different clocks, the input registers with input that has a larger width get priority and are packed in the MACC blocks.
- The synthesis tool supports register packing across different hierarchies for multipliers up to a maximum of 18x18 bits for signed multipliers and 17x17 bits for unsigned multipliers. The synthesis tool pipelines registers for multipliers that exceed these limits into multiple MACC blocks, as described in *Inferring MACC Blocks for Wide Multipliers*, on page 21.

- The synthesis tool packs different kinds of flip-flops at the inputs/outputs of the mults, multAdds, multSubs, and multAccs into MACC blocks:
  - D type flip-flop
  - D type flip-flop with asynchronous reset
  - D type flip-flop with enable
  - D type flip-flop with asynchronous reset and enable
  - D type flip-flop with synchronous reset
  - D type flip-flop with synchronous reset and enable
- The synthesis tool uses the MACC cascade feature with multi-input mult-adds and mult-subs, up to a maximum of 18x18 bits for signed multipliers and 17x17 bits for unsigned multipliers. The synthesis tool packs logic into MACC blocks efficiently using hard-wired cascade paths to improve the quality of results (QoR) for the design, as described in *Inferring MACCs for Multi-Input MultAdds/MultSubs* , on page 38.
- The synthesis tool uses the internal paths for adder feedback loops inside the MACC instead of connecting it externally for multAccs up to a maximum of 18x18 bits for signed multipliers and 17x17 bits for unsigned multipliers, as described in *Inferring MACC Blocks for Multiplier-Accumulators*, on page 51.
- The synthesis tool infers MACC block in DOTP mode as described in Inferring MACC block in DOTP mode, on page 62.

Controlling Inference with the syn_multstyle Attribute

Use the syn_multstyle attribute to control the inference of multiple MACC blocks. The attribute is briefly described here; for detailed information and more examples, refer to the *FPGA Synthesis Reference Manual*.

Controlling Default Inference

By default, multipliers with input widths of 3 or greater are packed in the MACC block, while smaller input widths are mapped to logic. If the multipliers are inferred as MACC blocks by default, you can use the syn_multstyle attribute to map the structures to logic:

VHDL
```
attribute syn_multstyle : string ;
attribute syn_multstyle of mult_sig : signal is "logic";
```

Verilog
```
wire [1:0] mult_sig /* synthesis syn_multstyle = "logic" */;
```

If the multipliers are mapped to logic by default, you can use the syn_multstyle attribute to override this and map the structures to MACC blocks, using the dsp value for the attribute:

```
VHDL      attribute syn_multstyle : string ;
          attribute syn_multstyle of mult_sig : signal is "dsp";
```

```
Verilog   wire [1:0] mult_sig /* synthesis syn_multstyle = "dsp" */;
```

Specifying the Scope of the Attribute

You can apply the attribute globally or to individual modules, as the following sdc syntax examples illustrate:

```
define_global_attribute syn_multstyle {dsp|logic}

define_attribute {object} syn_multstyle {dsp|logic}
```

Coding Style Examples

There are many ways to code your DSP structures, but the synthesis tool does not map all of them to MACC blocks. The following examples illustrate coding styles from which the synthesis tool can infer and implement MACC blocks. It is important that you use a supported coding structure so that the synthesis tool infers the MACC blocks.

Check the results of inference in the log file and the final netlist. The resource usage report in the synthesis log file (srr) shows details like the number of blocks. It also reports if they are configured as mult, multAdd, or multSub blocks. You should also check the final netlist to make sure that the structures you want were implemented.

See the following examples of recommended coding styles:

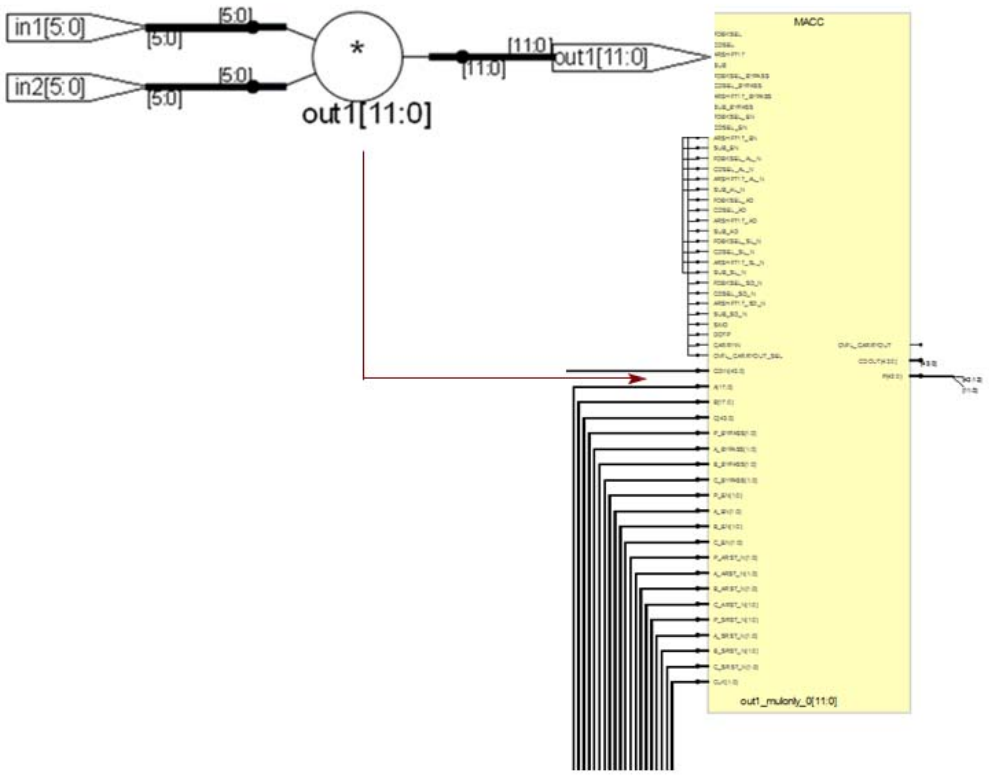
- Example 1: 6x6-Bit Unsigned Multiplier
- Example 2: 11x9-Bit Signed Multiplier
- Example 3: 18x18-Bit Signed Multiplier with Registered I/Os
- Example 4: 17x17-Bit Unsigned Multiplier with Different Resets
- Example 5: Unsigned Mult with Registered I/Os and Different Clocks
- Example 6: Multiplier-Adder
- Example 7: Multiplier-Subtractor

For other examples, see *Wide Multiplier Coding Examples*, on page 26.

Example 1: 6x6-Bit Unsigned Multiplier

The following design is a simple 6x6-bit unsigned multiplier, which the tool maps to MACC block, as shown in the subsequent figure.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity unsign_mult is
   port
      in1 : in std_logic_vector (5 downto 0);
      in2 : in std_logic_vector (5 downto 0);
      out1 : out std_logic_vector (11 downto 0)
);
end unsign_mult;


architecture behav of unsign_mult is begin
out1 <= in1 * in2;
end behav;
```



Resource Usage Report for Unsigned 6x6-Bit Multiplier

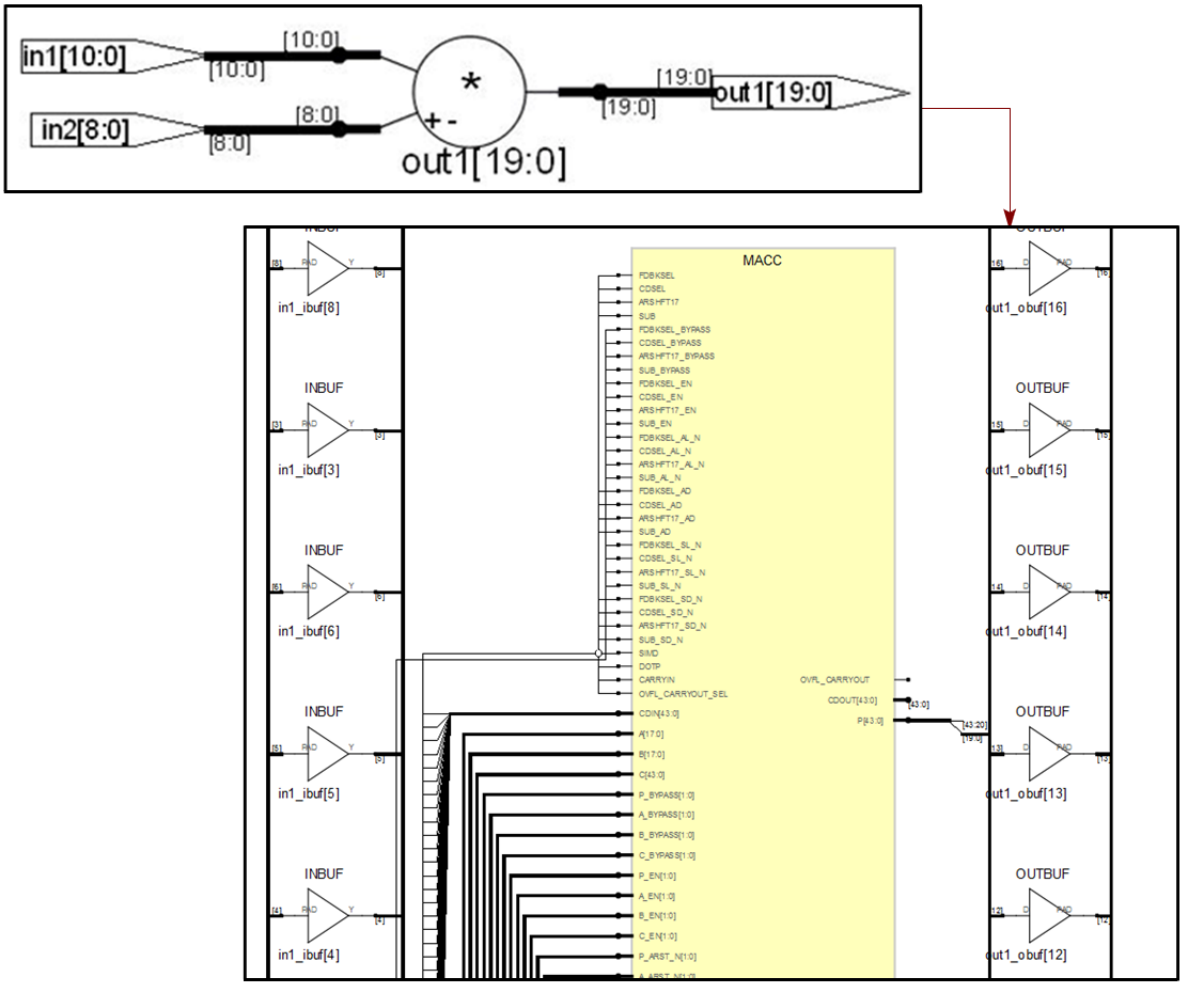
This section of the log file (`srr`) shows resource usage details. It shows that the multiplier code was implemented in one MACC multiplier block.

```
Mapping to part:

Sequential Cells:
SLE 0 uses

DSP Blocks:1
 MACC1 Mult

Total LUTs:0
```

Example 2: 11x9-Bit Signed Multiplier

This example is an 11x9-bit signed multiplier. It gets mapped into one MACC block, as shown in the figure below.

```
library IEEE;
use iEEE.std_logic_1164.all;
use iEEE.numeric_std.all;

entity sign_mult is
   port
      in1 : in signed (10 downto 0);
      in2 : in signed (8 downto 0);
      out1 : out signed (19 downto 0)
);
end sign_mult;

architecture behav of sign_mult is
begin
out1 <=in1 * in2 ;
end behav;
```



Resource Usage Report for 11x9-Bit Signed Multiplier

Mapping to part:

Sequential Cells:
SLE0 uses

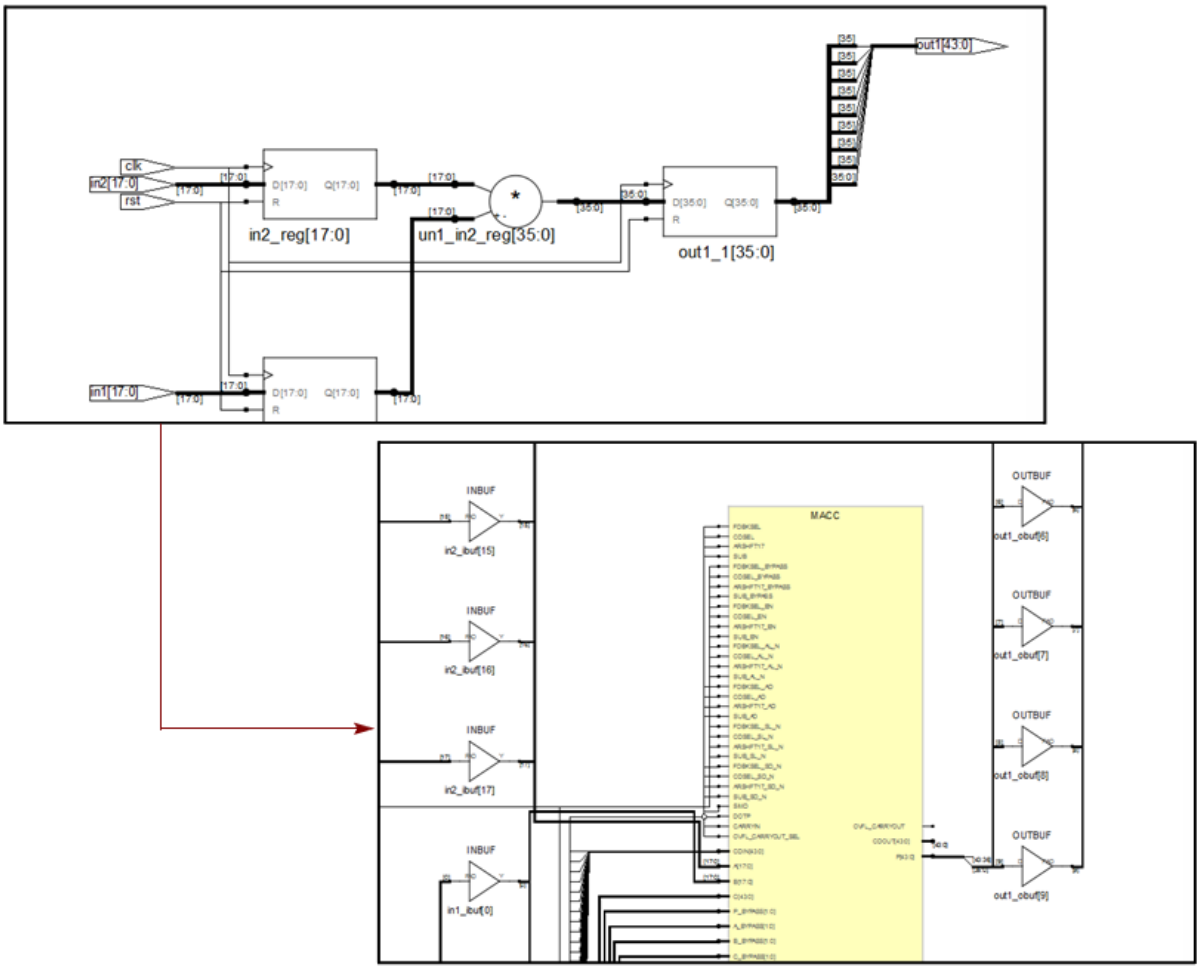
DSP Blocks:1
MACC Mult

Total LUTs:0

Example 3: 18x18-Bit Signed Multiplier with Registered I/Os

This is code for an 18x18 signed multiplier. The inputs and outputs are registered, with a synchronous active low reset signal. The synthesis tool fits all this logic into one MACC block, as shown below.

```
module sign18x18_mult ( in1, in2, clk, rst, out1 );

input signed [17:0] in1, in2;
input clk;
input rst;
output signed [40:0] out1;
reg signed [40:0] out1;
reg signed [17:0] in1_reg, in2_reg;

always @ ( posedge clk )
begin
   if ( ~rst )
      begin
         in1_reg <= 18'b0;
         in2_reg <= 18'b0;
         out1 <= 41'b0;
      end
      else
      begin
         in1_reg <= in1;
         in2_reg <= in2;
         out1 <= in1_reg * in2_reg;
      end
end

endmodule
```



Resource Usage Report for Signed I8xI8-Bit Multiplier with Registered I/Os

CFG11 use

Sequential Cells:
SLE0 uses

DSP Blocks 1
MACC 1 Mult

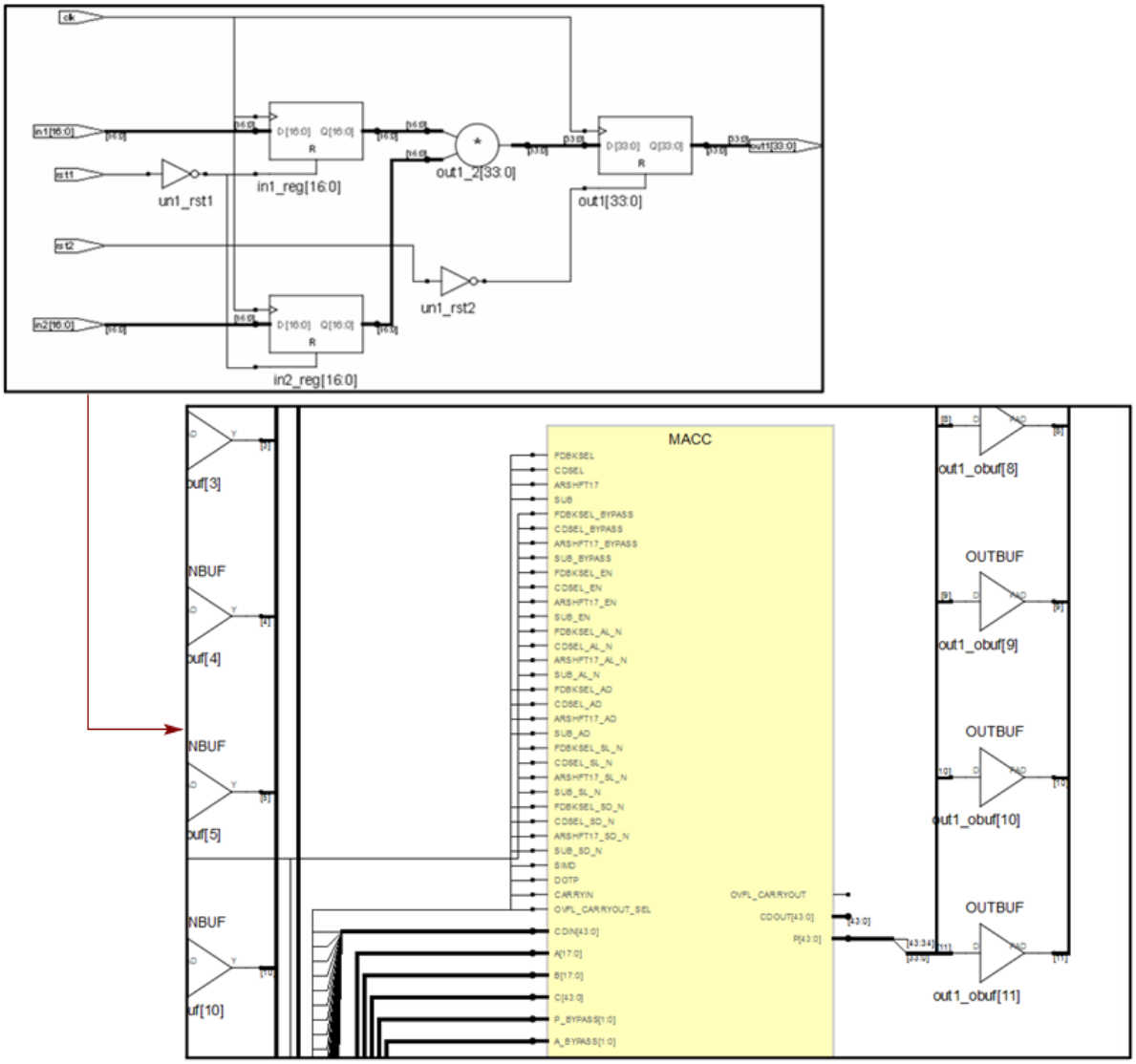
Global Clock Buffers: 1

Total LUTs: 1

Example 4: 17x17-Bit Unsigned Multiplier with Different Resets

This is a VHDL example of a 17x17-bit unsigned multiplier, which has input and output registers with different asynchronous resets. The tool packs all the logic into one MACC block as shown below.

```
library iEEE;
use iEEE.std_logic_1164.all;
use iEEE.std_logic_unsigned.all;

entity unsign17x17_mult is port
in1 : in std_logic_vector (16 downto 0); in2 : in std_logic_vector
(16 downto 0); clk : in std_logic;
rst1 : in std_logic;
rst2 : in std_logic;
out1 : out std_logic_vector (33 downto 0)
);
end unsign17x17_mult;

architecture behav of unsign17x17_mult is
signal in1_reg, in2_reg : std_logic_vector (16 downto 0 );
begin

process ( clk, rst1 )
begin
   if ( rst1 = '0') then
      in1_reg <= ( others => '0');
      in2_reg <= ( others => '0');
      elsif ( rising_edge(clk)) then
         in1_reg <= in1;
         in2_reg <= in2;
   end if;
end process;

process ( clk, rst2 )
begin
   if ( rst2 = '0') then
      out1 <= ( others => '0');
      elsif ( rising_edge(clk)) then
         out1 <= in1_reg * in2_reg;
   end if;
end process;
end behav;
```



Resource Usage Report for Unsigned 17x17-Bit Multiplier

SLE0 uses

DSP Blocks:1
MACC:1 Mult

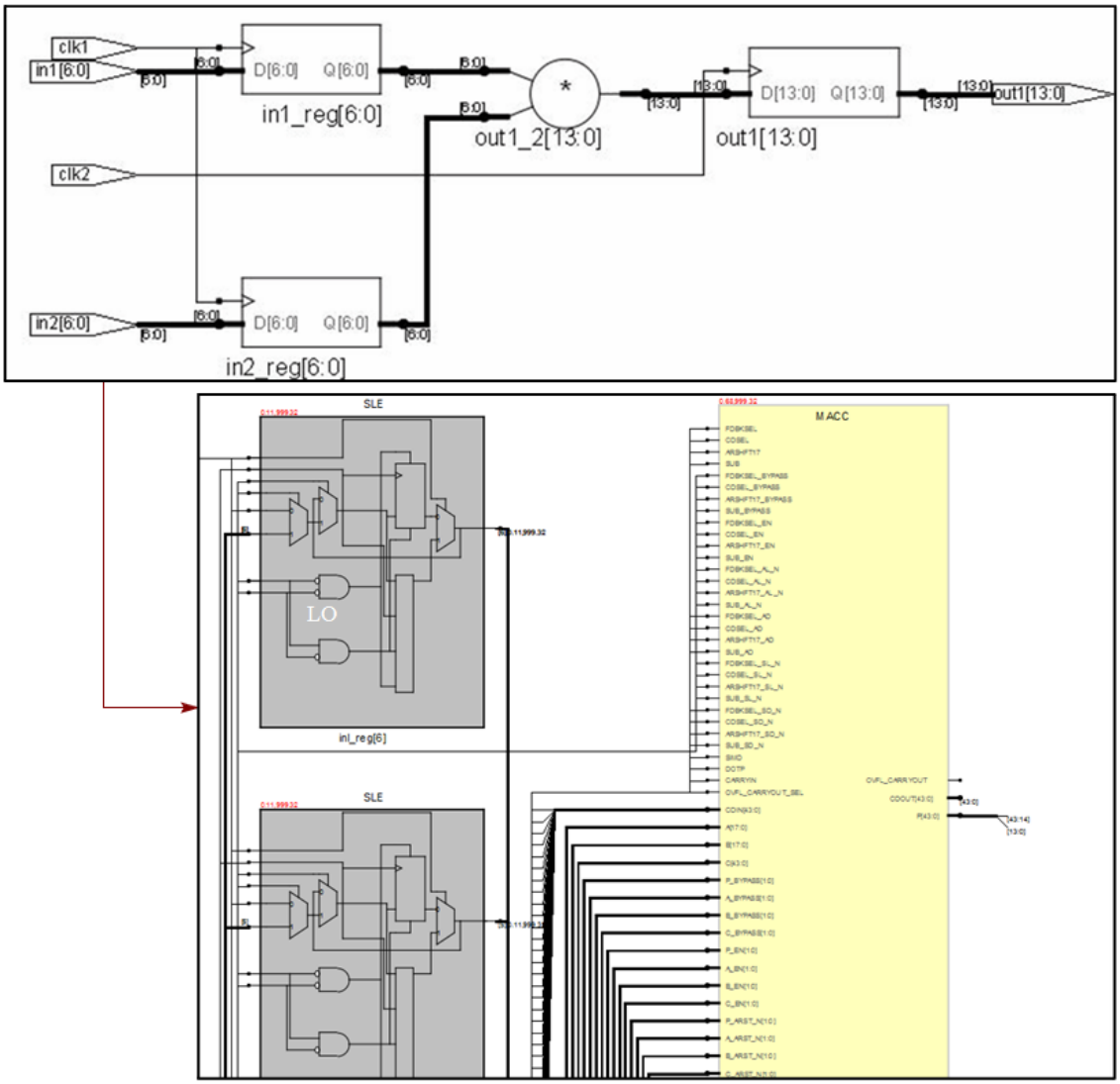
Global Clock Buffers: 1

Total LUTs:0

Example 5: Unsigned Mult with Registered I/Os and Different Clocks

This example shows an unsigned multiplier whose inputs and outputs are registered with different clocks, clk1 and clk2, respectively. In this design, the synthesis tool only packs the output registers and the multiplier into the MACC block. The input registers are implemented as logic outside the MACC block.

```
module unsign_mult ( in1, in2, clk1, clk2, out1 );
input [6:0] in1, in2;
input clk1,clk2;
output [13:0] out1;
reg [13:0] out1;
reg [6:0] in1_reg, in2_reg;

always @ ( posedge clk1 )
begin
   in1_reg <= in1;
   in2_reg <= in2;
end

always @ ( posedge clk2 )
begin
   out1 <= in1_reg * in2_reg;
end

endmodule
```



Resource Usage Report for Unsigned Multiplier with Different Clocks

The log file shows that all 14-input registers are implemented as logic, outside the MACC block.

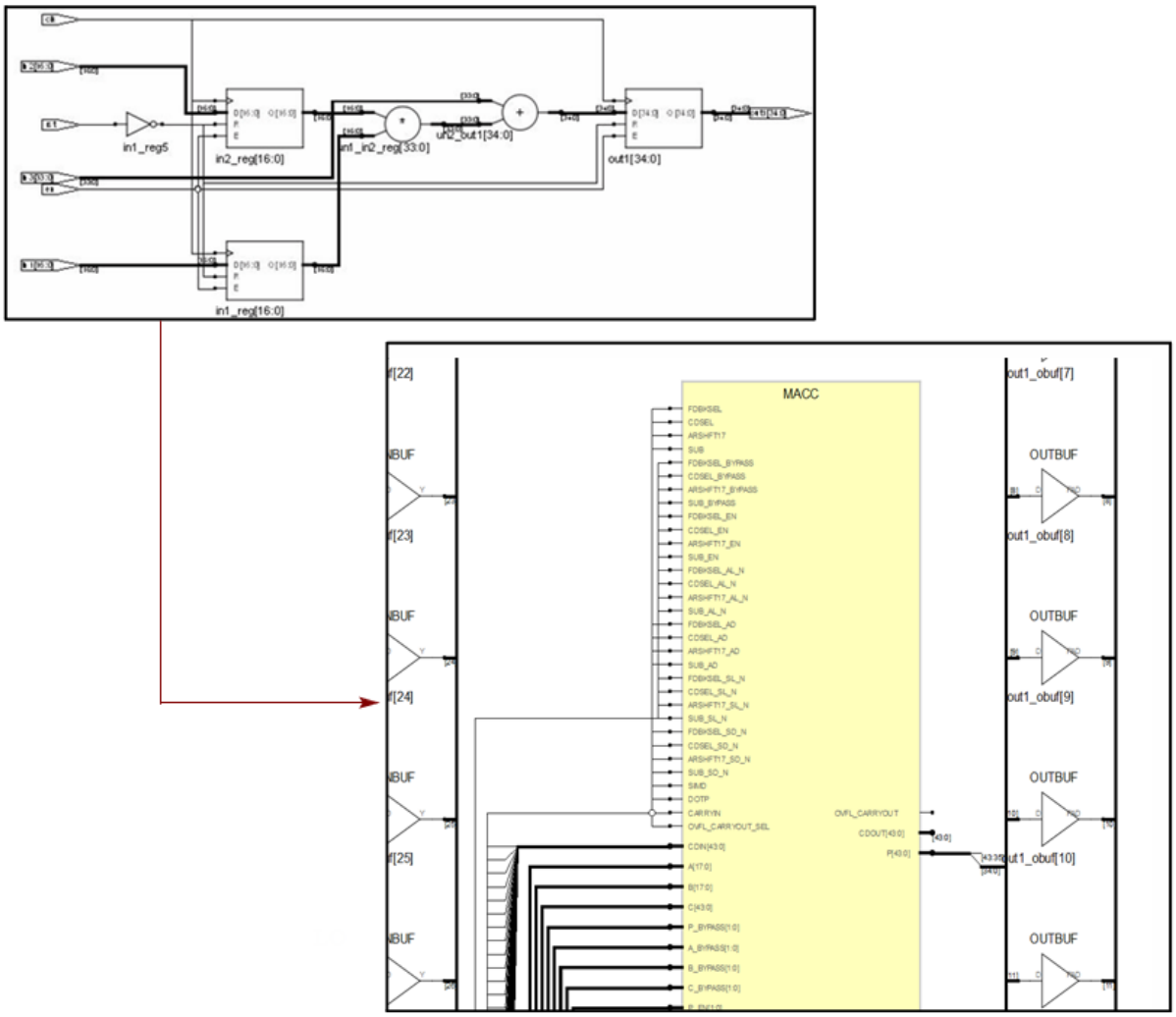
```
Mapping to part:
Cell usage:

Sequential Cells:
SLE14 uses

DSP Blocks:1
 MACC:1 Mult

Global Clock Buffers:2

Total LUTs:0
```

Example 6: Multiplier-Adder

This VHDL example shows a multiplier whose output is added with another input. The inputs and outputs are registered, and have enables and synchronous resets. The following figure shows how the design gets mapped into a MACC block.

```
library iEEE;
use iEEE.std_logic_1164.all;
use iEEE.std_logic_unsigned.all;

entity mult_add is port (
in1 : in std_logic_vector (16 downto 0);
in2 : in std_logic_vector (16 downto 0);
in3 : in std_logic_vector (33 downto 0);
clk : in std_logic;
rst : in std_logic;
en: in std_logic;
out1 : out std_logic_vector (34 downto 0)
);
end mult_add;
architecture behav of mult_add is
signal in1_reg, in2_reg : std_logic_vector (16 downto 0 );
signal mult_out : std_logic_vector ( 33 downto 0 );
begin

process ( clk )
begin
   if ( rising_edge(clk)) then
      if ( rst = '0' ) then
         in1_reg <= ( others => '0');
         in2_reg <= ( others => '0');
         out1 <= ( others => '0');
         elsif ( en = '1')then
            in1_reg <= in1;
            in2_reg <= in2;
            out1 <= ( '0' & mult_out ) + ('0' & in3 );
```

```
            end if;
        end if;
    end process;
    mult_out <= in1_reg * in2_reg;
    end behav;
```



Resource Usage Summary for Multiplier-Adder

CFG21 use

Sequential Cells:
SLE0 uses

DSP Blocks:1
MACC:1 MultAdd

Global Clock Buffers: 1

Total LUTs:

Example 7: Multiplier-Subtractor

There are two ways to implement multiplier and subtractor logic. The synthesis tool packs the logic differently, depending on how it is implemented.

- Subtract the result of multiplier from an input value (P = Cin - mult). The synthesis tool packs all logic into the MACC block.
- Subtract a value from the result of the multiplier (P = mult - Cin). The synthesis tool packs only the multiplier in the MACC block. The subtractor is implemented in logic outside the block.

See the following examples:

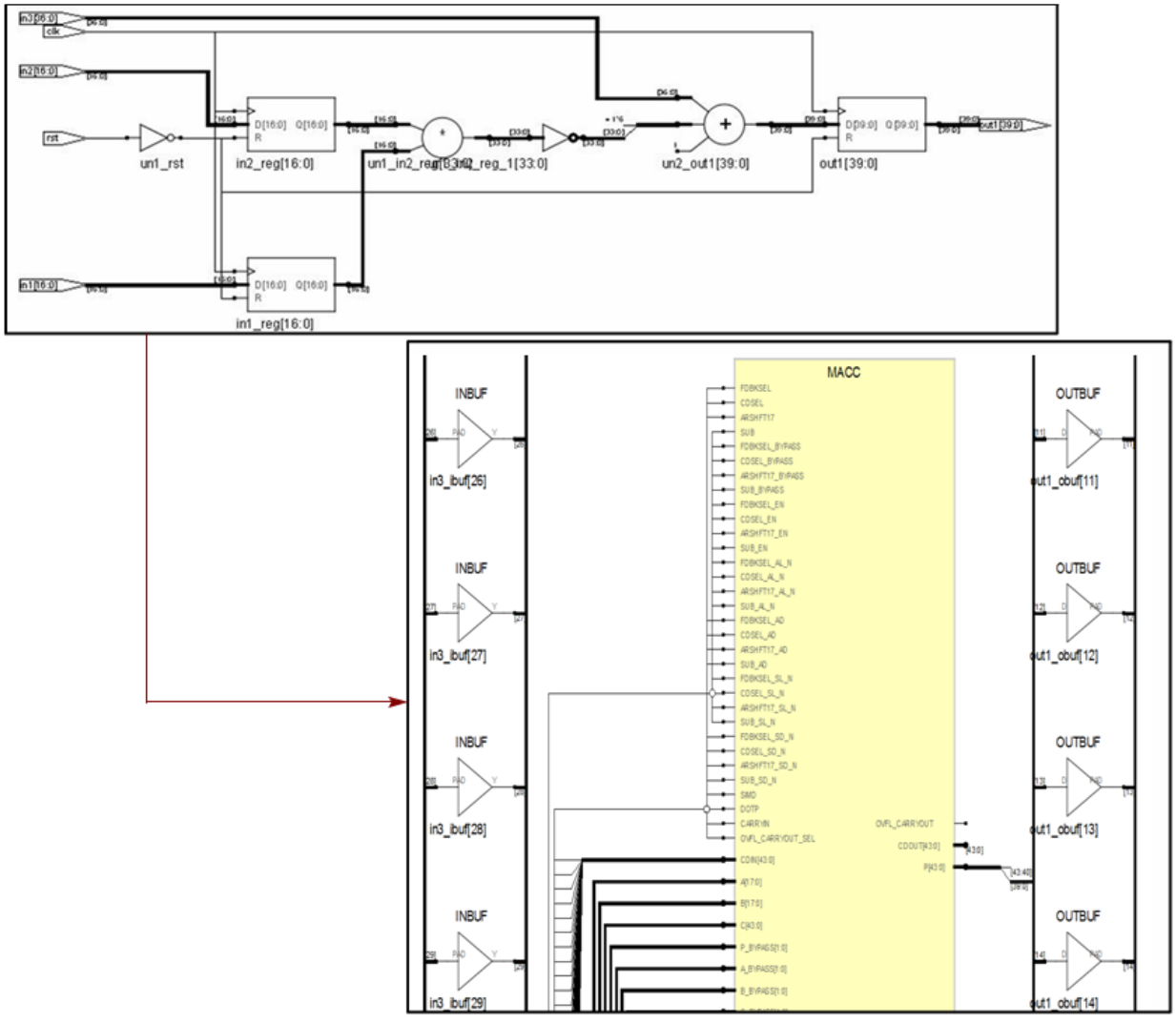
- Unsigned MultSub Verilog Example (P = Cin - Mult), on page 17
- Signed MultSub VHDL Example (P = Cin - Mult), on page 19
- Signed MultSub Verilog Example (P = Mult - Cin), on page 20
- Unsigned MultSub VHDL Example (P = Mult - Cin), on page 21

Unsigned MultSub Verilog Example (P = Cin - Mult)

The next figure shows how all logic for the example below is mapped into the MACC block.

```
module mult_sub ( in1, in2, in3, clk, rst, out1 );
input [16:0] in1, in2;
input [36:0] in3;
input clk;
input rst;
output [39:0] out1;
reg [39:0] out1;
reg [16:0] in1_reg, in2_reg;

always @ ( posedge clk )
begin
   if (~rst)
      begin
         in1_reg <= 17'b0;
```

```
            in2_reg <= 17'b0;
            out1 <= 40'b0;
        end
        else
            begin
                in1_reg <= in1;
                in2_reg <= in2;
                out1 <= in3 - (in1_reg * in2_reg);
            end
    end

    endmodule
```



Resource Usage Report for MultSub (P = Cin - Mult)

The log file resource usage report shows that everything is packed into one MACC block, and one multSub is inferred.

Mapping to part:

Sequential Cells:
SLE 0 uses

DSP Blocks:1
MACC: MultSub

Global Clock Buffers: 1

Total LUTs: 0

Signed MultSub VHDL Example (P = Cin - Mult)

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

entity mult_sub is port (
   in1 : in signed (8 downto 0);
   in2 : in signed (8 downto 0);
   in3 : in signed (16 downto 0); out1 : out signed (17 downto 0)
   );
end mult_sub;

architecture behav of mult_sub is begin
out1 <= in3 - ( in1 * in2 );
end behav;
```

Resource Usage Report for MultSub (P = Cin - Mult)

The log file resource usage report shows that everything is packed into one MACC block, and one multSub is inferred.

Mapping to part:

Sequential Cells:
SLE0 uses

DSP Blocks:1
MACC:1 MultSub

Total LUTs:0

Signed MultSub Verilog Example (P = Mult - Cin)

```
module mult_sub ( in1, in2, in3, clk, rst, out1 );
input signed [16:0] in1, in2;
input signed [36:0] in3;
input clk;
input rst;
output signed [39:0] out1;
reg signed [39:0] out1;
reg signed [16:0] in1_reg, in2_reg;

always @ ( posedge clk )
begin
   if ( ~rst )
      begin
         in1_reg <= 17'b0;
         in2_reg <= 17'b0;
         out1 <= 40'b0;
      end
      else
         begin
            in1_reg <= in1;
            in2_reg <= in2;
            out1 <= (in1_reg * in2_reg) - in3;
         end
      end

endmodule
```

Resource Usage Report for MultSub (P = Mult - Cin)

In this case, the log file shows that only the multiplier and input registers are mapped into the MACC block. The subtractor and output registers are mapped to logic.

Mapping to part:
Cell usage:

Carry primitives used for arithmetic functions:
ARI140 uses

Sequential Cells:
SLE40 uses

DSP Blocks:1
MACC: MultAdd

Global Clock Buffers:1

Total LUTs:0

Unsigned MultSub VHDL Example (P = Mult - Cin)

```
library iEEE;
use iEEE.std_logic_1164.all;
use iEEE.std_logic_unsigned.all;

entity mult_sub is port (
   in1 : in std_logic_vector (8 downto 0);
   in2 : in std_logic_vector (8 downto 0);
   in3 : in std_logic_vector (16 downto 0);
   out1 : out std_logic_vector (17 downto 0)
   );
end mult_sub;

architecture behav of mult_sub is
begin
   out1 <= ( in1 * in2 ) - in3;
end behav;
```

Resource Usage Report for MultSub (P = Mult - Cin)

In this case, the log file shows that only the multiplier is mapped into the MACC block. The subtractor is mapped to logic.

```
Mapping to part:
Carry primitives used for arithmetic functions:
ARI118 uses

Sequential Cells:
SLE0 uses

DSP Blocks:1
 MACC:1 MultAdd

Total LUTs:0
```

Inferring MACC Blocks for Wide Multipliers

A wide multiplier is a multiplier where the width of any of its inputs is larger than 18 bits (signed) or 17 bits (unsigned). The synthesis tool fractures wide multipliers and packs them into multiple MACC blocks, using the cascade and shift functions of the MACC block. A wide multiplier can be configured as either of the following:

- Just one input as wide
- Both inputs as wide

Wide multipliers are implemented by cascading multiple MACC blocks, using the CDOUT and CDIN pins to propagate the cascade output of result P from one MACC block to the cascade input for operand Cx to the next MACC block. The tool also performs the appropriate shifting.

There are some limits to cascade chains, shown in the following table. If the cascade chain exceeds this limit then tool breaks the chain and creates a new cascade chain.

SmartFusion2 and IGLOO2 MACC Block	Maximum Cascaded Size
M2S060T/M2GL060T	24 MACC blocks
M2S050T/M2GL050T	24 MACC blocks
M2S025T/M2GL025T	17 MACC blocks
M2S010T and M2S005 M2GL010T and M2GL005	11 MACC blocks
M2S150T/M2GL150T	40 MACC blocks
M2S090T/M2GL090T	28 MACC Blocks

RTG4 MACC Block	Maximum Cascaded Size
RT4G150	42 MACC blocks
RT4G150_ES	42 MACC blocks

See the following topics for more details about wide multipliers:

- Fracturing Algorithm, on page 23
- Mapping Fractured Multipliers, on page 23
- Cascade Chain, on page 24
- Log File Message, on page 24
- Pipelined Registers with Wide Multipliers, on page 25

Fracturing Algorithm

To be a candidate for fracturing on both inputs, an m-bit x n-bit multiplier must first meet these size requirements:

- For unsigned multipliers, either m or n or both must be greater than 17 bits.
- For signed multipliers, either m or n or both must be greater than 18 bits.

For an m-bit x n-bit multiplier that is a candidate for fracturing on both inputs, there are four multiplications. The final output is computed with these multipliers after performing the appropriate shifting.

```
Mult1 = 17—bit x 17—bit
Mult2 = (m—17)—bit x 17 bit
Mult3 = 17—bit x (n—17)—bit
Mult4 = (m—17)—bit x (n — 17)—bit
```

If the input widths of a fractured multiplier is more than 17 bits (unsigned) or 18 bits (signed), that multiplier is fractured again as needed, until the fractured multiplier can be packed into a single MACC block.

Mapping Fractured Multipliers

When an unsigned multiplier with an input width more than 17 bits or a signed multiplier with an input width more than 18 bits is fractured into multiple multipliers, these multipliers are always packed in multiple MACC blocks. During packing, the tool uses cascade and shift functions without considering the input bit width of fractured multipliers. You can override this default behavior with the syn_multstyle attribute, as described in *Controlling Inference with the syn_multstyle Attribute*, on page 4.

The number of MACC blocks used for packing depends on whether one or both multiplier inputs are configured as wide.

- One input wide

  If only one input is a candidate for fracturing, just that input is fractured. For example, the tool fractures a 20x4-bit unsigned multiplier as follows:

  ```
  Mult1= 17—bit x 4—bit multiplier
  Mult2= 3—bit x 4—bit multiplier
  ```

  Both these multipliers are packed into MACC blocks using cascade and shift functions. See *Example 8: Unsigned 20x17-Bit Multiplier (One Wide Input)*, on page 26 and *Example 9: 21x18-Bit Signed Multiplier (One Wide Input)*, on page 28 for examples.

- Both inputs wide

  If both inputs are candidates for fracturing, they are fractured according to the fracturing algorithm. A 51x26 wide multiplier is fractured as follows:

```
Mult1= 17—bit x 17—bit
Mult2= 34—bit x 17—bit
Mult3= 17—bit x 9—bit
Mult4= 34—bit x 9—bit
```

  Mult2 & Mult4 are further fractured:

Mu1t2	Mu1t4
Mult2_1 = 17—bit x 17—bit	Mult4_1 = 17—bit x 9—bit
Mult2_2 = 17—bit x 17—bit	Mult4_2 = 17—bit x 9—bit

  Based on this fracturing, you get 6 multipliers that are packed into 6 MACC blocks using cascade and shift functions. See *Example 10: Unsigned 26x26-Bit Multiplier (Two Wide Inputs)*, on page 28 and *Example 11: 35x35-Bit Signed Multiplier (Two Wide Inputs)* , on page 29 for examples.

Cascade Chain

SmartFusion2 M2S050T devices support a maximum of 24 MACC blocks being connected in a cascade chain. After fracturing, if the number of mults, multAdds, or multSubs is more than 24, the tool breaks the chain and starts a new cascade chain.

When a multiplier with inputs of 102x102 is synthesized, it is implemented using 36 MACC blocks. A cascade chain is created and the tool breaks the chain after connecting 24 MACC blocks in the cascade, and creates another chain for what is remaining.

If a wide multiplier is followed by an adder or subtractor, only the wide multiplier is packed into the MACC blocks using the cascade and shift functions. The adder or subtractor is mapped to logic.

Log File Message

For each wide multiplier that is implemented using the cascade and shift function, the tool prints a note in the log file. The following is an example:

```
@N: : test.v(43) I Multiplier un1_A[51:0] is implemented with multiple MACC
Blocks using cascade/shift feature.
```

Pipelined Registers with Wide Multipliers

The synthesis tool pipelines registers at the inputs and outputs of wide multipliers in different hierarchies into multiple MACC blocks. The registers must meet the following requirements to be pipelined into wide multiplier structures using cascade and shift functions:

- All the registers to be pipelined must use the same clock.
- Registers to be pipelined in wide multipliers can only be D type flip-flops or D type flip-flop with asynchronous `resets`.
- All input and output registers to be pipelined should be of the same type.
- All registers must have the same control signals.
- The tool first considers output registers for pipelining. If those are not sufficient, the tool considers input registers.
- The maximum number of pipeline stages (including input and output registers) that can be accommodated in wide multiplier structure is <number of MACC blocks> + 1.

The following describe some details of wide multiplier implementations:

- If the input and output registers have different clocks (both inputs have a common clock and the output has a different clock), the output register gets priority and the tool pipelines the output registers into multiple MACC blocks.
- If the output is unregistered and the inputs are registered with different clocks, the input registers are not pipelined in the MACC block.
- For a wide multiplier with registers at inputs and outputs, and an adder/subtractor driven by a wide multiplier, the tool only considers the input registers for pipelining into multiple MACC blocks, as long as all the registers use the same clock. The adder/subtractor and output register are mapped to logic.
- For a wide multiplier with registers at inputs and outputs, and an adder/subtractor driven by a wide multiplier in a different hierarchy, the tool only considers the input registers for pipelining into multiple MACC blocks, as long as all the registers use the same clock. The adder/subtractor and output register are mapped to logic.

For an example, see *Example 13: 35x35-Bit Signed Mult with 2 Pipelined Register Stages*, on page 32.

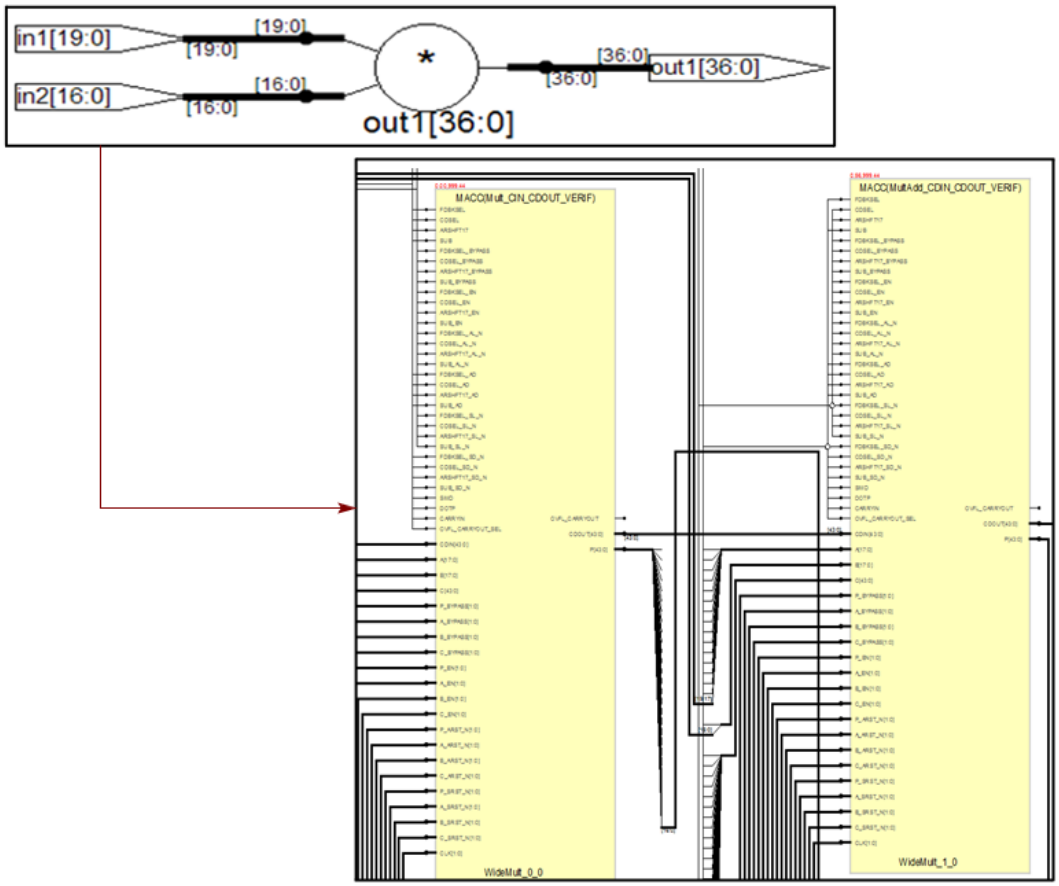
Wide Multiplier Coding Examples

The following examples show how to code wide multipliers so that they are inferred and mapped to MACC blocks, according to the guidelines explained in *Inferring MACC Blocks for Wide Multipliers*, on page 21.

Example 8: Unsigned 20x17-Bit Multiplier (One Wide Input)

This multiplier is split and mapped to two MACC blocks.

```
library iEEE;
use iEEE.std_logic_1164.all;
use iEEE.std_logic_unsigned.all;

entity unsign20x17_mult is port (
   in1 : in std_logic_vector (19 downto 0);
   in2 : in std_logic_vector (16 downto 0);
   out1 : out std_logic_vector (36 downto 0)
   );
end unsign20x17_mult;

architecture behav of unsign20x17_mult is
begin
   out1 <= in1 * in2;
end behav;
```



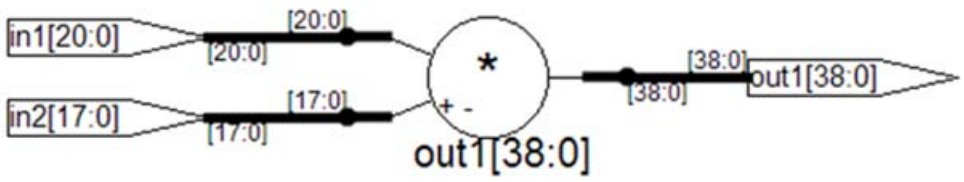
Resource Usage Report for Unsigned 20x17-Bit Multiplier

The report shows that the synthesis tool inferred 1 multAdd and 1 mult, as described in *Mapping Fractured Multipliers*, on page 23.

```
Mapping:

Sequential Cells:
SLE0 uses

DSP Blocks:2
 MACC:1 Mult
 MACC:1 MultAdd

Total LUTs:0
```

Example 9: 21x18-Bit Signed Multiplier (One Wide Input)

```
module sign21x18_mult ( in1, in2, out1 );

input signed [20:0] in1;
input signed [17:0] in2;
output signed [38:0] out1;
wire signed [38:0] out1;
assign out1 = in1 * in2;

endmodule
```



Resource Usage Report for Signed 21x18-Bit Multiplier

In accordance with the fracturing algorithm, the synthesis tool reports the inference of 1 mult and 1 multAdd:

Mapping :

Sequential Cells:

SLE0 uses

DSP Blocks:2
MACC:1 Mult
MACC:1 MultAdd

Total LUTs:0

Example 10: Unsigned 26x26-Bit Multiplier (Two Wide Inputs)

```
library iEEE;
use iEEE.std_logic_1164.all;
use iEEE.std_logic_unsigned.all;

entity unsign26x26_mult is port (
   in1 : in std_logic_vector (25 downto 0);
   in2 : in std_logic_vector (25 downto 0);
   out1 : out std_logic_vector (51 downto 0)
   );
```

```
end unsign26x26_mult;

architecture behav of unsign26x26_mult is
begin
out1 <= in1 * in2;

end behav;
```

Resource Usage Report for Unsigned 26x26-Bit Multiplier

After synthesis, the log report shows that the synthesis tool split this multiplier and mapped it to four MACC blocks. It infers 1 mult and 3 multAdd blocks.

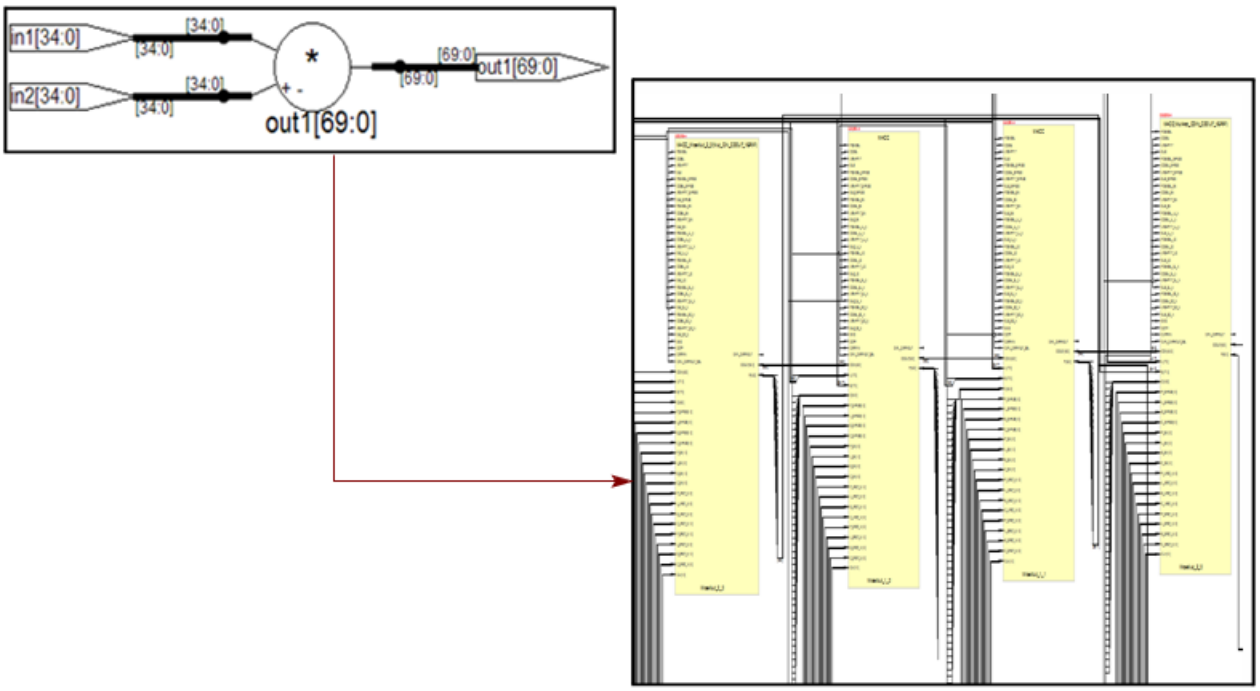
Mapping to part:

Sequential Cells:
SLE0 uses

DSP Blocks:4
MACC:1 Mult
MACC:3 MultAdd

Example 11: 35x35-Bit Signed Multiplier (Two Wide Inputs)

```
module sign35x35_mult ( in1, in2, out1 );
input signed [34:0] in1;
input signed [34:0] in2;
output signed [69:0] out1;
wire signed [69:0] out1;
assign out1 = in1 * in2;
endmodule
```



Resource Usage Report for Signed 35x35-Bit Multiplier

The synthesis tool infers 1 mult and 3 multAdd blocks.

Mapping :

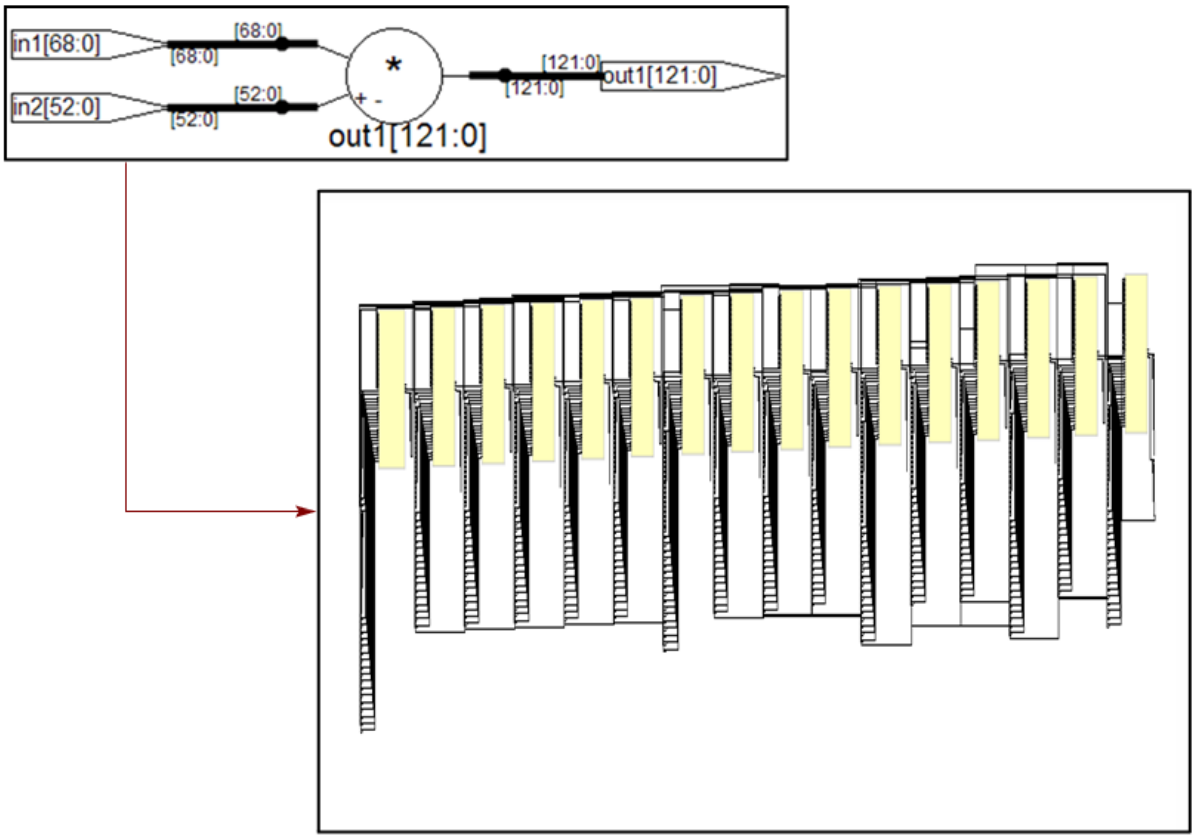
Sequential Cells:
SLE0 uses

DSP Blocks:4
MACC:1 Mult
MACC:3 MultAdds

Total LUTs:0

Example 12: 69x53-Bit Signed Multiplier

```
module sign69x53_mult ( in1, in2, out1 );
input signed [68:0] in1;
input signed [52:0] in2;
output signed [121:0] out1;
wire signed [121:0] out1;
assign out1 = in1 * in2;
endmodule
```



Resource Usage Report for Signed 69x53-Bit Multiplier

The synthesis tool fractures the 69x53 multiplier into one mult and 15 multAdds.

Mapping :

Sequential Cells:
SLE0 uses

DSP Blocks:16
MACC:1 Mult
MACC:15 MultAdds

Total LUTs:0

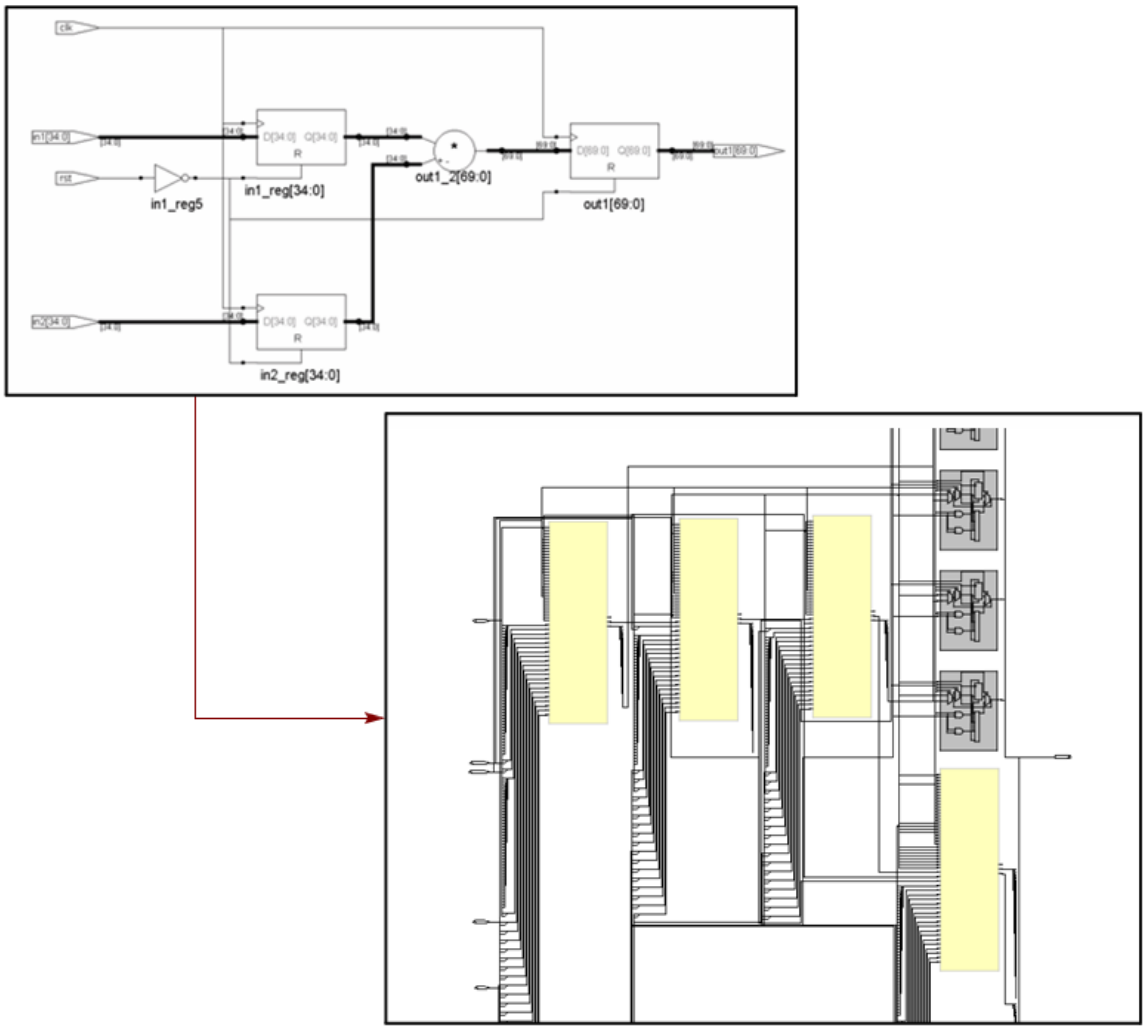
Example 13: 35x35-Bit Signed Mult with 2 Pipelined Register Stages

```
module sign35x35_mult ( in1, in2, clk, rst, out1 );
input signed [34:0] in1, in2;
input clk;
input rst;
output signed [69:0] out1;
reg signed [69:0] out1;
reg signed [34:0] in1_reg, in2_reg;

always @ ( posedge clk or negedge rst)
begin
   if ( ~rst )
      begin
         in1_reg <= 35'b0;
         in2_reg <= 35'b0;
         out1 <= 41'b0;
      end
      else
      begin
         in1_reg <= in1;
         in2_reg <= in2;
         out1 <= in1_reg * in2_reg;
      end
end

endmodule
```

The register pipelining algorithm first pipelines registers at the output of the MACC block, and controls pipeline latency by balancing the number of register stages. To balance the stages, the tool adds registers at either the input or output of the MACC block as required.

This 35x35 signed multiplier requires four MACC blocks, so the tool can pipeline a maximum of 5 register stages. The outputs of instances Widemult_0_0 and Widemult_2_0 are registered. The tool packs the registers at the inputs of the MACC blocks and infers sequential primitives at the output of the MACC blocks for register balancing.

The following figure shows part of the results; not all the registers are shown in the Technology view.



Resource Usage Report for Signed 35x35-Bit Multiplier

The synthesis tool infers 1 mult and 3 multAdd blocks.

Mapping :
Cell usage:

Sequential Cells:
SLE34 uses

DSP Blocks:4
MACC:1 Mult

MACC:3 MultAdds

Global Clock Buffers:1

Total LUTs:0

Example 14: FIR 4 Tap Filter

```
module flat_directform_top (CLK,
   DATAI, COEFI, COEFI_VALID,
   FIRO,
   COEF_SEL );

   parameter TAPS = 4;     // number of filter taps
   parameter DATA_WIDTH    = 12;
   parameter COEF_WIDTH    = 14;
   parameter SYSTOLIC      = 1; // 0 = Direct Form 1 = Pipe-lined Systolic Form
   localparam COEF_ADDR_WIDTH = ceil_log2(TAPS);

   input CLK;  /* synthesis syn_maxfan = 10000 */
   input COEFI_VALID;
   input [DATA_WIDTH-1:0] DATAI;
   input [COEF_WIDTH-1:0] COEFI;
   input [COEF_ADDR_WIDTH-1:0] COEF_SEL;
   output[40:0] FIRO;

   // Coefficient Write Block

   reg[TAPS-1:0] coeff_write_select;
   reg signed [COEF_WIDTH-1:0] coeffreg [TAPS-1:0];
   integer i;

   always  @ (COEFI_VALID, COEF_SEL)
   begin
      for (i=0;i < TAPS; i=i+1)
      begin
         if (i == COEF_SEL)
            coeff_write_select[i] = COEFI_VALID;
         else
            coeff_write_select[i] = 1'b0;
      end  //for
   end // always

   always  @ (posedge CLK)
   begin
      for (i=0;i < TAPS; i=i+1)
      begin
         if (coeff_write_select[i]) coeffreg[i] <= COEFI;
            // Coefficient Register Should Pack Into Mathblock
      end  //for
   end // always

   // Sample Data
```

```
reg signed[DATA_WIDTH-1:0] sample_data[TAPS-1:0];

always  @ (posedge CLK)
begin
   sample_data[0] <= DATAI;
   for (i = 1; i < TAPS; i = i + 1) sample_data[i] <= sample_data[i-1];
end // always

// Calculate Dot Product
reg signed[40:0] FIR_DP;

always  //@ (posedge CLK)
begin
   FIR_DP = 0;
   for (i = 0; i < TAPS; i = i + 1)
   begin
      FIR_DP = FIR_DP + (sample_data[i] * coeffreg[i]);
      // FIR_DP = FIR_DP + (sample_data[i] * coeffreg[i])
      /* synthesis syn_multstyle = "logic" */;
   end  //for
end  // always

generate

if (SYSTOLIC == 1)
begin

reg signed[40:0] pipe_regs[TAPS-1:0];
always  @ (posedge CLK)
begin
   pipe_regs[0] <= FIR_DP;
   for (i = 1; i < TAPS; i = i + 1) pipe_regs[i] <= pipe_regs[i-1];
end // always

assign FIRO = pipe_regs[TAPS-1];

end
else
begin
   reg signed[40:0] pipe_reg;
   always @ (posedge CLK)
   begin
      pipe_reg  <= FIR_DP;
   end // always
   assign FIRO = pipe_reg;

end

endgenerate

/*////////////////////////////////////////////////////
// Function to Calculate Address Width for Coefficients
////////////////////////////////////////////////////*/
function [31:0] ceil_log2;
   input integer x;
```

```
      integer tmp, res;
   begin
      tmp = 1;
      res = 0;
      while (tmp < x) begin
         tmp = tmp * 2;
         res = res + 1;
      end
      ceil_log2 = res;
   end
endfunction

endmodule
```

FIR 4 Tap filter has four stages of pipelined registers at the output. As described in *Example 13: 35x35-Bit Signed Mult with 2 Pipelined Register Stages*, on page 32, the register pipelining algorithm first pipelines registers at the output of the MACC block, and controls pipeline latency by balancing the number of register stages. To balance the stages, the tool adds registers at either the input or output of the MACC block, as required. Depending on the number of pipeline stages, there are a number of levels for the input registers. The tool then packs one level of registers at the input and output into the MACC block and implements the remaining registers using SLE blocks.

The following formula calculates the number of registers implemented using SLE with coding style for FIR 4 Tap filter:

```
n(n-1) / 2 x (a + b1)) + (b2 x (n-1))
```

n = 4 (Tap size)

a = COEFI[13:0] = 14

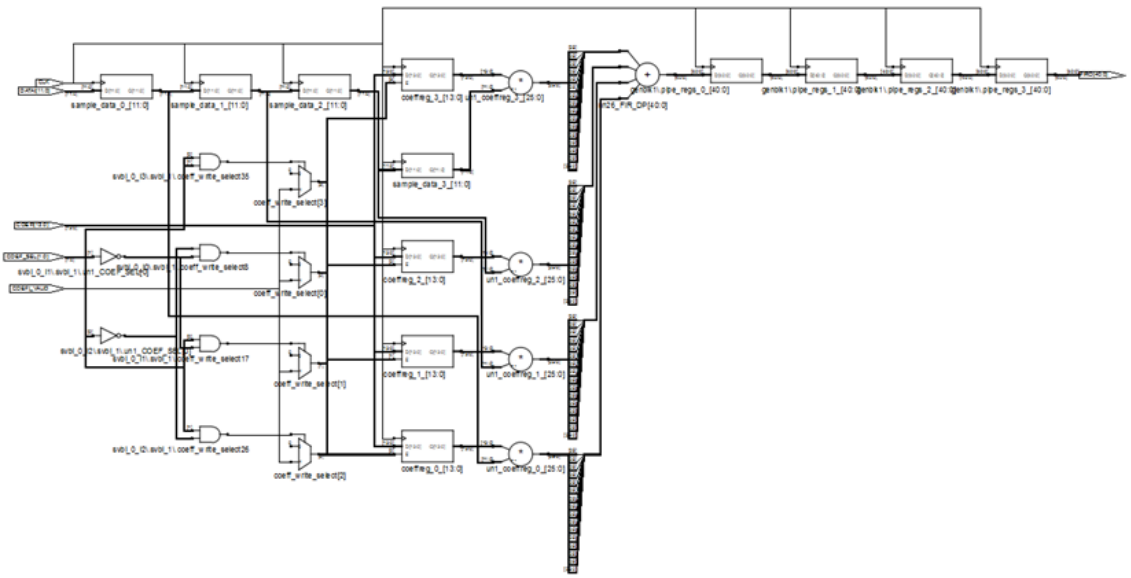
b1 = b2 = DATA[11:0] = 12.

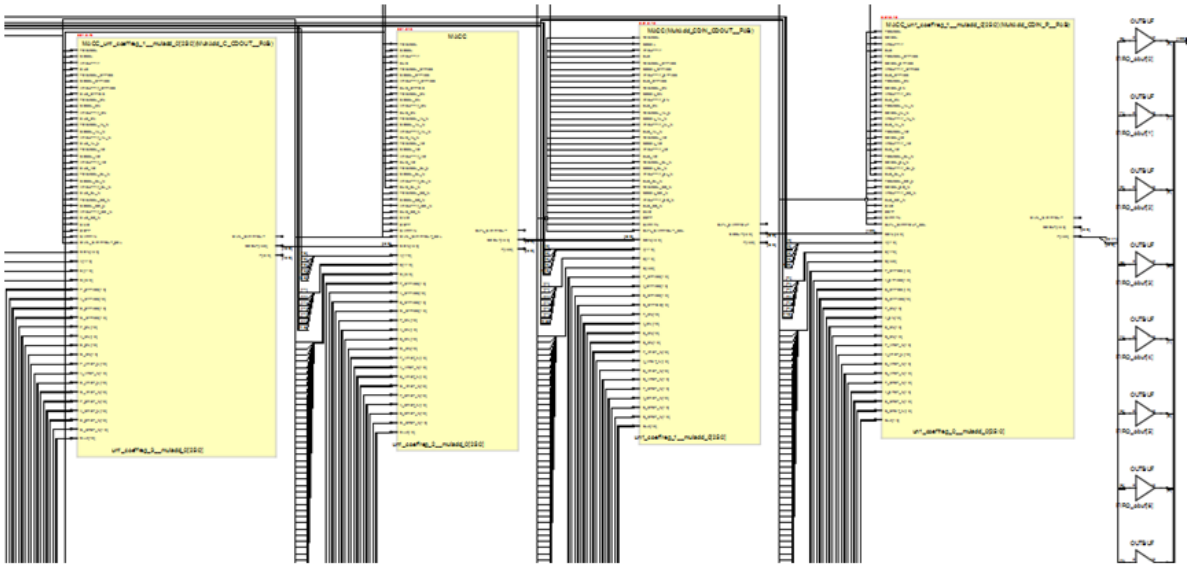
There is a register chain at the sample data input and only one register at the COEFI input. During synthesis, all output registers are pushed to the input side of the multipliers during pipelining. A warning message in the log file informs you that the tool is removing sequential instance *coeffreg* because it is equivalent to instance sample_data*. The synthesis tool optimizes the register at the COEFI input and uses the output from the equivalent sample_data register. Therefore, all registers being pushed for pipelining at input b1 are optimized and the value of b1 becomes 0.

If you substitute values for n, a, b1, and b2 into the equation, you get this formula:

```
4(4-1) / 2 x (14 + 0) ) + (12 x (4 -1) = (12/2 x 14 + (12 x 3) = 84 + 36=120
```

Use this formula for any FIR tap filter written with this coding style, to calculate the number of registers implemented using SLE.





Resource Usage FIR 4 Tap Filter

Mapping :
Cell usage:
CFG34 uses

Sequential Cells:
SLE120 uses

DSP Blocks:4
MACC:4 MultAdds

Total LUTs:4

Inferring MACCs for Multi-Input MultAdds/MultSubs

The MACC block cascade feature supports multi-input multAdd and multSub implementations for devices with MACC blocks. The tool packs logic into MACC blocks efficiently using hard-wired cascade paths, and improves the quality of results (QoR) for the design.

To use the cascade feature, the design must meet these requirements:

- The input size for multipliers must not be greater than 18x18 bits (signed) and 17x17 bits (unsigned).
- Signed multipliers must have the proper sign-extension.
- All multiplier output bits must feed the adder.
- Multiplier inputs and outputs may be registered or unregistered.

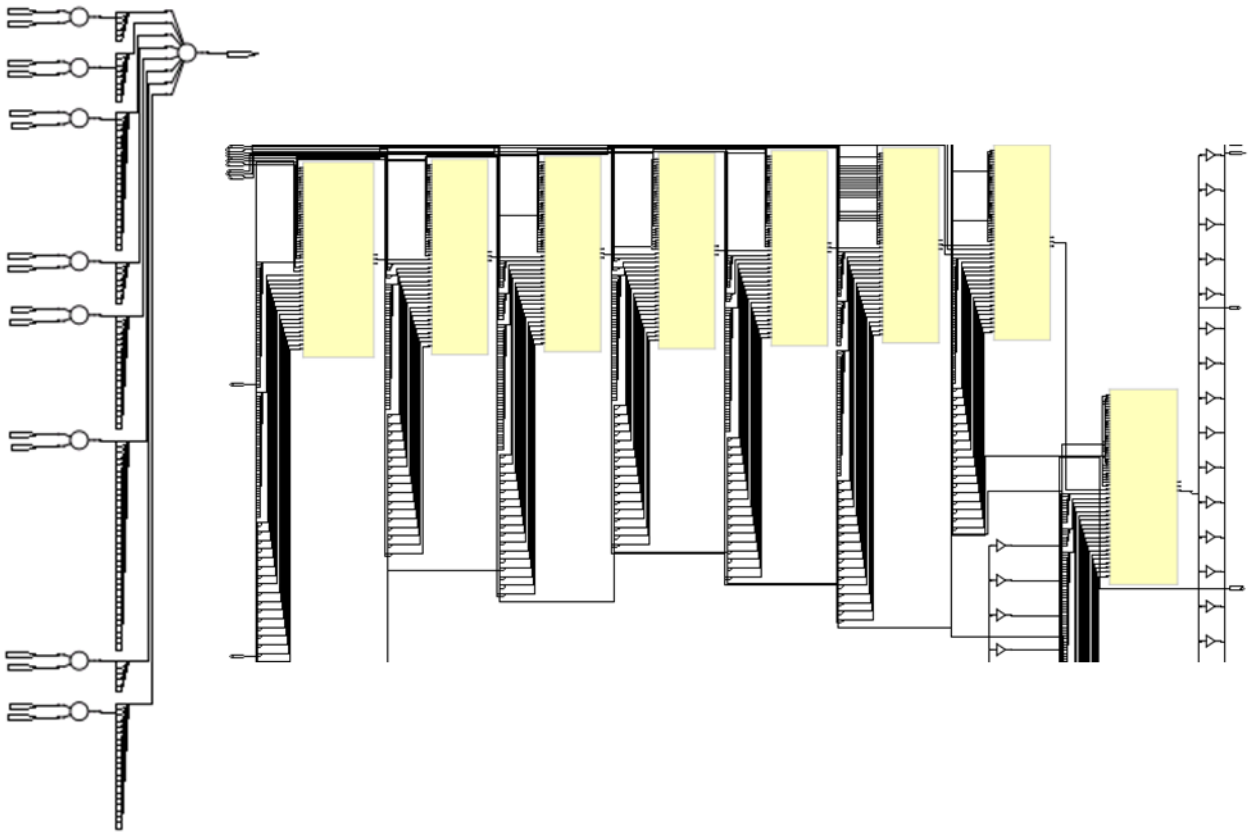
Example 15: VHDL Test for 8 MultAdd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity test is
generic (widtha : integer := 18;
         widthb : integer := 18;
         widthc : integer := 16;
         widthd : integer := 17;
         widthe : integer := 9;
         widthf : integer := 9;
         widthg : integer := 17;
         widthh : integer := 17;
         widthi : integer := 7;
         widthj : integer := 15;
         widthk : integer := 3;
```

```
         widthl : integer := 3;
         widthm : integer := 18;
         widthn : integer := 18;
         widtho : integer := 8;
         widthp : integer := 12;
         width_out : integer := 41
         );
port (ina : in std_logic_vector(widtha-1 downto 0);
      inb : in std_logic_vector(widthb-1 downto 0);
      inc : in std_logic_vector(widthc-1 downto 0);
      ind : in std_logic_vector(widthd-1 downto 0);
      ine : in std_logic_vector(widthe-1 downto 0);
      inf : in std_logic_vector(widthf-1 downto 0);
      ing : in std_logic_vector(widthg-1 downto 0);
      inh : in std_logic_vector(widthh-1 downto 0);
      ini : in std_logic_vector(widthi-1 downto 0);
      inj : in std_logic_vector(widthj-1 downto 0);
         ink : in std_logic_vector(widthk-1 downto 0);
         inl : in std_logic_vector(widthl-1 downto 0);
         inm : in std_logic_vector(widthm-1 downto 0);
         inn : in std_logic_vector(widthn-1 downto 0);
         ino : in std_logic_vector(widtho-1 downto 0);
         inp : in std_logic_vector(widthp-1 downto 0);
         dout : out std_logic_vector(width_out-1 downto 0)
         );
end entity test;

architecture arc of test is
      function sign_ext ( v_in : std_logic_vector; new_size : natural)
         return std_logic_vector is
         variable size_in : natural;
         variable result : std_logic_vector (new_size - 1 downto 0);
      begin
         result := (others => v_in(v_in'left));
         result (v_in'length - 1 downto 0) := v_in;
         return result;
      end sign_ext;

      signal ina_sig : std_logic_vector(widtha-1 downto 0);
      signal inb_sig : std_logic_vector(widthb-1 downto 0);
      signal inc_sig : std_logic_vector(widthc-1 downto 0);
      signal ind_sig : std_logic_vector(widthd-1 downto 0);
      signal ine_sig : std_logic_vector(widthe-1 downto 0);
      signal inf_sig : std_logic_vector(widthf-1 downto 0);
      signal ing_sig : std_logic_vector(widthg-1 downto 0);
      signal inh_sig : std_logic_vector(widthh-1 downto 0);
      signal ini_sig : std_logic_vector(widthi-1 downto 0);
      signal inj_sig : std_logic_vector(widthj-1 downto 0);
      signal ink_sig : std_logic_vector(widthk-1 downto 0);
      signal inl_sig : std_logic_vector(widthl-1 downto 0);
      signal inm_sig : std_logic_vector(widthm-1 downto 0);
      signal inn_sig : std_logic_vector(widthn-1 downto 0);
      signal ino_sig : std_logic_vector(widtho-1 downto 0);
      signal inp_sig : std_logic_vector(widthp-1 downto 0);
```

```
      signal prod1 : std_logic_vector(widtha+widthb-1 downto 0);
      signal prod2 : std_logic_vector(widthc+widthd-1 downto 0);
      signal prod3 : std_logic_vector(widthe+widthf-1 downto 0);
      signal prod4 : std_logic_vector(widthg+widthh-1 downto 0);
      signal prod5 : std_logic_vector(widthi+widthj-1 downto 0);
      signal prod6 : std_logic_vector(widthk+widthl-1 downto 0);
      signal prod7 : std_logic_vector(widthm+widthn-1 downto 0);
      signal prod8 : std_logic_vector(widtho+widthp-1 downto 0);

      signal padprod1 : signed(width_out-widtha-widthb-1 downto 0);
      signal padprod2 : signed(width_out-widthc-widthd-1 downto 0);
      signal padprod3 : signed(width_out-widthe-widthf-1 downto 0);
      signal padprod4 : signed(width_out-widthg-widthh-1 downto 0);
      signal padprod5 : signed(width_out-widthi-widthj-1 downto 0);
      signal padprod6 : signed(width_out-widthk-widthl-1 downto 0);
      signal padprod7 : signed(width_out-widthm-widthn-1 downto 0);
      signal padprod8 : signed(width_out-widtho-widthp-1 downto 0);
   begin

ina_sig <= sign_ext (ina,widtha);
inb_sig <= sign_ext (inb,widthb);
inb_sig <= sign_ext (inb,widthc);
inb_sig <= sign_ext (inb,widthd);
inb_sig <= sign_ext (inb,widthe);
inb_sig <= sign_ext (inb,widthf);
inb_sig <= sign_ext (inb,widthg);
inb_sig <= sign_ext (inb,widthh);
inb_sig <= sign_ext (inb,widthi);
inb_sig <= sign_ext (inb,widthj);
inb_sig <= sign_ext (inb,widthk);
inb_sig <= sign_ext (inb,widthl);
inb_sig <= sign_ext (inb,widthm);
inb_sig <= sign_ext (inb,widthn);
inb_sig <= sign_ext (inb,widtho);
inb_sig <= sign_ext (inb,widthp);
prod1 <= ina_sig * inb_sig;
prod2 <= inc_sig * ind_sig;
prod3 <= ine_sig * inf_sig;
prod4 <= ing_sig * inh_sig;
prod5 <= ini_sig * inj_sig;
prod5 <= ink_sig * inl_sig;
prod6 <= inm_sig * inn_sig;
prod7 <= ino_sig * inp_sig;

padprod1 <= (others => prod1 (widtha+widthb-1));
padprod2 <= (others => prod2 (widthc+widthd-1));
padprod3 <= (others => prod3 (widthe+widthf-1));
padprod4 <= (others => prod4 (widthg+widthh-1));
padprod5 <= (others => prod5 (widthi+widthj-1));
padprod6 <= (others => prod6 (widthk+widthl-1));
padprod7 <= (others => prod7 (widthm+widthn-1));
padprod8 <= (others => prod8 (widtho+widthp-1));

dout <= ((padprod1 & signed(prod1)) + (padprod2 & signed(prod2)) +
```

```
            (padprod3 & signed(prod3)) + (padprod4 & signed(prod4)) +
            (padprod5 & signed(prod5)) + (padprod6 & signed(prod6)) +
            (padprod7 & signed(prod7)) + (padprod8 & signed(prod8)));

    end arc;
```



Resource Usage Report for 8 MultAdd

The synthesis tool infers 8 multAdd blocks.

Mapping to part:

Sequential Cells:
SLE0 uses

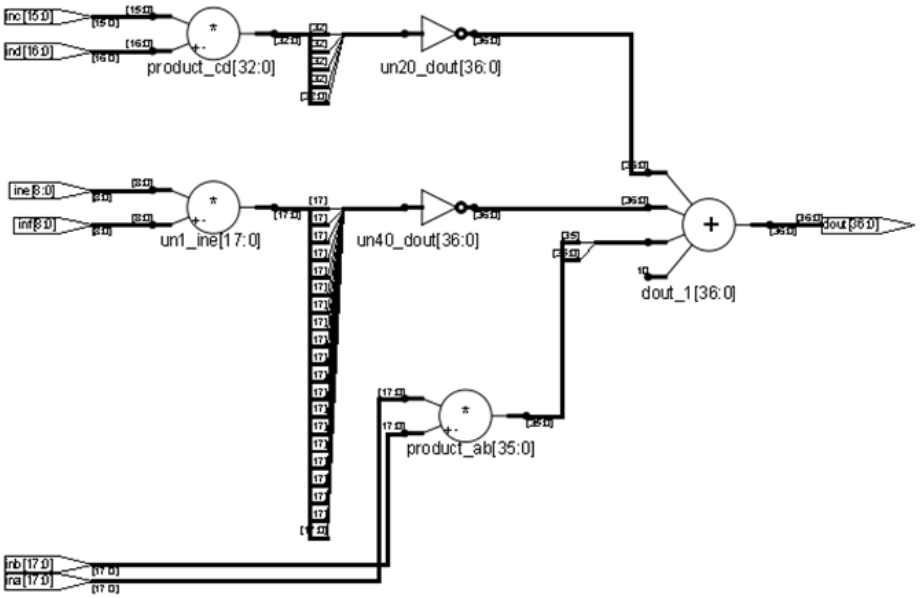
DSP Blocks:8
MACC:8 MultAdds

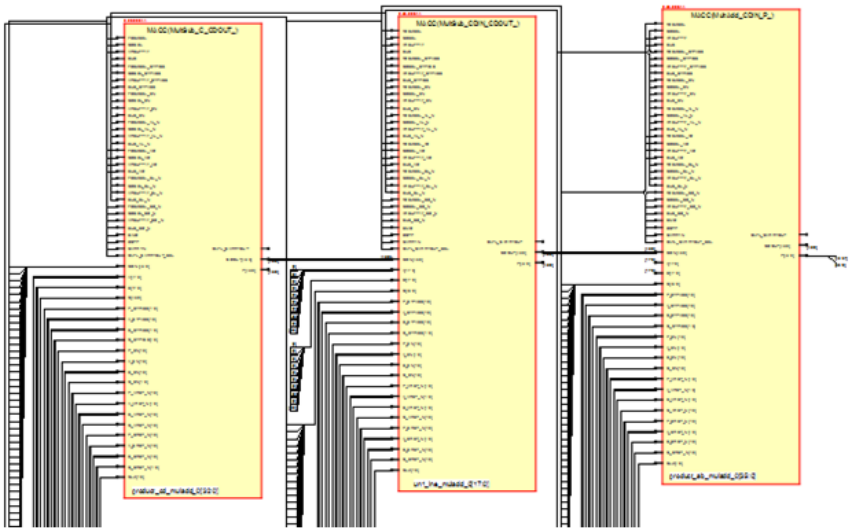
Total LUTs:0

Example 16: Verilog Test for 3 MultSub

```
`timescale 1 ns/100 ps

`ifdef synthesis
module test ( ina, inb, inc, ind, ine, inf, dout);
`else
module test_rtl ( ina, inb, inc, ind, ine, inf, dout);
`endif

parameter widtha = 18;
parameter widthb = 18;
parameter widthc = 16;
parameter widthd = 17;
parameter widthe = 9;
parameter widthf = 9;
parameter width_out = 37;

input signed [widtha-1:0] ina;
input signed [widthb-1:0] inb;
input signed [widthc-1:0] inc;
input signed [widthd-1:0] ind;
input signed [widthe-1:0] ine;
input signed [widthf-1:0] inf;
output reg signed [width_out-1:0] dout;

function signed [widtha+widthb-1:0] product_ab;
   input [widtha-1:0] DA;
   input [widthb-1:0] DB;
   reg [widtha-1:0] D_A;
   reg [widthb-1:0] D_B;
   integer DataAi;
   integer DataBi;
   reg signed [widtha+widthb-1:0] add_sub;

   begin
      D_A = {widtha{1'b1}};
      D_B = {widthb{1'b1}};
      if(DA[widtha-1])
         DataAi = -(D_A-DA+1);
      else
         DataAi = DA;

      if(DB[widthb-1])
         DataBi = -(D_B-DB+1);
      else
         DataBi = DB;
         add_sub = (DataAi * DataBi);
         product_ab = add_sub;
      end
endfunction

function signed [widthc+widthd-1:0] product_cd;
   input [widthc-1:0] DC;
   input [widthd-1:0] DD;
```

```
   reg [widthc-1:0] D_C;
   reg [widthd-1:0] D_D;
   integer DataCi;
   integer DataDi;
   reg signed [widthc+widthd-1:0] add_sub;

   begin
      D_C = {widthc{1'b1}};
      D_D = {widthd{1'b1}};
      if(DC[widthc-1])
         DataCi = -(D_C-DC+1);
      else
         DataCi = DC;

      if(DD[widthd-1])
         DataDi = -(D_D-DD+1);
      else
         DataDi = DD;

         add_sub = (DataCi * DataDi);
         product_cd = add_sub;
   end
endfunction

function signed [widthe+widthf-1:0] product_ef;
   input [widthe-1:0] DE;
   input [widthf-1:0] DF;
   reg [widthe-1:0] D_E;
   reg [widthf-1:0] D_F;
   integer DataEi;
   integer DataFi;
   reg signed [widthe+widthf-1:0] add_sub;

   begin
      D_E = {widthe{1'b1}};
      D_F = {widthf{1'b1}};
      if(DE[widthe-1])
         DataEi = -(D_E-DE+1);
      else
         DataEi = DE;

      if(DF[widthf-1])
         DataFi = -(D_F-DF+1);
      else
         DataFi = DF;

         add_sub = (DataEi * DataFi);
         product_ef = add_sub;
   end
endfunction

always @(*)
   dout = product_ab(ina, inb) - product_cd(inc, ind) - product_ef(ine, inf);

endmodule
```





Resource Usage Report for 3 MultSub

The synthesis tool infers 1 multAdd and 2 multSub blocks.

Mapping :
Sequential Cells:
SLE0 uses

DSP Blocks: 3
MACC:1 MultAdd
MACC:2 MultSubs

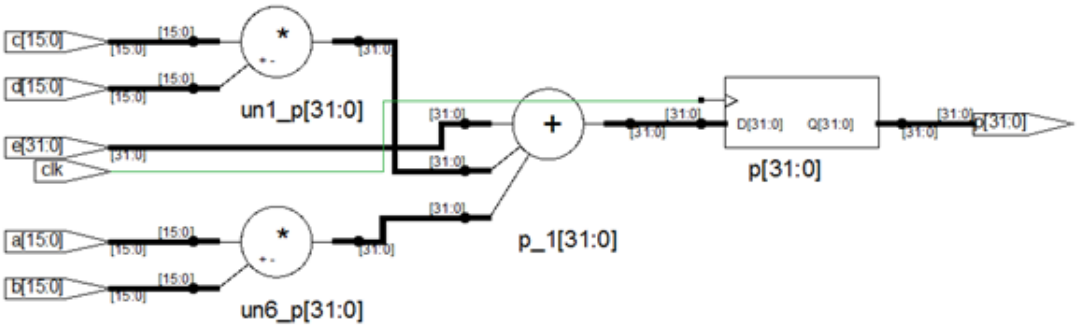
Example 17: Complex Expression Example

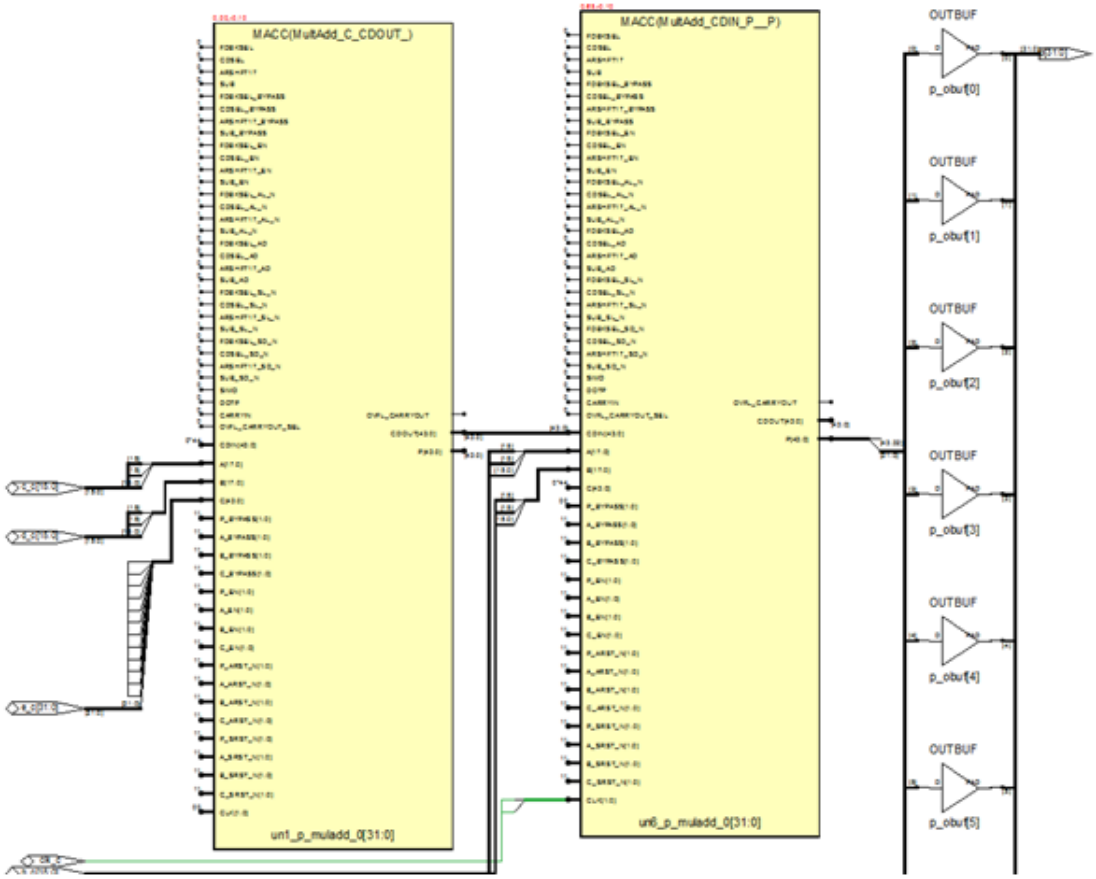
```
module test(clk,a, b, c, d, e, p);

parameter M = 16;
parameter N = 16;

input clk;
input signed[M-1:0] a, b;
input signed[N-1:0] c, d;
input signed[N*2-1:0] e;
output signed[M+N-1:0] p;

reg [M+N-1:0] p;
always@(posedge clk)
begin
p = e + (c * d) + (a * b);
end
endmodule
```

Resource Usage Report for Test

The synthesis tool infers two multAdd blocks.

```
Mapping :
Cell usage:
CLKINT        1 use

Sequential Cells:
SLE           0 uses

DSP Blocks:    2
MACC:         2 MultAdds
```





Inferring MACC Blocks for Multiplier-AddSub

The MACC block supports dynamic additions and subtractions. It uses the sub input of the MACC block to select the ADD or SUB operations. The design must conform to these prerequisites:

- Input size for multipliers must not be greater than 18x18 bits (signed) and 17x17 bits (unsigned). The tool does not infer multAdds and multSubs with wide multipliers.
- Signed multipliers must have the proper sign extension.
- The multiplier output used for addition or subtraction must be specified:

  Prod = A * B
  Sum = Sub ? (C - Prod) : (C + Prod)

- Multiplier inputs and outputs can be registered or unregistered.

Example 18: One MultAddSub (Verilog)

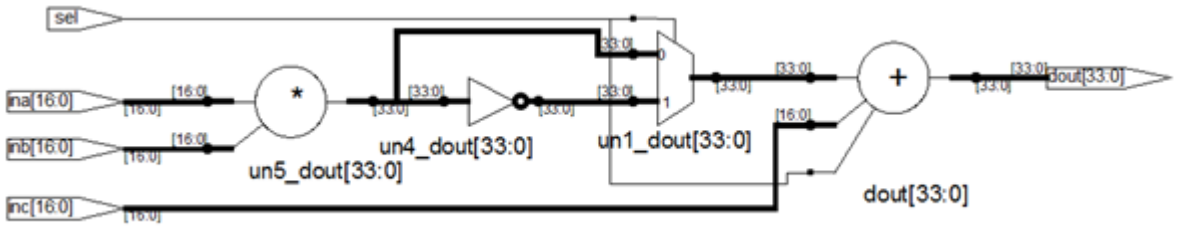
```
module test ( ina, inb, inc, dout, sel);

parameter widtha = 17;
parameter widthb = 17;
parameter widthc = 17;
parameter width_out = 34;

input [widtha-1:0] ina;
input [widthb-1:0] inb;
input [widthc-1:0] inc;
input sel;

output [width_out-1:0] dout;

assign dout = sel ? inc - (ina * inb)  : inc + (ina * inb) ;
endmodule
```

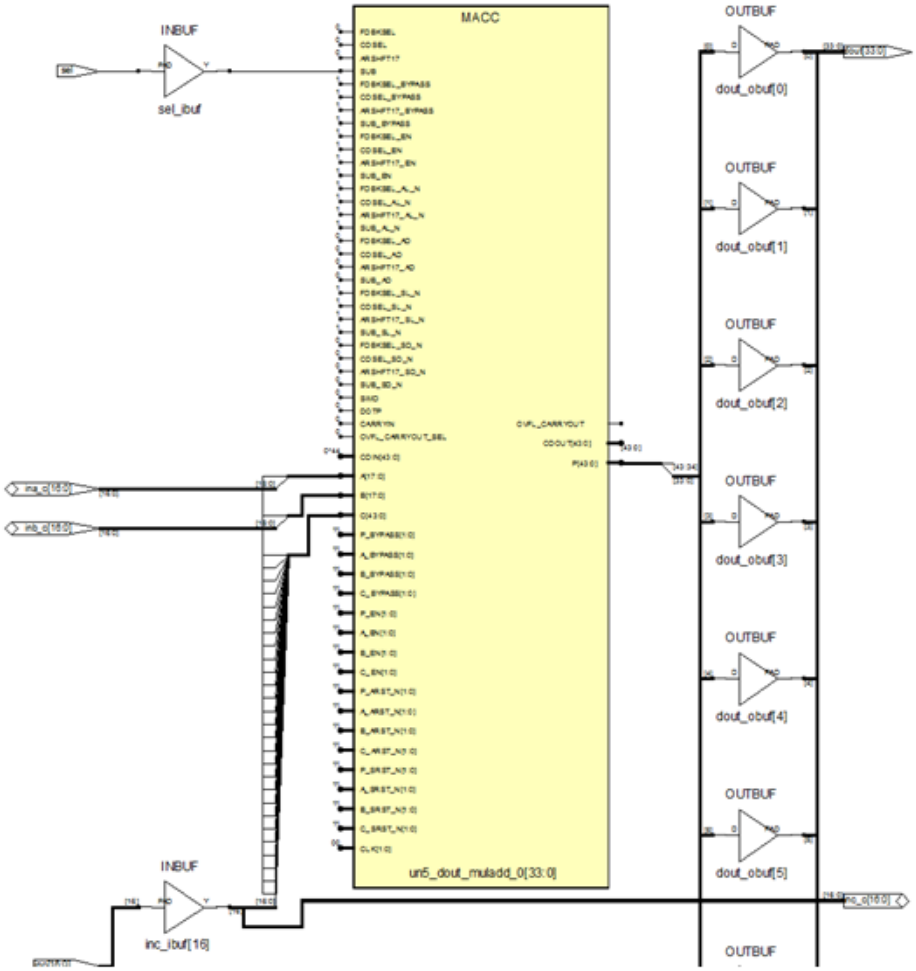
Resource Usage Report for Test

The synthesis tool infers 1 MultAddSub block.

Mapping :

Sequential Cells:
SLE 0 uses
DSP Blocks: 1
MACC: 1 MultAddSub





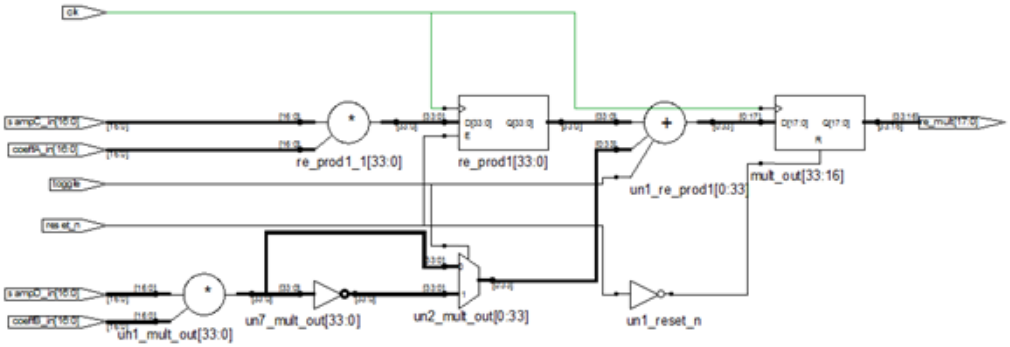
Example 19: One MultAddSub (VHDL)

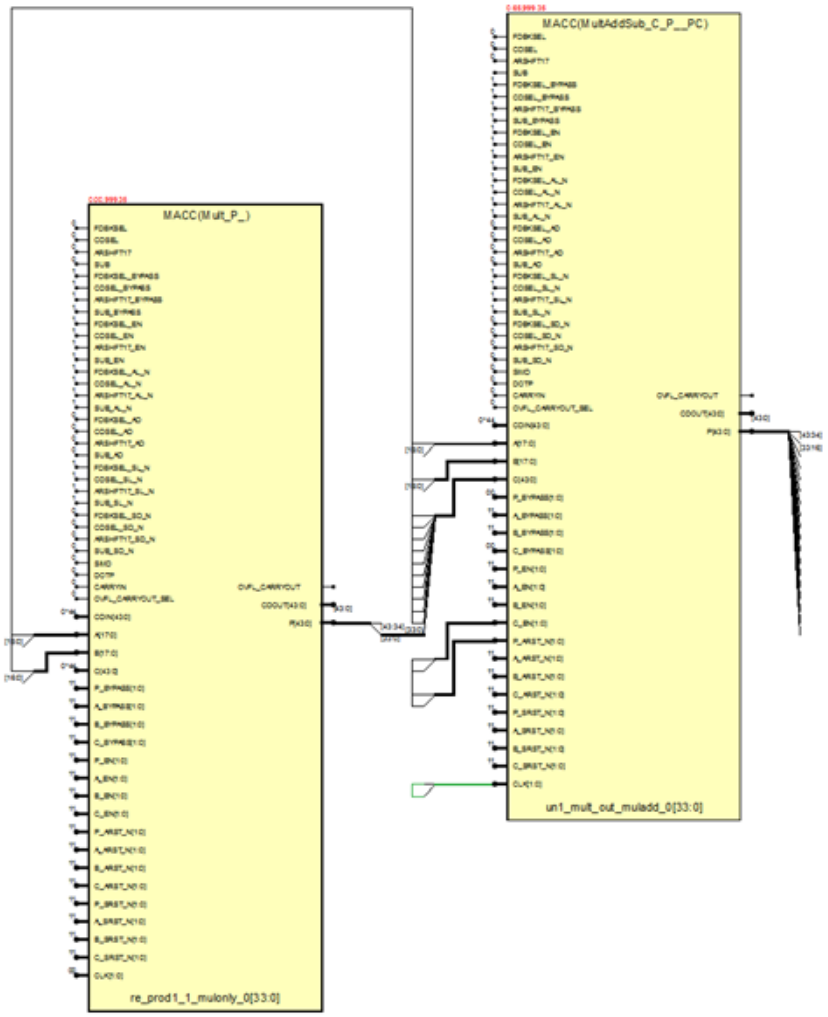
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity test is
      generic (a_width : integer := 17;
      b_width : integer := 17);
      port(     clk : in std_logic;
            reset_n : in std_logic;
            sampC_in : in std_logic_vector(a_width-1 downto 0);
            sampD_in : in std_logic_vector(a_width-1 downto 0); --imag
            coeffA_in : in std_logic_vector(b_width-1 downto 0);
            coeffB_in : in std_logic_vector(b_width-1 downto 0); --imag
            toggle : in std_logic; -- 1=SUB; 0=ADD
            re_mult : out std_logic_vector(a_width downto 0)
      );
end test;

architecture DEF_ARCH of test is

      signal re_prod1 : std_logic_vector(a_width+b_width-1 downto 0);
      signal mult_out : std_logic_vector(a_width+b_width-1 downto 0);
begin
      process(clk,reset_n)
      begin
      if (reset_n = '0') then
               mult_out <= (others => '0'); -- 1=SUB; 0=ADD
elsif rising_edge(clk) then
         re_prod1 <= sampC_in * coeffA_in;
         if (toggle = '0') then
               mult_out <= re_prod1 + (sampD_in * coeffB_in);
         else
               mult_out <= re_prod1 - (sampD_in * coeffB_in);
         end if;
   end if;
end process;
re_mult(a_width downto 0) <= mult_out(a_width+b_width-1 downto b_width-1);
end def_arch;
```

Resource Usage Report for Test

The synthesis tool infers 1 MultAddSub block and 1 Mult block.

```
Mapping :
Cell usage:
CLKINT       1 use

Sequential Cells:
SLE          0 uses

DSP Blocks:    2
MACC:        1 MultAddSub
MACC:        1 Mult
```





Inferring MACC Blocks for Multiplier-Accumulators

The multiplier-accumulator structures use internal paths for adder feedback loops inside the MACC block instead of connecting them externally. To implement these structures, the design must meet these requirements:

- The input size for multipliers must not be greater than 18x18 bits (signed) and 17x17 bits (unsigned).
- Signed multipliers must have the proper sign extension.
- All multiplier output bits must feed the adder.
- The output of the adder must be registered.
- The registered output of the adder must feed back to the adder for accumulation.
- Only multiplier-Accumulator structures with one multiplier can be packed inside the MACC block, because the MACC block contains only one multiplier, .

The multiplier-accumulator structure also supports synchronous loadable registers. To infer these structures the design must meet the requirements listed above, as well as the requirements listed here:

- For the loading multiplier-accumulator structure, new load data must be passed to input C.
- The loadEn signal must be registered.

Example 20: Verilog Test for 18X18 MultAcc with Load

```
`timescale 1 ns/100 ps

`ifdef synthesis
module test ( clk, rst, ld, ina, inb, inc, dout);
`else
module test_rtl ( clk, rst, ld, ina, inb, inc, dout);
`endif

parameter widtha = 18;
parameter widthb = 18;
parameter widthc = 41;
parameter width_out = 41;

input clk, rst, ld;
input signed [widtha-1:0] ina;
input signed [widthb-1:0] inb;
input signed [widthc-1:0] inc;
output reg signed [width_out-1:0] dout;

wire signed [width_out-1:0] mult1;
reg signed [widtha-1:0] ina_reg;

always@(posedge clk or negedge rst)
   if(~rst)
      ina_reg <= {widtha{1'b0}} ;
```

```
   else
      ina_reg <= ina ;

assign mult1 = ina_reg * inb;

reg ld_reg;

always@(posedge clk or negedge rst)
   if(~rst)
      ld_reg <= {width_out{1'b0}} ;
   else
      ld_reg <= ld ;

always@(posedge clk or negedge rst)
   if(~rst)
      dout <= {width_out{1'b0}} ;
   else
      if(ld_reg)
         dout <= inc ;
      else
         dout <= mult1 + dout ;

endmodule
```

Resource Usage Report for 18x18 MultAcc with Load

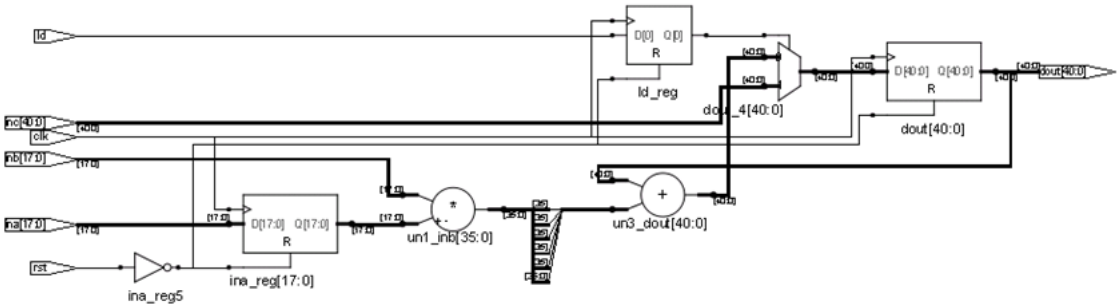
The synthesis tool infers 1 multAcc block.

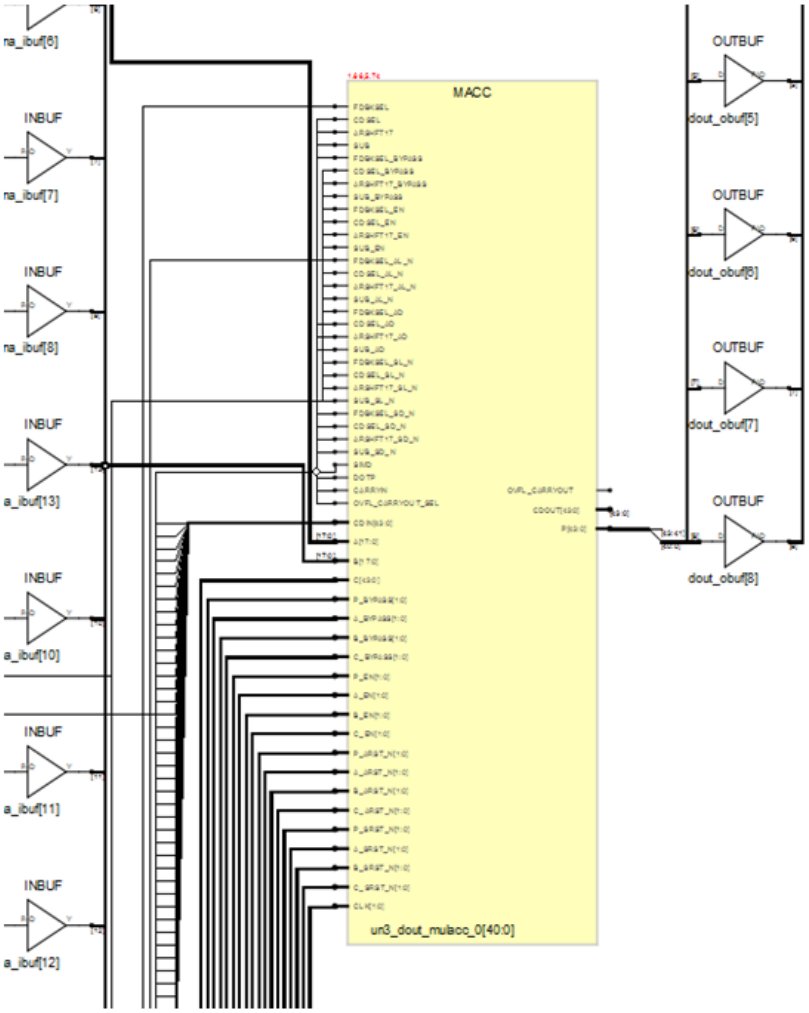
Mapping :
Cell usage:
CFG11 use
CFG241 uses

Sequential Cells:
SLE1 use

DSP Blocks:1
MACC: 1 MultAcc

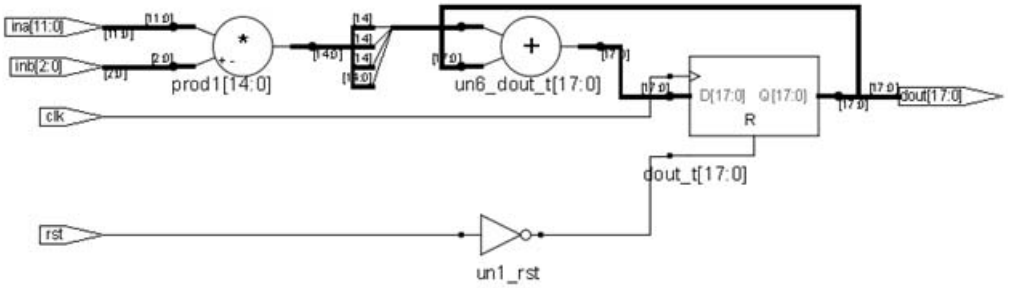
Total LUTs: 42

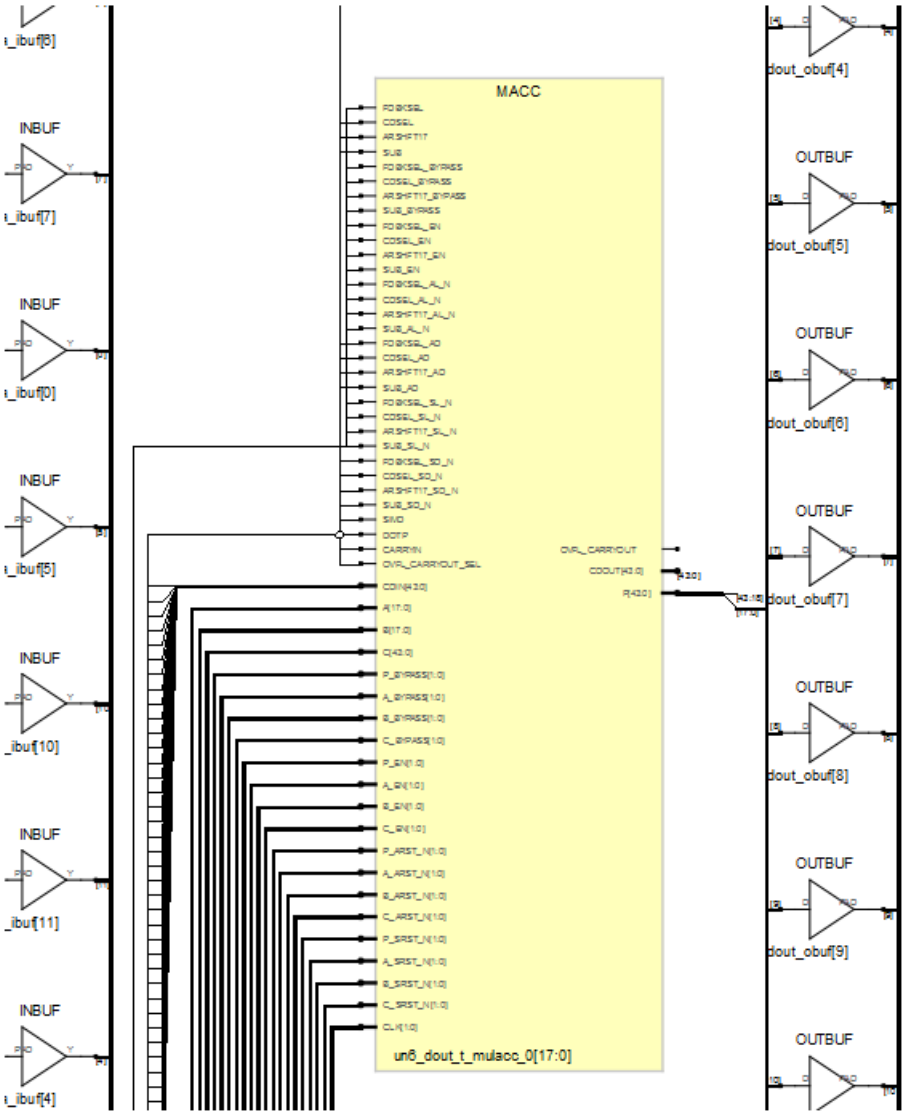




Example 21: VHDL Test for 12X3 MultAcc Without Load

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity test is
generic (widtha : integer := 12;
         widthb : integer := 3;
         width_out : integer := 18
         );
port (clk : in std_logic;
      rst : in std_logic;
      ina : in std_logic_vector(widtha-1 downto 0);
      inb : in std_logic_vector(widthb-1 downto 0);
      dout : out std_logic_vector(width_out-1 downto 0)
      );
end entity test;

architecture arc of test is
   signal prod1 : std_logic_vector(widtha+widthb-1 downto 0);
   signal padprod1 : signed(width_out-widtha-widthb-1 downto 0);
   signal dout_t : std_logic_vector(width_out-1 downto 0);
begin

prod1 <= (signed(ina) * signed(inb));
padprod1 <= (others => prod1(widtha+widthb-1));

process(clk,rst)
begin
   if(rst='0')then
      dout_t <= (others => '0');
   elsif(clk'event andclk='1')then
         dout_t <= conv_std_logic_vector((padprod1 & signed(prod1)),width_out)
            + dout_t;
   end if;
end process;

dout <= dout_t;

end arc;
```





Resource Usage Report for 12x3 MultAcc Without Load

The synthesis tool infers 1 MultAcc block.

Mapping :

Sequential Cells:
SLE0 uses

DSP Blocks:1
MACC:1 MultAcc

Total LUTs:0

Coding Examples for Timing and QoR Improvement

The following examples show coding styles that result in better timing and QoR.

- Example 22: MultAdd, on page 56
- Example 23: MultAdd with Pipelined Registers, on page 59

Example 22: MultAdd

This example is a normal multAdd structure which gives ~456MHz after place and route.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity test is
   port(clk                : in std_logic;
   reset_n              : in std_logic;
   Xn_in              : in signed(15 downto 0);
   Yn_out              : out signed(15 downto 0)
   );

end test;

architecture DEF_ARCH of test is

    constant    b0_coeff        : signed(15 downto 0) := x"7FFF";
   constant    b1_coeff        : signed(15 downto 0) := x"7FFF";
   constant    b2_coeff        : signed(15 downto 0) := x"7FFF";
   constant    a1_coeff        : signed(15 downto 0) := x"7FFF";
   constant    a2_coeff        : signed(15 downto 0) := x"7FFF";
   constant    scale_factor    : signed(15 downto 0) := x"FF3F";
```

```
    signal Xn_reg1           : signed(15 downto 0);
   signal Xn_reg2           : signed(15 downto 0);
--    signal Xn_reg3           : signed(15 downto 0);
--    signal Yn_reg1           : signed(15 downto 0);
--    signal Yn_reg2           : signed(15 downto 0);
   signal b0_mult           : signed(31 downto 0);
   signal b1_mult           : signed(31 downto 0);
   signal b2_mult           : signed(31 downto 0);
   signal a1_mult           : signed(31 downto 0);
   signal a2_mult           : signed(31 downto 0);
   signal pad_b0_mult       : signed(11 downto 0);
   signal pad_b1_mult       : signed(11 downto 0);
   signal pad_b2_mult       : signed(11 downto 0);
   signal pad_a1_mult       : signed(11 downto 0);
   signal pad_a2_mult       : signed(11 downto 0);
--    signal scale_reg        : signed(31 downto 0);
signal scale_reg1       : signed(15 downto 0);
signal scale_reg2       : signed(15 downto 0);
signal sum_out              : signed(43 downto 0);
signal sum_out1          : signed(43 downto 0);
signal sum_out2          : signed(43 downto 0);
signal sum_out3          : signed(43 downto 0);
--    signal sum_out4          : signed(43 downto 0);

begin

    process(clk, reset_n)
   begin
      if (reset_n = '0') then
            Xn_reg1       <= (others => '0');
            Xn_reg2       <= (others => '0');
            scale_reg1    <= (others => '0');
            scale_reg2    <= (others => '0');
            sum_out        <= (others => '0');
            sum_out1        <= (others => '0');
            sum_out2        <= (others => '0');
            sum_out3        <= (others => '0');
--            sum_out4        <= (others => '0');

        elsif rising_edge(clk) then
            Xn_reg1        <= Xn_in;
            Xn_reg2        <= Xn_reg1;
            scale_reg1    <= sum_out(31 downto 16);
            scale_reg2    <= scale_reg1;

      sum_out1        <= (pad_b0_mult & (b0_mult)) + (pad_b1_mult & (b1_mult));
      sum_out2        <= (pad_b2_mult & (b2_mult)) + sum_out1;
      sum_out3        <= (pad_a1_mult & (a1_mult)) + sum_out2;
      sum_out        <= (pad_a2_mult & (a2_mult)) + sum_out3;

        end if;
      end process;

   b0_mult        <= Xn_in      * b0_coeff;
   b1_mult        <= Xn_reg1    * b1_coeff;
   b2_mult        <= Xn_reg2    * b2_coeff;
   a1_mult        <= scale_reg1 * a1_coeff;
   a2_mult        <= scale_reg2 * a2_coeff;
```

```
    pad_b0_mult     <= (others => b0_mult(31));
    pad_b1_mult     <= (others => b1_mult(31));
    pad_b2_mult     <= (others => b2_mult(31));
    pad_a1_mult     <= (others => a1_mult(31));
    pad_a2_mult     <= (others => a2_mult(31));

    Yn_out    <= sum_out(31 downto 16);


end def_arch;

--
------------------------------------------------------------------------------

-- end of code

------------------------------------------------------------------------------
```

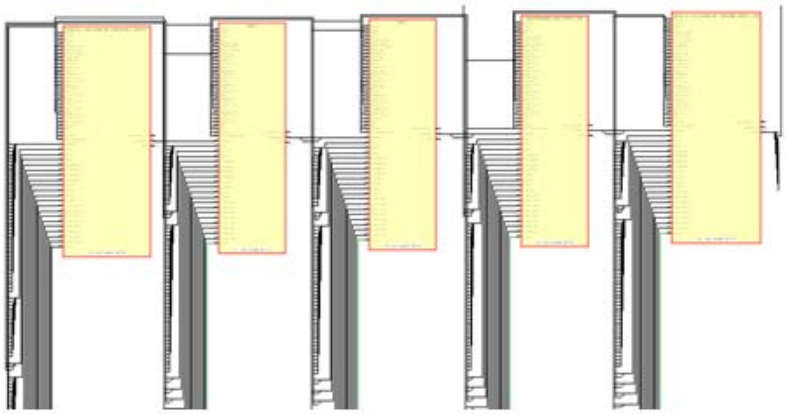
Resource Usage Report

The synthesis tool infers five multAdd blocks with 32 SLE's.

Mapping :
Cell usage:
CLKINT 2 uses

Sequential Cells:
SLE 32 uses

DSP Blocks: 5
MACC: 5 MultAdds



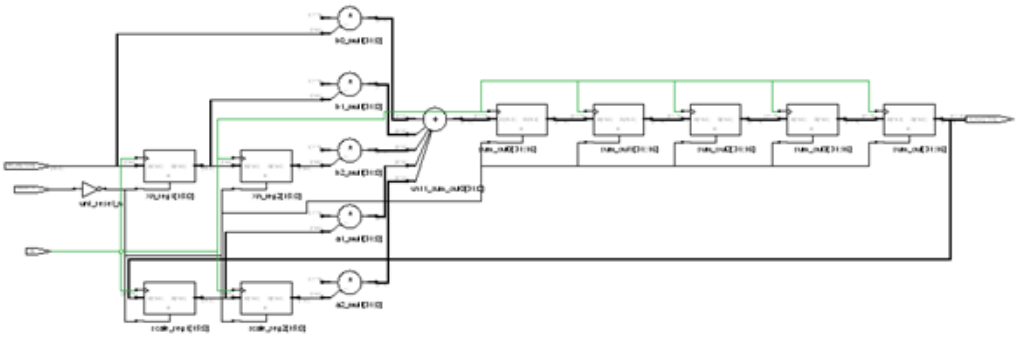
Example 23: MultAdd with Pipelined Registers

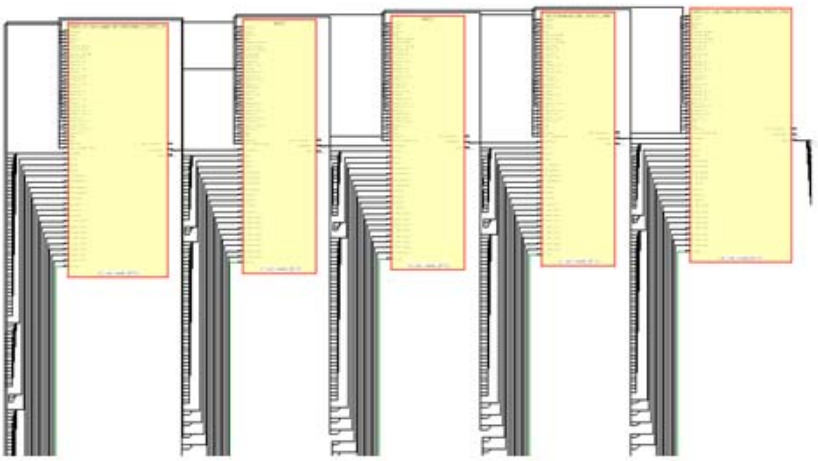
This example has the same functionality as Example 22, but the coding style is changed to pipelined registers. With pipeline registers, the synthesis tool does pipeline register retiming and then inserts registers at the input side to improve timing. The timing performance is improved to ~400MHz. The tool also infers five MACC blocks in the cascade chain.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity TinyIIR_SF2_v6 is
   port(clk                : in std_logic;
      reset_n            : in std_logic;
      Xn_in             : in signed(15 downto 0);
      Yn_out             : out signed(15 downto 0)
      );
end TinyIIR_SF2_v6;

architecture DEF_ARCH of TinyIIR_SF2_v6 is

   constant    b0_coeff        : signed(15 downto 0) := x"7FFF";
   constant    b1_coeff        : signed(15 downto 0) := x"7FFF";
   constant    b2_coeff        : signed(15 downto 0) := x"7FFF";
   constant    a1_coeff        : signed(15 downto 0) := x"7FFF";
   constant    a2_coeff        : signed(15 downto 0) := x"7FFF";
   constant    scale_factor    : signed(15 downto 0) := x"FF3F";

   signal Xn_reg1            : signed(15 downto 0);
   signal Xn_reg2            : signed(15 downto 0);
--    signal Xn_reg3            : signed(15 downto 0);
--    signal Yn_reg1            : signed(15 downto 0);
--    signal Yn_reg2            : signed(15 downto 0);
   signal b0_mult             : signed(31 downto 0);
   signal b1_mult             : signed(31 downto 0);
   signal b2_mult             : signed(31 downto 0);
   signal a1_mult             : signed(31 downto 0);
   signal a2_mult             : signed(31 downto 0);
   signal pad_b0_mult         : signed(11 downto 0);
   signal pad_b1_mult         : signed(11 downto 0);
   signal pad_b2_mult         : signed(11 downto 0);
   signal pad_a1_mult         : signed(11 downto 0);
   signal pad_a2_mult         : signed(11 downto 0);
--    signal scale_reg          : signed(31 downto 0);
   signal scale_reg1       : signed(15 downto 0);
   signal scale_reg2       : signed(15 downto 0);
   signal sum_out             : signed(43 downto 0);
   signal sum_out0           : signed(43 downto 0);
   signal sum_out1           : signed(43 downto 0);
   signal sum_out2           : signed(43 downto 0);
   signal sum_out3           : signed(43 downto 0);

begin
```

```
process(clk, reset_n)
begin
   if (reset_n = '0') then
         Xn_reg1       <= (others => '0');
         Xn_reg2       <= (others => '0');
--            Xn_reg3       <= (others => '0');
         scale_reg1    <= (others => '0');
         scale_reg2    <= (others => '0');
         sum_out        <= (others => '0');
         sum_out0        <= (others => '0');
         sum_out1        <= (others => '0');
         sum_out2        <= (others => '0');
         sum_out3        <= (others => '0');

        elsif rising_edge(clk) then
            Xn_reg1        <= Xn_in;
            Xn_reg2        <= Xn_reg1;
--            Xn_reg3        <= Xn_reg2;
            scale_reg1    <= sum_out(31 downto 16);
            scale_reg2    <= scale_reg1;

        -- IIR filter Summation adder
            sum_out0        <= (pad_b0_mult & b0_mult) +
               (pad_b1_mult & b1_mult) +
               (pad_b2_mult & b2_mult) +
               (pad_a1_mult & a1_mult) +
               (pad_a2_mult & a2_mult);


        -- Forces pipelining of summation adder
            sum_out1       <= sum_out0;
            sum_out2       <= sum_out1;
            sum_out3       <= sum_out2;
            sum_out        <= sum_out3;

        end if;
end process;

--IIR filter coefficient multiplies
   b0_mult        <= Xn_in    * b0_coeff;
   b1_mult        <= Xn_reg1    * b1_coeff;
   b2_mult        <= Xn_reg2    * b2_coeff;
   a1_mult        <= scale_reg1 * a1_coeff;
   a2_mult        <= scale_reg2 * a2_coeff;

-- sign extension
   pad_b0_mult     <= (others => b0_mult(31));
   pad_b1_mult     <= (others => b1_mult(31));
   pad_b2_mult     <= (others => b2_mult(31));
   pad_a1_mult     <= (others => a1_mult(31));
   pad_a2_mult     <= (others => a2_mult(31));

    Yn_out    <= sum_out(31 downto 16);

end def_arch;

--
-------------------------------------------------------------------------
```

```
-- end of code
--------------------------------------------------------------------------------
```





Resource Usage Report

Mapping :
Cell usage:
CLKINT 2 uses

Sequential Cells:
SLE 136 uses

DSP Blocks: 5
MACC: 5 MultAdds

Global Clock Buffers: 2

Total LUTs: 0

Inferring MACC block in DOTP mode

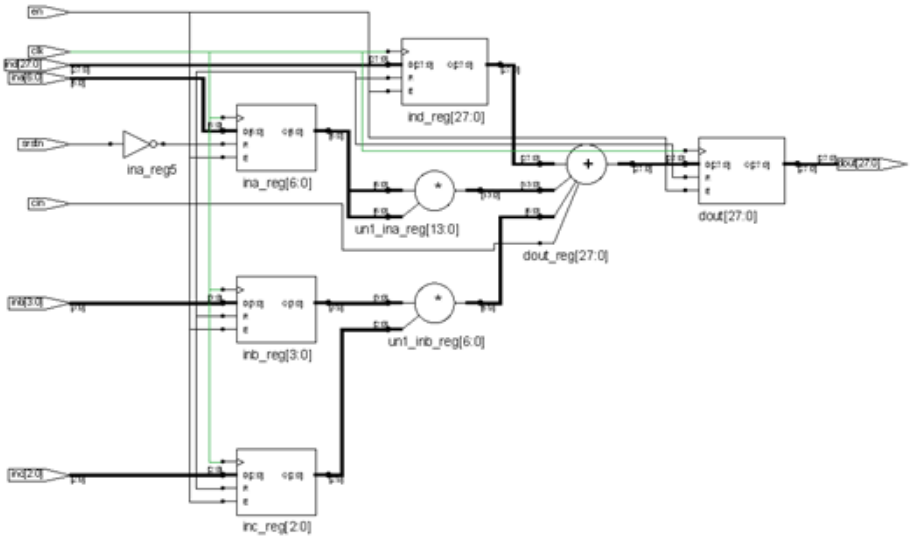
The MACC block when configured in DOTP mode has two independent signed 9x9-bit or unsigned 8x8-bit multipliers followed by addition of these two products. The sum of the dual independent products is stored in the upper 35 bits of the 44-bit register.

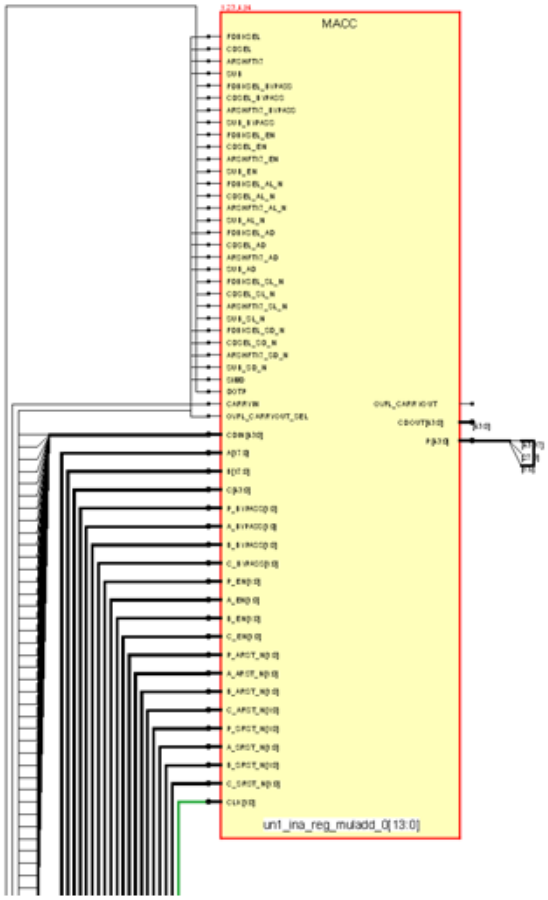
Example 24: Unsigned MultAdd Computation

The RTL is for DOTP computation of sqr(a) + bc + d + cin. All the inputs and outputs are registered with asynchronous active-low resets and active-high enable signals. The synthesis tool infers a single MACC in DOTP mode with MultAdd configuration and packs the adder input registers in SLEs.

```
module dotp_add_ioreg_unsign_srstn_en (clk, srstn, en, ina, inb, inc, ind, cin,
dout);
input    clk, srstn, en;
input cin;
input [6:0] ina;
input [3:0] inb;
input [2:0] inc;
input [27 : 0] ind;
output reg [30:0] dout;

reg [6:0] ina_reg;
reg [3:0] inb_reg;
reg [2:0] inc_reg;
reg [27 : 0] ind_reg;
reg cin_reg;
wire [30:0] dout_reg;

always@(posedge clk) begin
   if (!srstn) begin
      ina_reg    <=    {7{1'b0}};
      inb_reg    <=    {4{1'b0}};
      inc_reg    <=    {3{1'b0}};
      ind_reg    <=    {28{1'b0}};
      cin_reg    <=    1'b0;
      dout    <=    {31{1'b0}};
end else if (en) begin
      ina_reg    <=    ina;
      inb_reg    <=    inb;
      inc_reg    <=    inc;
      ind_reg    <=    ind;
      cin_reg    <=    cin;
      dout    <=    dout_reg;
end else    begin
      ina_reg    <=    ina_reg;
      inb_reg    <=    inb_reg;
      inc_reg    <=    inc_reg;
      ind_reg    <=    ind_reg;
      cin_reg    <=    cin;
```

```
      dout     <=     dout;
   end
end
assign dout_reg = (ina_reg * ina_reg) + (inb_reg * inc_reg) + ind_reg + cin;
endmodule
```





Resource Usage Report

Mapping :
Cell usage:

CLKINT 1 use
CFG2 1 use

Sequential Cells:
SLE 28 uses

DSP Blocks: 1
MACC: 1 MultAdd

Global Clock Buffers: 1

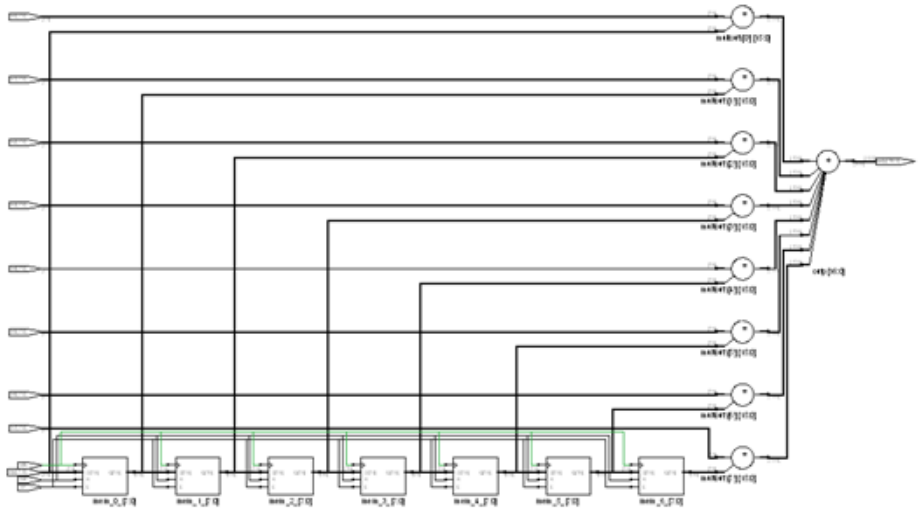
Total LUTs: 1

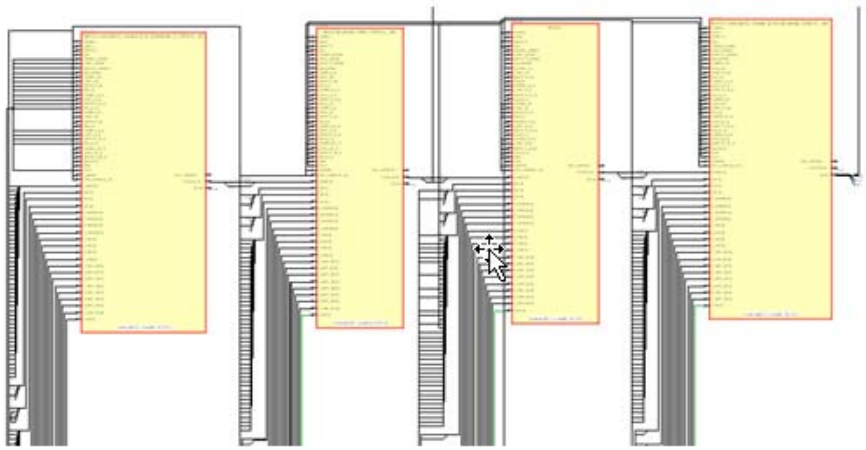
Example 25: Direct-Form 8-tap Finite Impulse Filter

The RTL is for DOTP computation of (ab + bc) +/- d. All the inputs and outputs are registered with asynchronous active-low resets and active-high enable signals.

```
module fir_direct_8tap(inp,h0,h1,h2,h3,h4,h5,h6,h7,clk,rst,en,outp);

parameter inpwidth    =    8;
parameter coefwidth    =    8;
parameter multoutwidth    =    (inpwidth    +    coefwidth);
parameter outwidth        =    (inpwidth    +    coefwidth    +    1);
parameter taplen    =    8;

input    [inpwidth-1 : 0]     inp;
input    [coefwidth-1 : 0]    h0, h1, h2, h3, h4, h5, h6, h7;
input     clk, rst,    en;
output      [outwidth-1 : 0] outp;
reg    [inpwidth-1    : 0]    mem [0 : taplen-2];
wire    [multoutwidth-1 : 0]    multout[0 :    taplen-1];
wire    [outwidth-1 : 0]    addout[0 : taplen-2];
wire    [coefwidth-1 :    0]    coef[0 : taplen-1];
integer i;
assign    coef[0]    =    h0;
assign    coef[1] =     h1;
assign     coef[2] =     h2;
assign     coef[3] =     h3;
assign     coef[4] =     h4;
assign     coef[5] =     h5;
assign     coef[6] =     h6;
assign     coef[7] =     h7;
always @(posedge clk)      begin
   if (rst)    begin
      for (i=0; i<=(taplen-2); i=i+1)    begin
      mem[i]<=0;
      end
   end
else if (en)    begin
      mem[0]<=inp;
      for(i=1;i<=(taplen-2);i=i+1)    begin
         mem[i] <= mem[i-1];
         end
      end
end
assign multout[0]=coef[0]*inp;
generate
   genvar i2;
   for (i2=1;i2<=taplen-1;i2=i2+1)
      begin: mult
      assign multout[i2] = coef[i2]* mem[i2-1];
      end
endgenerate
   assign addout[0]=multout[taplen-1]+multout[taplen-2];
generate
   genvar i3;
      for (i3=0;i3<=(taplen-3);i3=i3+1)
```

```
begin: adding
assign addout[i3+1] = addout[i3]+ multout[(taplen-3)-i3];
end
endgenerate
assign outp=addout[taplen-2];// final addout

endmodule
```





Resource Usage Report

Mapping :
Cell usage:
CLKINT 1 use
CFG1 1 use
CFG2 1 use

Sequential Cells:
SLE 48 uses

DSP Blocks: 4
MACC: 4 MultAdds

Global Clock Buffers: 1

Total LUTs: 2

Note: This example can be complied in P&R only for SmartFusion2 and IGLOO2 family. It is not supported for RTG4 device, since the tool infers MACC block instead of MACC_RT for RTG4 technology.

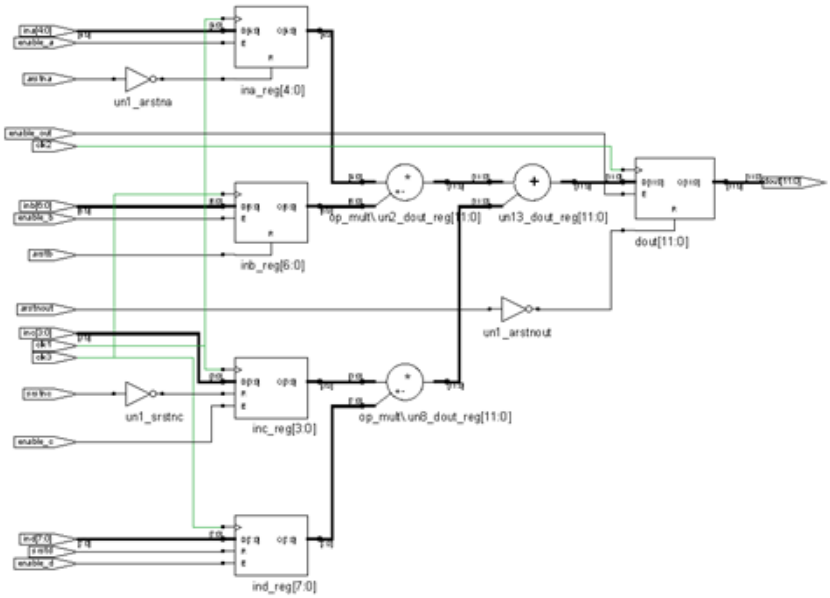
Example 26: DOTP with multiple clocks

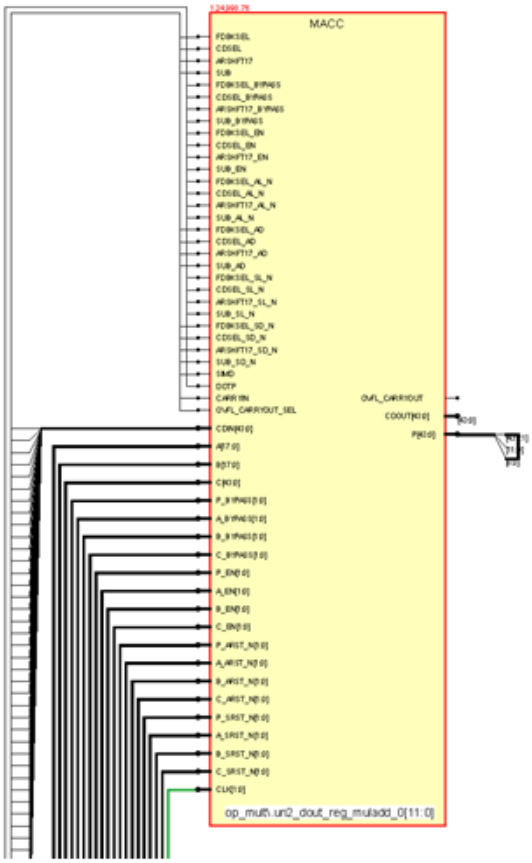
The RTL is for DOTP computation of (ab + cd). The A and B inputs are registered with asynchronous active-low resets and active-high enable signals, the C and D inputs are registered with synchronous active-low resets and synchronous active-high resets but with active-high enable signals. The output is registered with asynchronous active-low reset and active-high enable signals. The clocks are the same for A and C inputs. B and D inputs and the output have their corresponding clocks.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity dotp_ioreg_multiple_syn is
generic    (    widtha        :    integer    :=    5;
         widthb        :    integer    :=    7;
         widthc        :    integer    :=    4;
         widthd        :    integer    :=    8;
         width_out    :    integer    :=    12);
port    (    clk1        :    in std_logic;
            clk2        :    in std_logic;
            clk3        :    in std_logic;
            arstna        :    in std_logic;
            arstb        :    in std_logic;
            srstnc        :    in std_logic;
            srstd        :    in std_logic;
            arstnout    :    in std_logic;
            enable_a    :    in std_logic;
            enable_b    :    in std_logic;
            enable_c    :    in std_logic;
            enable_d    :    in std_logic;
            enable_out    :    in std_logic;
            ina            :    in std_logic_vector(widtha-1 downto 0);
```

```
            inb            :    in std_logic_vector(widthb-1 downto 0);
            inc            :    in std_logic_vector(widthc-1 downto 0);
            ind            :    in std_logic_vector(widthd-1 downto 0);
            dout        :    out std_logic_vector(width_out-1 downto 0));
end dotp_ioreg_multiple_syn;

architecture arch of dotp_ioreg_multiple_syn is

signal     ina_reg    :    std_logic_vector(widtha-1 downto 0);
signal     inb_reg    :    std_logic_vector(widthb-1 downto 0);
signal     inc_reg    :    std_logic_vector(widthc-1 downto 0);
signal     ind_reg    :    std_logic_vector(widthd-1 downto 0);
signal dout_reg        :    std_logic_vector(width_out-1 downto 0);

begin
   process(clk1, arstna)    begin
   if arstna = '0' then
      ina_reg    <=    (others => '0');
   elsif (clk1'event and clk1 = '1') then
         if enable_a = '1' then
            ina_reg    <=    ina;
      end if;
   end if;
end process;

process(clk3, arstb)    begin
   if arstb = '1' then
      inb_reg    <=    (others => '0');
elsif (clk3'event and clk3 = '1') then
         if enable_b = '1' then
            inb_reg    <=    inb;
      end if;
   end if;
end process;

    process(clk1)    begin
   if (clk1'event and clk1 = '1') then
      if    srstnc = '0' then
      inc_reg    <=    (others => '0');
   elsif enable_c = '1' then
      inc_reg    <=    inc;
   end if;
end if;
end process;

process(clk3)    begin
   if (clk3'event and clk3 = '1')    then
         if    srstd = '1' then
               ind_reg    <=    (others => '0');
         elsif enable_d = '1' then
               ind_reg    <=    ind;
         end if;
      end if;
end process;

    dout_reg    <=    (ina_reg * inb_reg) + (inc_reg * ind_reg);
```

```
process(clk2, arstnout)    begin
   if arstnout = '0' then
      dout     <=     (others => '0');
   elsif (clk2'event and clk2 = '1') then
         if enable_out = '1' then
               dout     <=     dout_reg;
            end if;
      end if;
end process;

end arch;
```





Resource Usage Report

Mapping :

Cell usage:
CLKINT 3 uses
CFG1 2 uses
CFG2 2 uses

Sequential Cells:
SLE 24 uses

DSP Blocks: 1
MACC: 1 MultAdd

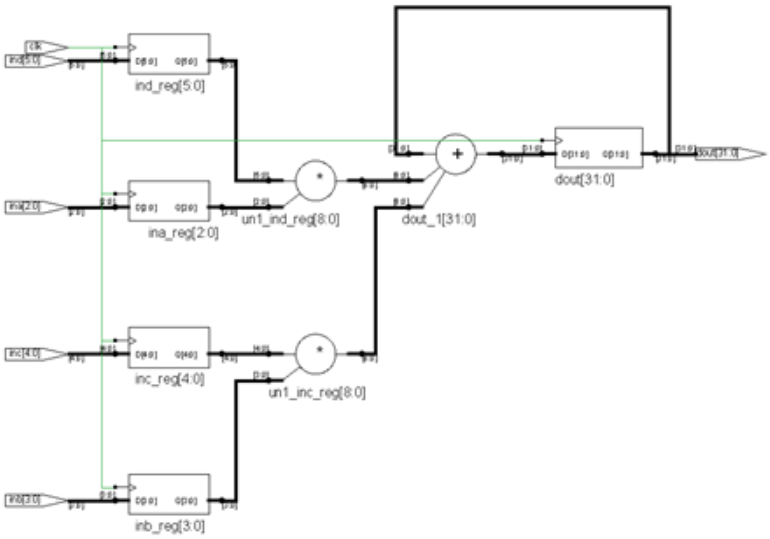
Global Clock Buffers: 3

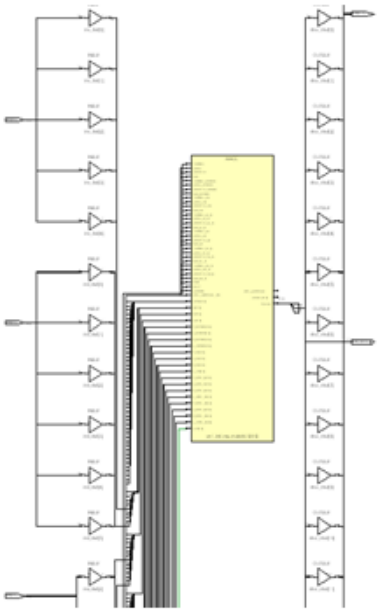
Total LUTs: 4

Example 27: DOTP with MultACC

The RTL below is for MultACC. After synthesis, MACC is inferred in DOTP mode.

```
module dotp_acc_unsign_rtl (clk, ina, inb, inc, ind, dout);

parameter widtha = 3;
parameter widthb = 4;
parameter widthc = 5;
parameter widthd = 6;
parameter width_out = 32;

input clk;
input [widtha-1 : 0] ina;
input [widthb-1 : 0] inb;
input [widthc-1 : 0] inc;
input [widthd-1 : 0] ind;
output reg [width_out-1 : 0] dout;

reg [widtha-1 : 0] ina_reg;
reg [widthb-1 : 0] inb_reg;
reg [widthc-1 : 0] inc_reg;
reg [widthd-1 : 0] ind_reg;

wire [width_out-1 : 0] prod;

always @ (posedge clk)    begin
   ina_reg    <=    ina;
   inb_reg    <=    inb;
   inc_reg    <=    inc;
   ind_reg    <=    ind;
   dout    <=    prod + dout;
end

assign prod        =    (ina_reg * ind_reg) + (inb_reg * inc_reg);

endmodule
```





Resource Usage Report

Mapping :
Cell usage:
CLKINT 1 use

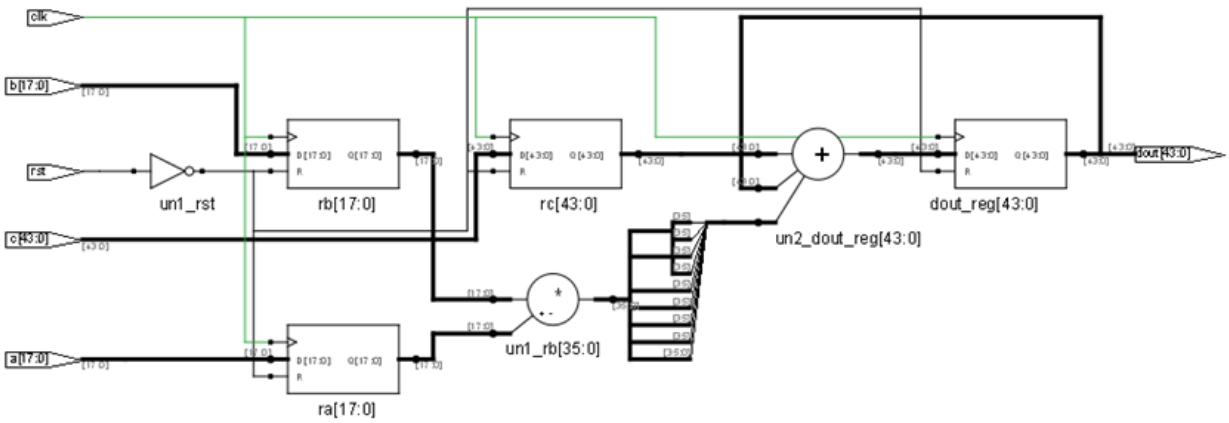
Sequential Cells:
SLE 0 uses

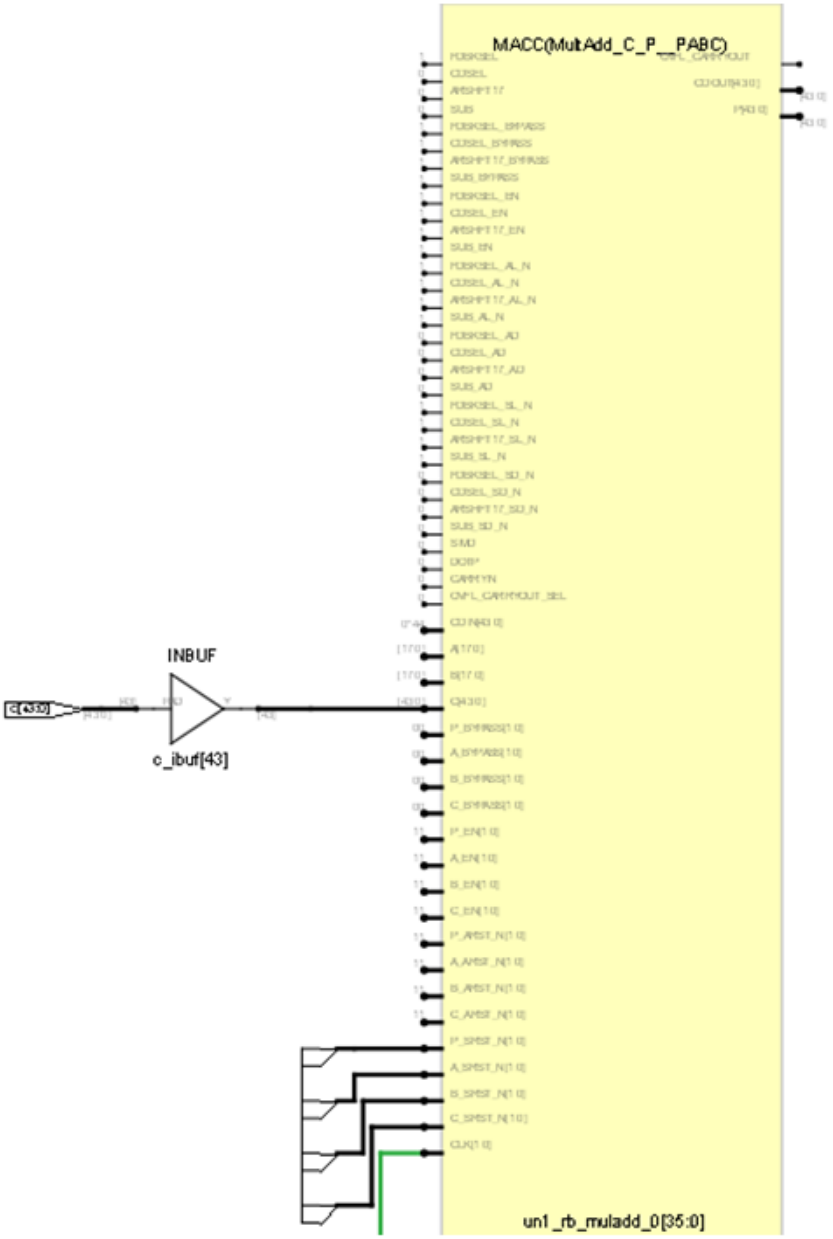
```
DSP Blocks:    1
MACC:        1 MultAcc

I/O ports: 51
I/O primitives: 51
INBUF        19 uses
OUTBUF       32 uses

Global Clock Buffers: 1

Total LUTs:    0
```

Example 28: MultAcc with C input

The RTL below is for MultAcc with C input. After synthesis, MACC is inferred with C input packing.

```
`ifdef synthesis
module test (clk, rst, a, b, c, dout);
`else
module test_rtl (clk, rst, a, b, c, dout);
`endif

input clk, rst;
input  signed [17:0] a, b;
input  signed [43:0] c;
output signed [43:0] dout;

reg signed [17:0] ra, rb;
reg signed [43:0] dout_reg, rc;

always @(posedge clk)
begin
   if (~rst) begin
         dout_reg <= 44'b0;
         ra       <= 17'b0;
      rb       <= 17'b0;
         rc       <= 44'b0;
      end
      else begin
         ra       <= a;
      rb       <= b;
         rc       <= c;
         dout_reg <= dout_reg + rc + ra * rb;
      end
   end

assign dout = dout_reg;

endmodule
```





Resource Usage Report

Mapping:
Cell usage:
CLKINT 1 use

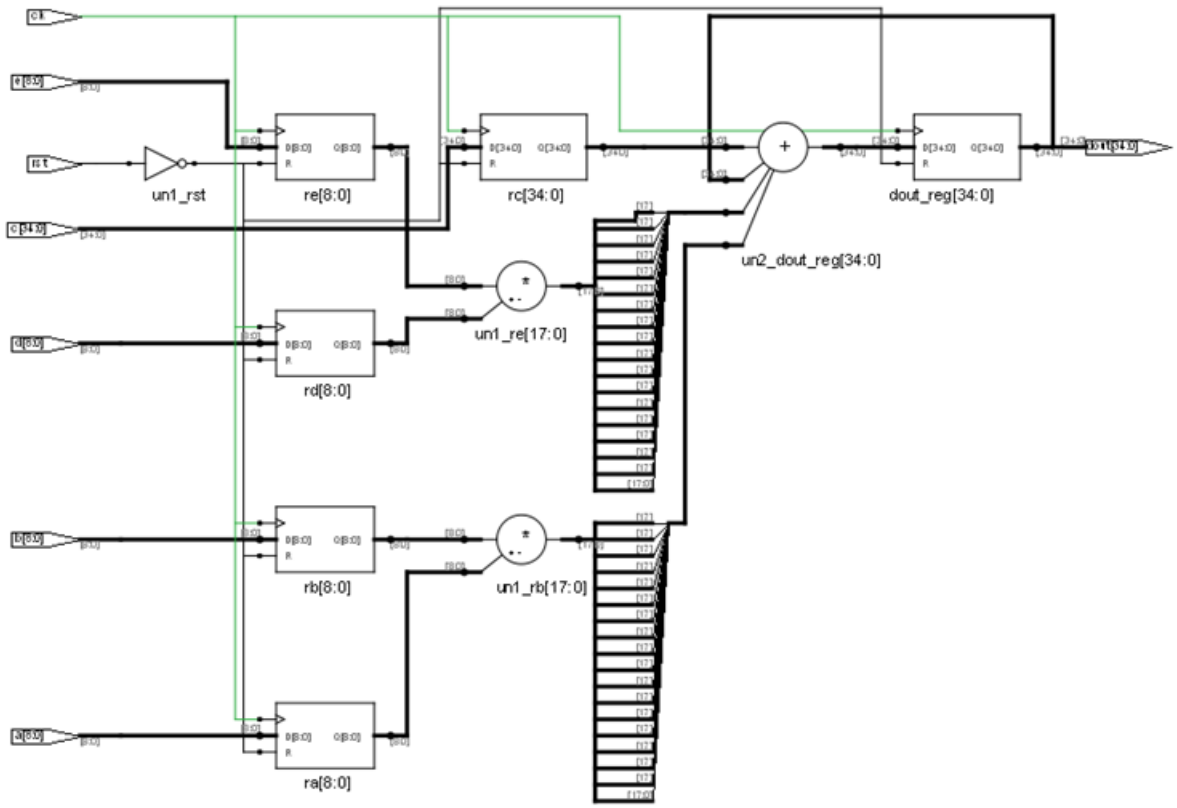
Sequential Cells:
SLE 0 uses

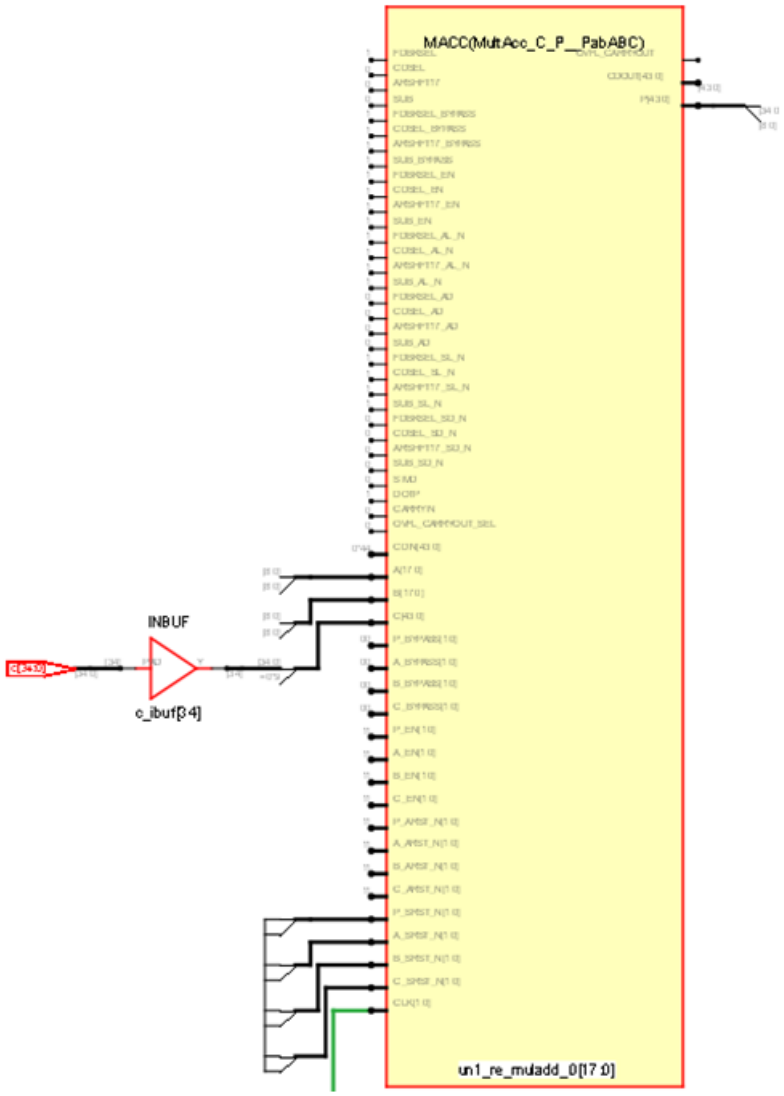
DSP Blocks: 1
MACC: 1 MultAdd

Example 29: DOTP MultAcc with C input

The RTL below is for MultAcc with C input. After synthesis, MACC is inferred in DOTP mode with C input packing.

```
module test (clk, rst, a, b, c, d, e, dout);
input clk, rst;
input  signed [8:0] a, b, d, e;
input  signed [34:0] c;
output signed [34:0] dout;

reg signed [8:0] ra, rb, rd, re;
reg signed [34:0] dout_reg, rc;

always @(posedge clk)
begin
   if (~rst) begin
         dout_reg <= 35'b0;
         ra       <= 9'b0;
      rb       <= 9'b0;
      rc       <= 35'b0;
      rd       <= 9'b0;
         re       <= 9'b0;
      end
      else begin
         ra       <= a;
      rb       <= b;
         rc       <= c;
         rd       <= d;
      re       <= e;
         dout_reg <= dout_reg + rc + (ra * rb) + (rd * re);
      end
   end

assign dout = dout_reg;

endmodule
```





Resource Usage Report

Mapping to part:
Cell usage:
CLKINT 1 use

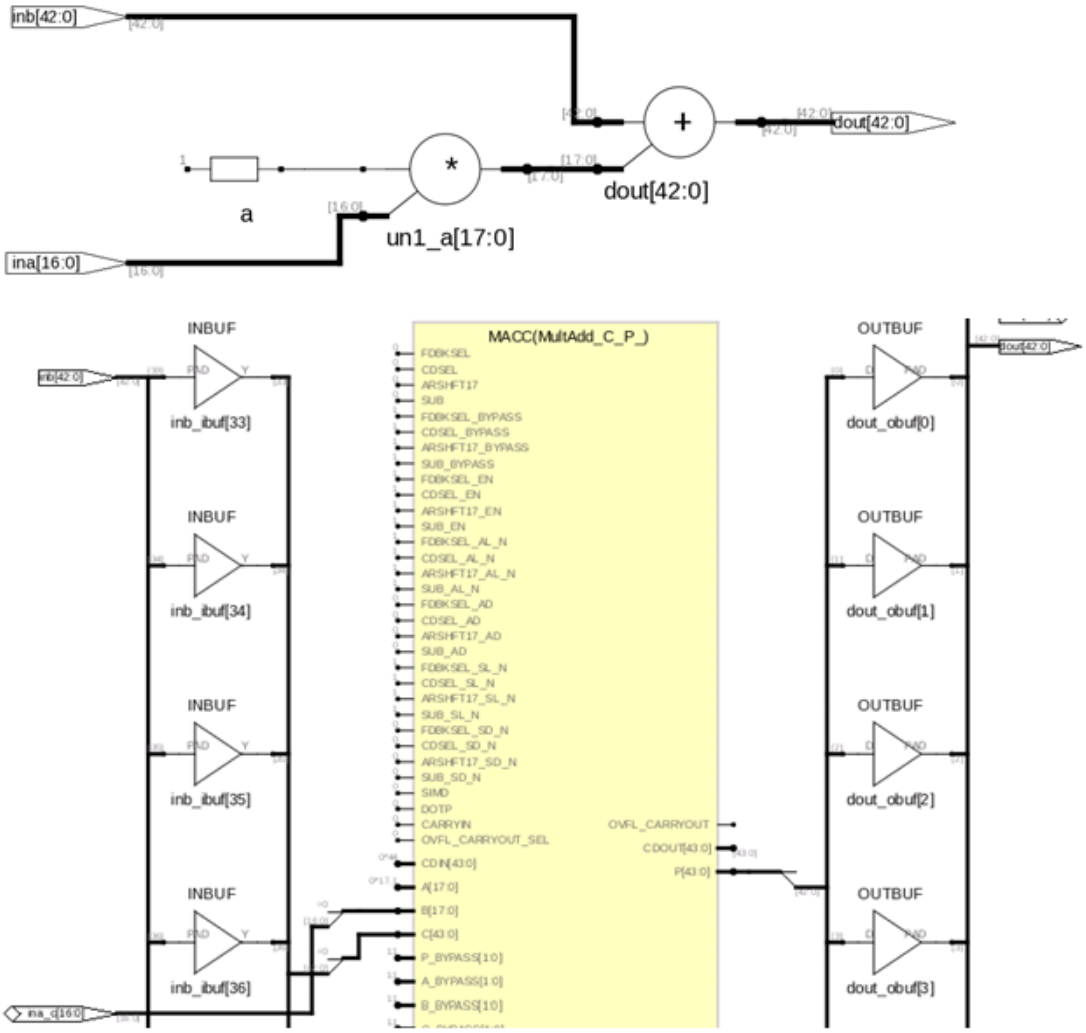
Sequential Cells:
SLE 0 uses

DSP Blocks: 1
MACC: 1 MultACC

Example 30: MultAdd with constant

This example shows the usage of syn_keep and syn_multstyle attributes to infer MACC block for the equation p = inb + (ina * 1).

```
module test ( ina, inb, dout);
parameter widtha = 17;
parameter widthb = 43;
parameter width_out = 43;
input  [widtha-1:0] ina;
input  [widthb-1:0] inb;
output [width_out-1:0] dout;
wire  a  /*synthesis syn_keep=1*/;
assign a = 1'b1;
wire [widthb-1:0] temp  /* synthesis syn_multstyle = dsp */;
assign temp = (a * ina) ;
assign dout = temp + inb;
endmodule
```



Resource Usage Report for MultAdd

This section of the log file (srr) shows resource usage details. It shows that the MultAdd code was implemented in one MACC multiplier block.

Mapping to part: M2S150T

Sequential Cells:
SLE 0 uses

DSP Blocks:1
MACC1 Mult

Total LUTs:0

Limitations

For successful SmartFusion2, IGLOO2 and RTG4 MACC inference with the synthesis software, it is important that you use a supported coding structure, because there are some limitations to what the synthesis tool infers. See *Coding Style Examples*, on page 5 and *Wide Multiplier Coding Examples*, on page 26 for examples of supported structures. Currently, the tool does not support the following:

- Dynamic add/sub support in Dot Product mode
- Overflow and carry-out extraction
- Arithmetic right shift for operand C

  When asserted, the tool performs a 17-bit arithmetic right shift on operand C that goes into the accumulator.

Synopsys, Inc.
690 East Middlefield Road
Mountain View, CA 94043 USA
solvnet.synopsys.com