Inferring Microchip PolarFire Math Block

Synopsys® Application Note, April 2021

The Synopsys® Synplify Pro® synthesis tool automatically infers and implements Microchip® PolarFire Math block. The PolarFire architecture includes dedicated Math block components, which are 18x18-bit signed multiply-accumulate blocks. The blocks can perform DSP-related operations like addition followed by multiplication, multiplication followed by addition, multiplication followed by subtraction, and multiplication with accumulate.

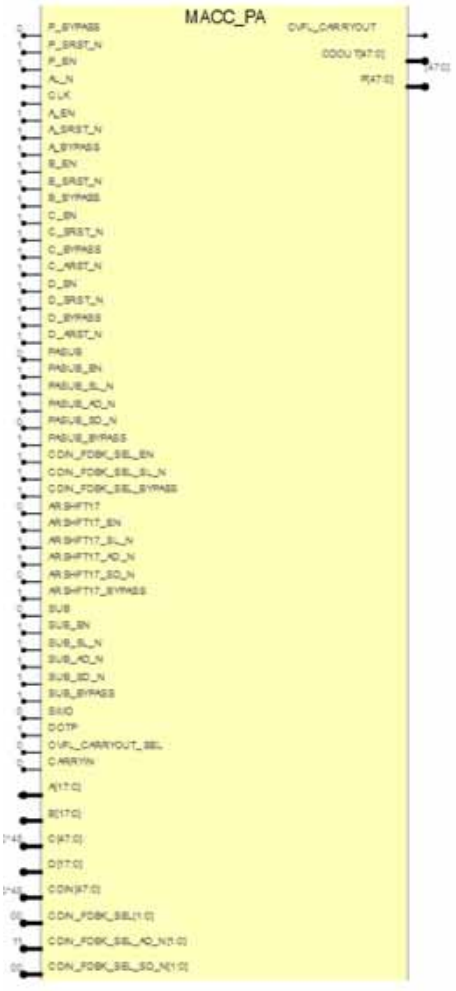
This application note provides a general description of the Microchip PolarFire Math block component and shows you how to infer and implement it with the Synplify Pro software.

The following topics describe the details:

- The PolarFire Math Block, on page 2
- Inferring Math Block, on page 3
- Controlling Inference with the syn_multstyle Attribute, on page 4
- Coding Style Examples, on page 6
- Inferring Math Blocks for Wide Multipliers, on page 22
- Wide Multiplier Coding Examples, on page 26
- Inferring Math Block for Multi-Input Mult-Adds/Mult-Subs, on page 42
- Inferring Math Blocks for Multiplier-AddSub, on page 51
- Inferring Math Blocks for Multiplier-Accumulators, on page 55
- Coding Examples for Timing and QoR Improvement, on page 60
- Inferring Math block in DOTP mode, on page 66
- Inferring Math Block for Pre-Adder, on page 85
- Inferring MACC_PA_BC_ROM Block for Multiplier, on page 103
- Inferring MACC_PA_BC_ROM Block for Mult-Add/Mult-Sub/Mult-Acc, on page 116
- Inferring MACC_PA_BC_ROM Block for Multiplier with Pre-adder, on page 132
- Inferring MACC_PA_BC_ROM Block in DOTP mode, on page 146
- Inferring MACC_PA_BC_ROM Block for Cascaded Chain Using BCOUT->B Connection, on page 157
- Inferring MACC_PA Block in SIMD mode, on page 179
- Limitations, on page 184

The PolarFire Math Block

The PolarFire device supports 18x18-bit signed multiplication, multiply-add, and multiply-accumulate Math blocks. The multiplier takes two 18-bit signed signals and multiplies them for a 36-bit result. The result is then extended to 48 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 Math block. The MACC_PA macro has the capability of MACC macro from SmartFusion2, IGLOO2, and RTG4 with a pre-adder function and extends the accumulator width from 44 bits to 48 bits. The following figure shows the 18x18-bit PolarFire Math block.



All signals of the Math block, except CDIN and CDOUT, have optional registers. All registers must use the same clock. A, B, P, CDIN_FDBK_SEL, PASUB and SUB registers must use the same asynchronous reset, so that these registers get packed into the Math block. For a complete list of all the block options and their configurations, refer to the Microchip documentation.

Inferring Math Block

The Synplify Pro tool can infer Math block components. You can write your RTL so that the synthesis tool recognizes the structures and maps them to Math components. The Synplify Pro tool extracts the following logic structures from the hardware description and maps them to Math blocks—PreaddMult (preadder followed by a multiplier), mults (multiplier), multAdds (multiplier followed by an adder), multSubs (multiplier followed by a subtractor), and multAccs (multiplier-accumulator structures)

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

- The Microchip Math 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 Math blocks, as described in Inferring Math Blocks for Wide Multipliers, on page 22.
- The synthesis tool supports the inference of multiple Math block components across different hierarchies. The multipliers, input registers, output registers, and subtractors/adders are packed into the same Math block, even if they are in different hierarchies.
- The synthesis tool packs registers at the inputs and outputs of preaddMult, mults, multAdds, multSubs, and multAccs into Math blocks.

  By default, the tool maps all multiplier inputs with a width of 3 or greater to Math 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 preadderMult, mults, multAdds, multSubs, and multAccs into Math 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 gets priority and are packed in the Math 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 Math blocks, as described in Inferring Math Blocks for Wide Multipliers, on page 22.
- The synthesis tool packs different kinds of flip-flops at the inputs/outputs of the preaddMults, mults, multAdds, multSubs, and multAccs into Math 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 Math block cascade feature with multi-input multAdds and multSubs, up to a maximum of 18x18 bits for signed multipliers and 17x17 bits for unsigned multipliers. The synthesis tool packs logic into Math blocks efficiently using hard-wired cascade paths to improve the quality of results (QoR) for the design, as described in Inferring Math Block for Multi-Input Mult-Adds/Mult-Subs, on page 42.
- The synthesis tool uses the internal paths for adder feedback loops inside the Math block 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 Math Blocks for Multiplier-Accumulators, on page 55.
- The synthesis tool infers Math block in DOTP mode as described in Inferring Math block in DOTP mode, on page 66.
- The synthesis tool infers Math block with pre-adder as described in Inferring Math Block for Pre-Adder, on page 85.

Controlling Inference with the syn_multstyle Attribute

Use the syn_multstyle attribute to control the inference of multiple Math blocks. The attribute is described briefly here. For detailed information and more examples, see the Reference Manual.

Controlling Default Inference

By default, multipliers with input widths of 3 or greater are packed in the Math block, while smaller input widths are mapped to logic. If the multipliers are inferred as Math 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 Math 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" */;

SIMD mode

SIMD mode of MACC_PA is supported. Two multipliers as below can be packed in a single MACC_PA block.

```
P[17:0] = B[8:0] * A[8:0]
P[47:18] = B[17:9] * A[17:9]
```

You can add the synthesis attribute as below to enable SIMD mode:

```
syn_multstyle = "simd:n"
```

In the above example, n denotes an integer value that identifies a pair of multipliers to pack in SIMD mode.

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|simd:n}
define_attribute {object} syn_multstyle {dsp|logic|simd:n}
```

Coding Style Examples

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

See the following examples of recommended coding styles:

Example 1: 6x6-Bit Unsigned Multiplier, on page 7

Example 2: 11x9-Bit Signed Multiplier, on page 8

Example 3: 18x18-Bit Signed Multiplier with Registered I/Os, on page 10

Example 4: 17x17-Bit Unsigned Multiplier with Different Resets, on page 12

Example 5: Unsigned Mult with Registered I/Os and Different Clocks, on page 14

Example 6: Multiplier-Adder, on page 16

Example 7: Multiplier-Subtractor, on page 18

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

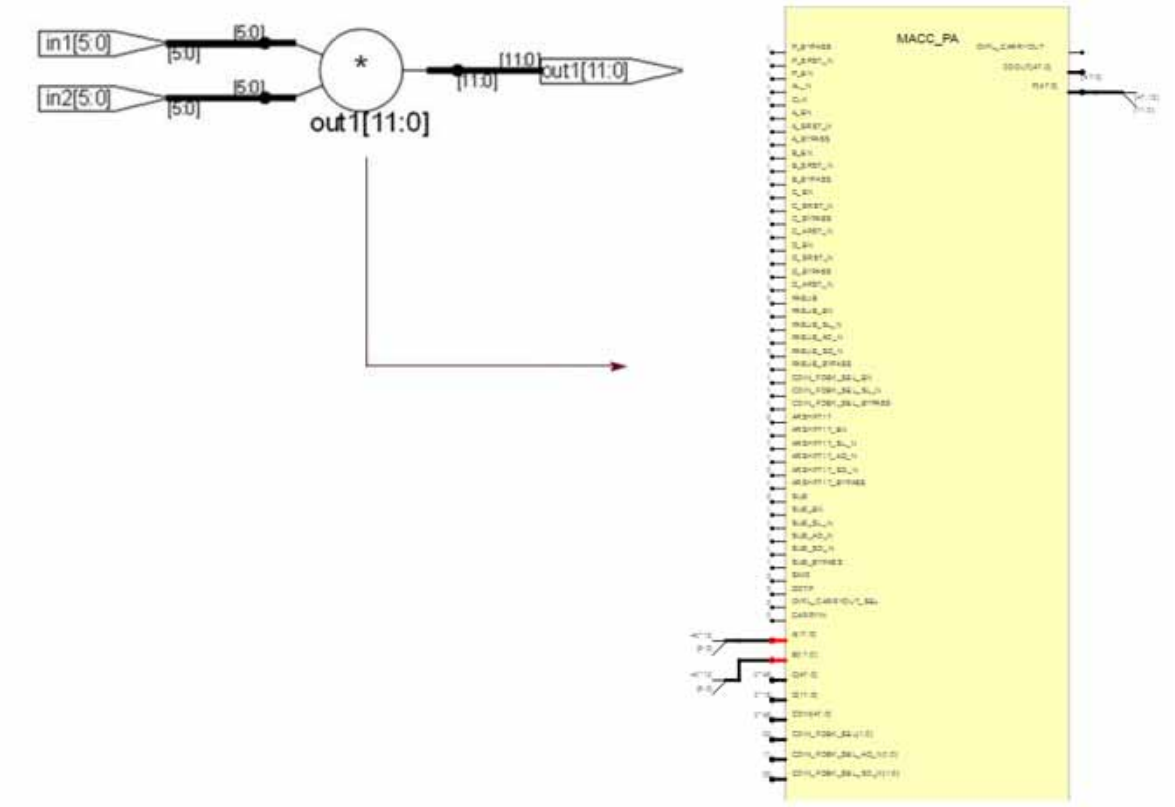
Example 1: 6x6-Bit Unsigned Multiplier

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

RTL

```
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;
```

SRS (RTL) and SRM (Technology) View



Resource Usage

The resource usage details show that the multiplier code was implemented in one Math block.

```
Mapping to part: pa5m300fbga896std

Sequential Cells:
SLE 0 uses

DSP Blocks:1
MACC_PA 1 Mult

Total LUTs:0
```

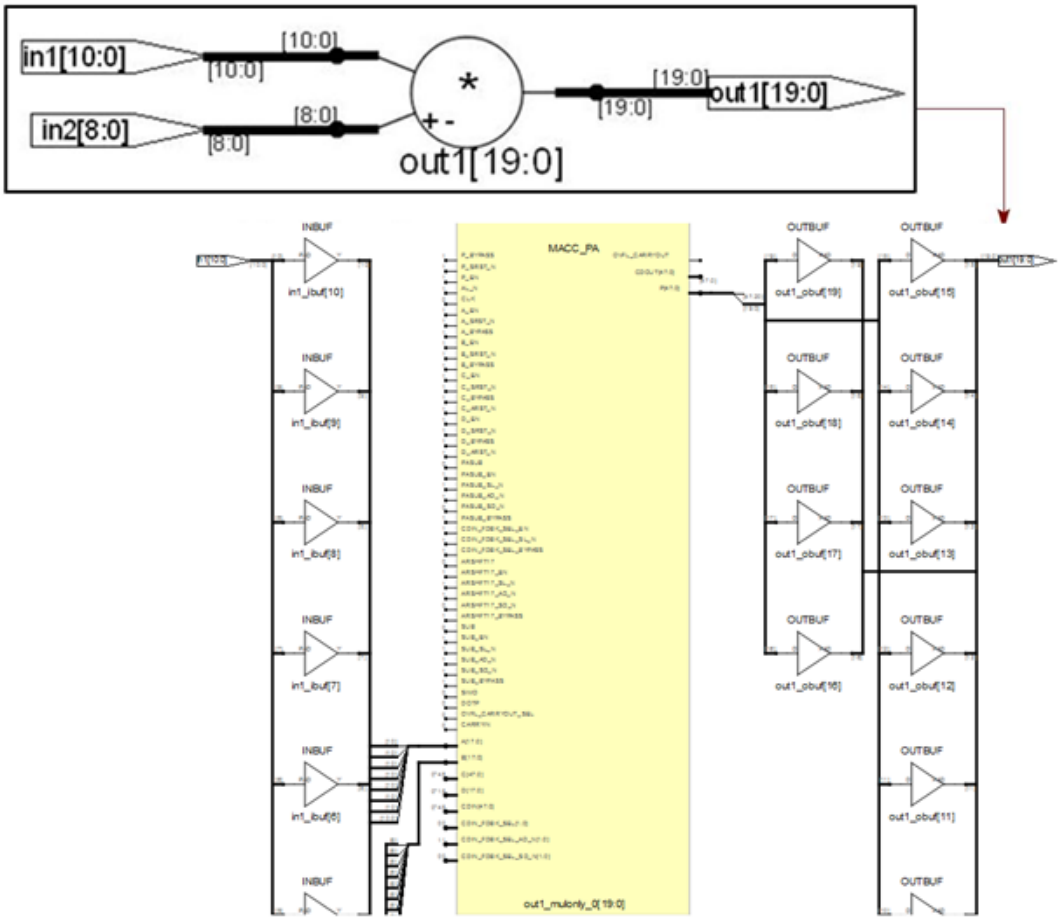
Example 2: 11x9-Bit Signed Multiplier

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

RTL

```
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;
```

SRS (RTL) and SRM (Technology) View



Resource Usage

Mapping to part: pa5m300fbga896std

Sequential Cells:
SLE0 uses

DSP Blocks:1
MACC_PA 1 Mult

Total LUTs:0

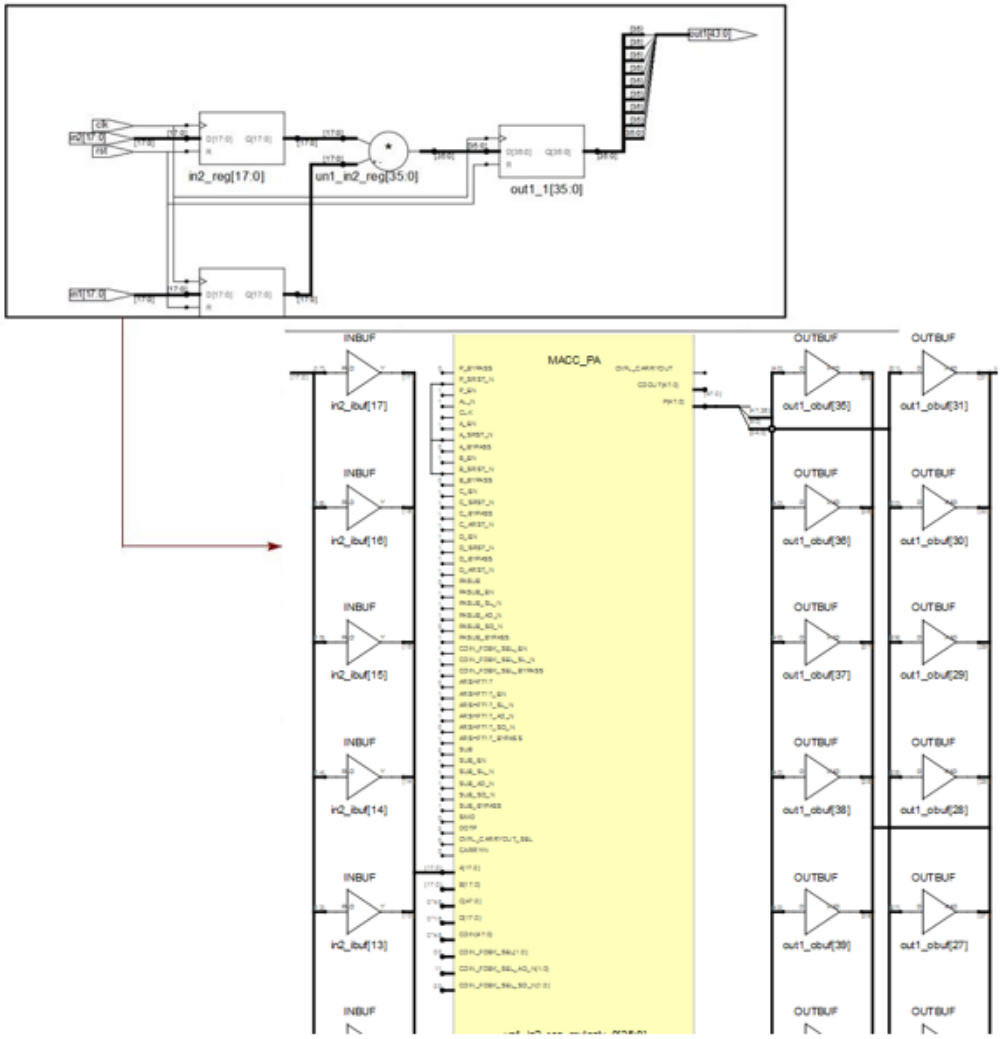
Example 3: 18x18-Bit Signed Multiplier with Registered I/Os

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 Math block, as shown in this example.

RTL

```
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
```

SRS (RTL) and SRM (Technology) View



Resource Usage

Sequential Cells: SLE0 uses

DSP Blocks 1
MACC_PA 1 Mult

Total LUTs: 0

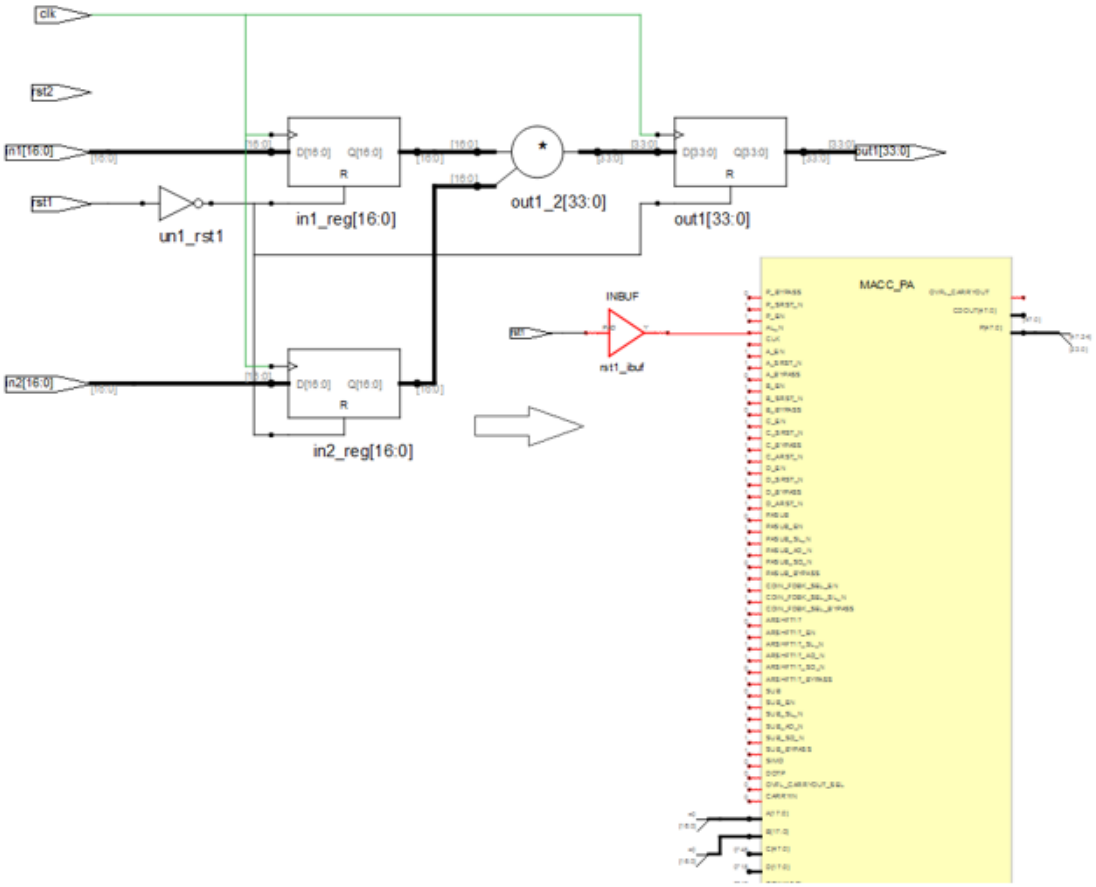
Example 4: 17x17-Bit Unsigned Multiplier with Different Resets

This is the 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 Math block as shown in this example.

RTL

```
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;
```

SRS (RTL) and SRM (Technology) View



Resource Usage

SLE0 uses

DSP Blocks:1
MACC_PA :1 Mult

Global Clock Buffers: 1

Total LUTs:0

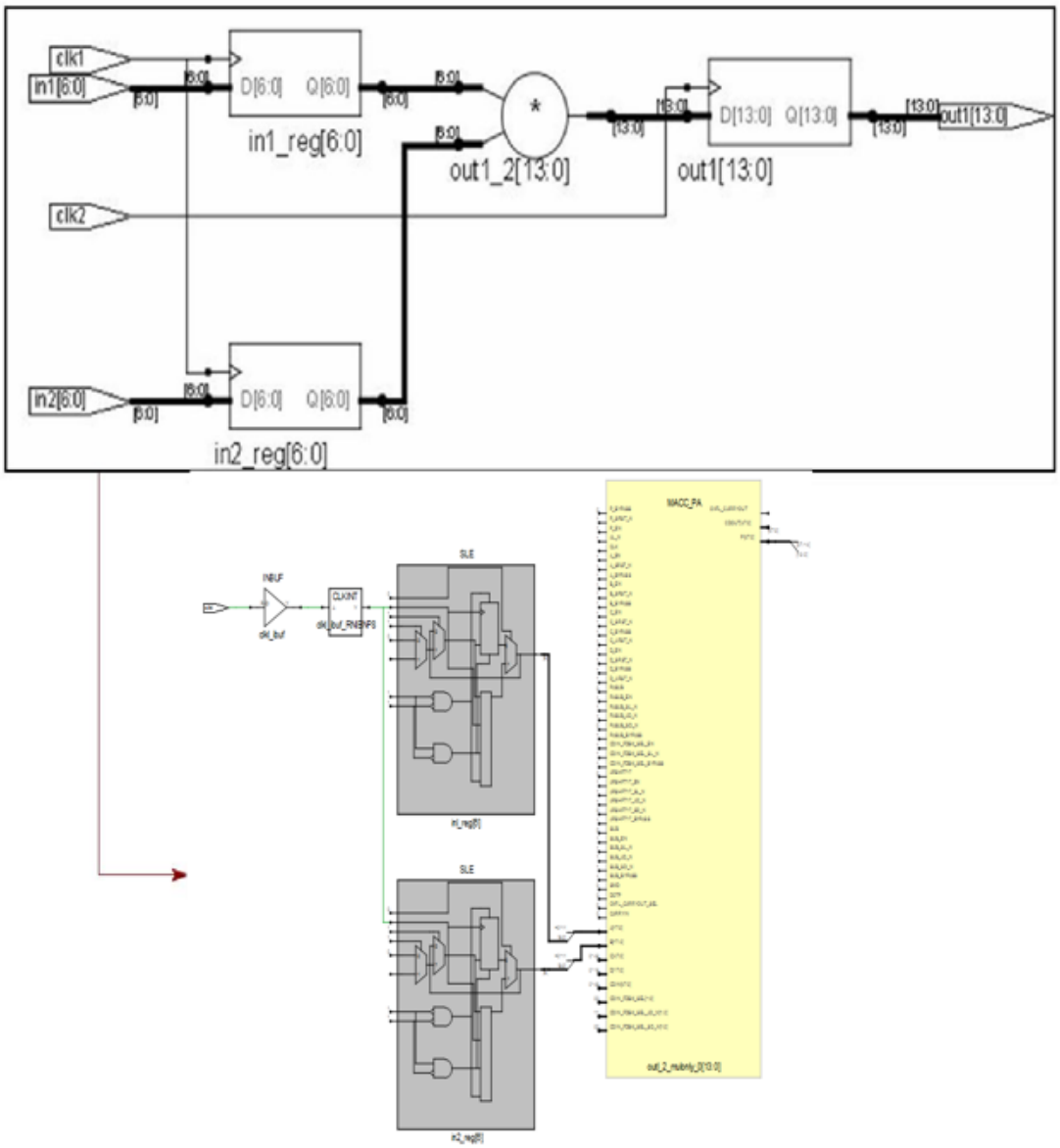
Example 5: Unsigned Mult with Registered I/Os and Different Clocks

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

RTL

```
module unsign_mult ( inl, in2, clkl, clk2, outl );
input [6:0] inl, in2;
input clkl,clk2;
output [13:0] outl;
reg [13:0] outl;
reg [6:0] inl_reg, in2_reg;

always @ ( posedge clkl )
begin
   inl_reg <= inl;
   in2_reg <= in2;
end

always @ ( posedge clk2 )
begin
   outl <= inl_reg * in2_reg; end

endmodule
```

SRS (RTL) and SRM (Technology) View



Resource Usage

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

Mapping to part: pa5m300fbga896std
Cell usage:

Sequential Cells:
SLE14 uses

DSP Blocks:1
MACC_PA :1 Mult

Global Clock Buffers:1

Total LUTs:0

Example 6: Multiplier-Adder

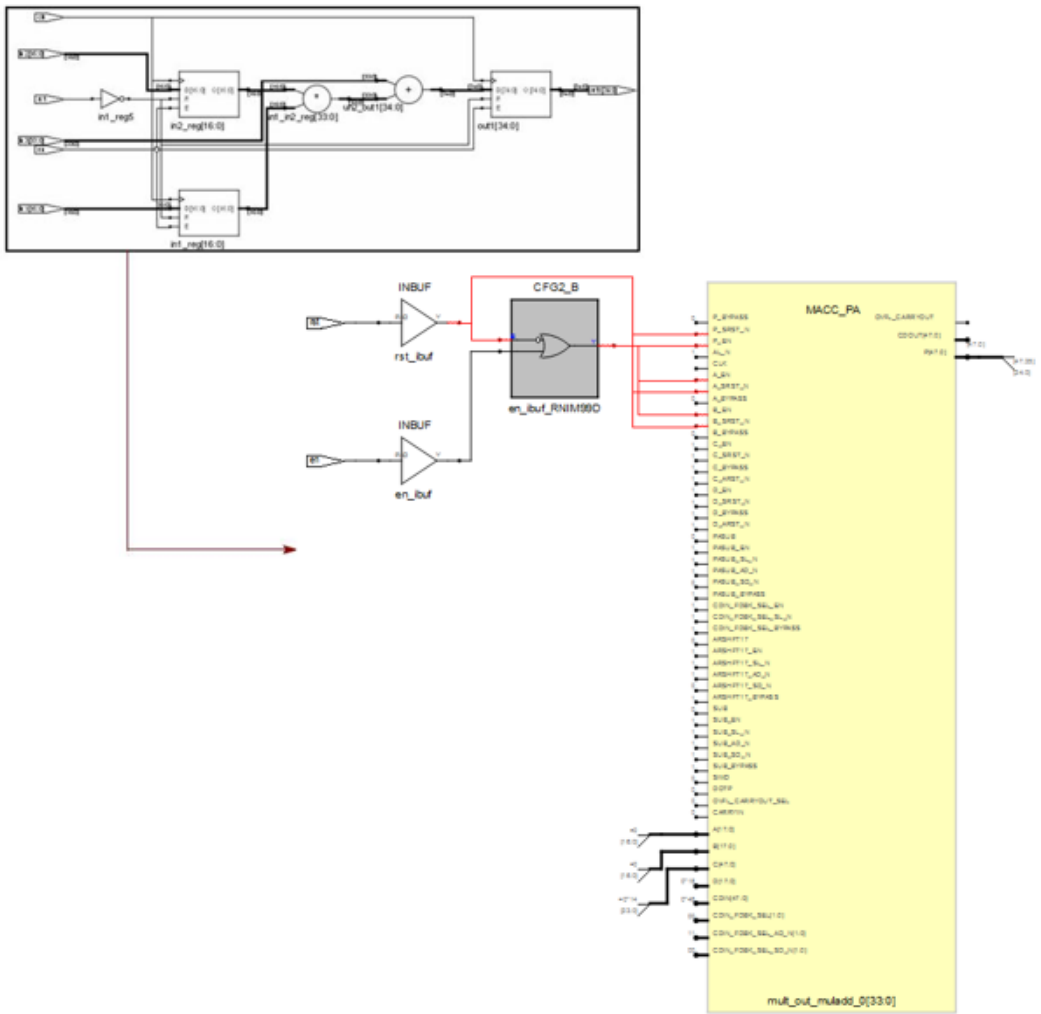
This VHDL example shows how the output of a multiplier is added to another input. The inputs and outputs are registered, and have enables and synchronous resets. The figure in this example shows how the design gets mapped into a Math block.

RTL

```
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;
```

SRS (RTL) and SRM (Technology) View



Resource Usage

Sequential Cells:
SLE0 uses

DSP Blocks:1
MACC_PA :1 MultAdd

Total LUTs: 1

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

See the following examples:

- Unsigned MultSub Verilog Example (P = Cin - Mult), on page 18
- Signed MultSub VHDL Example (P = Cin - Mult), on page 20
- Signed MultSub Verilog Example (P = Mult - Cin), on page 20
- Unsigned MultSub VHDL Example (P = Mult - Cin), on page 21

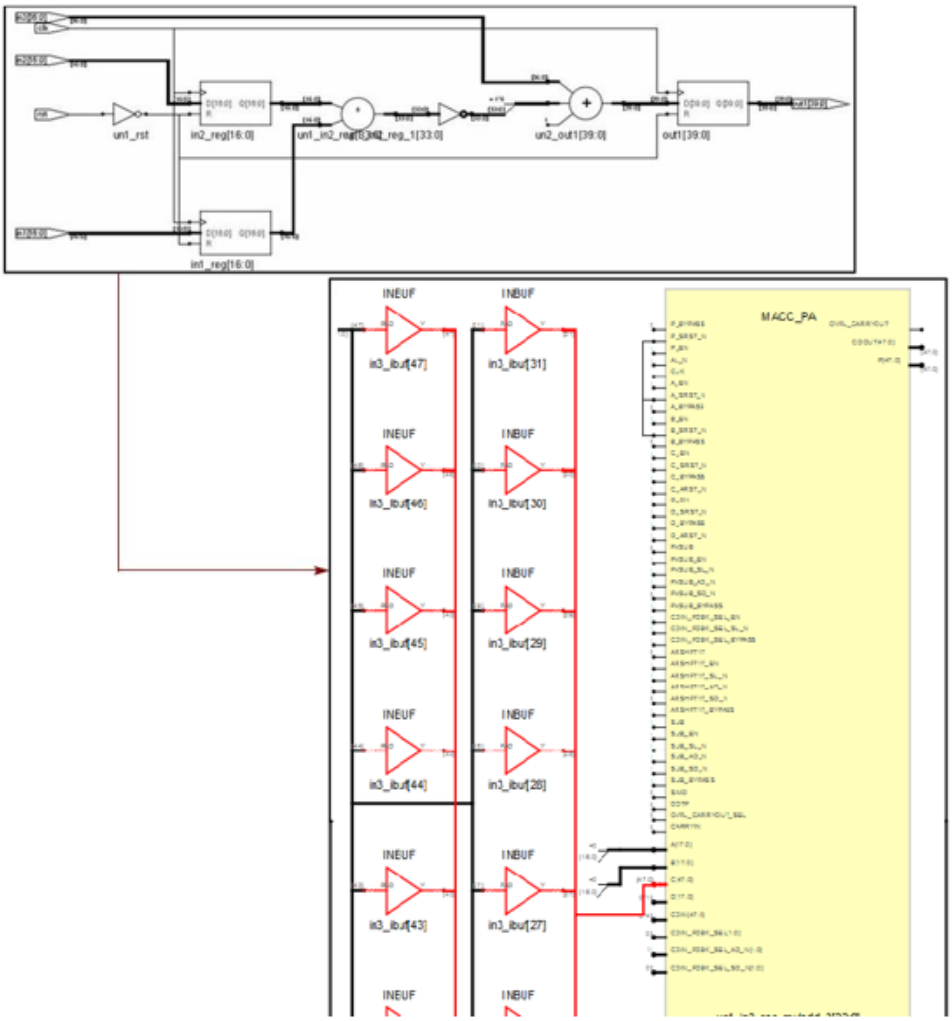
Unsigned MultSub Verilog Example (P = Cin - Mult)

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

RTL

```
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
```

SRS (RTL) and SRM (Technology) View



Resource Usage

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

Mapping to part: pa5m300fbga896std

Sequential Cells:
SLE 0 uses

DSP Blocks:1
MACC_PA : MultSub

Total LUTs: 0

Signed MultSub VHDL Example (P = Cin - Mult)

RTL

```
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

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

Mapping to part: pa5m300fbga896std

Sequential Cells:
SLE0 uses

DSP Blocks:1
MACC_PA :1 MultSub

Total LUTs:0

Signed MultSub Verilog Example (P = Mult - Cin)

RTL

```
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

In this case, the log file shows that multiplier, subtractor and input registers are mapped into the Math block.

Mapping to part: pa5m300fbga896std
Cell usage:

Sequential Cells:
SLE 0 uses

DSP Blocks:1
MACC_PA : MultAdd

Global Clock Buffers:1

Total LUTs: 37

Unsigned MultSub VHDL Example (P = Mult - Cin)

RTL

```
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

The log file shows that only the multiplier is mapped into the Math block. The subtractor is mapped to logic.

Mapping to part: pa5m300fbga896std
Sequential Cells:
SLE0 uses

DSP Blocks:1
MACC_PA :1 MultAdd

Total LUTs:17

Inferring Math 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 Math blocks, using the cascade and shift functions of the Math 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 Math blocks, using the CDOUT and CDIN pins to propagate the cascade output of result P from one Math block to the cascade input for operand Cx to the next Math 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.

PolarFire MACC_PA Block	Maximum Cascaded Size
PA5M300	66 MACC_PA blocks

See the following topics for more details about wide multipliers:

- Fracturing Algorithm, on page 22
- Mapping Fractured Multipliers, on page 23
- Cascade Chain, on page 24
- Log File Message, on page 24
- Pipelined Registers with Wide Multipliers, on page 24

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 Math 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 Math 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 Math 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 Math 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 and 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 Math blocks using cascade and shift functions. See Example 10: Unsigned 26x26-Bit Multiplier (Two Wide Inputs), on page 29 and Example 11: 35x35-Bit Signed Multiplier (Two Wide Inputs), on page 30 for examples.

Cascade Chain

PolarFire PA5M300 devices support a maximum of 66 Math blocks when connected in a cascade chain. After fracturing, if the number of mults, multAdds, or multSubs is more than 66, 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 Math blocks. A cascade chain is created and the tool breaks the chain after connecting 66 Math 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 Math 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:FF150 : test.v(24) | Multiplier out1[203:0] implemented with multiple MACC_PA blocks using cascade/shift feature

Pipelined Registers with Wide Multipliers

The synthesis tool pipelines register at the inputs and outputs of wide multipliers in different hierarchies into multiple Math 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 they 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 Math 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 Math blocks.
- If the output is unregistered and the inputs are registered with different clocks, the input registers are not pipelined in the Math 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 Math 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 Math blocks, as long as all the registers use the same clock. The adder/subtractor and output register are mapped to logic.

See Example 13: 35x35-Bit Signed Mult with 2 Pipelined Register Stages, on page 33.

Wide Multiplier Coding Examples

The following examples show how to code wide multipliers so that they are inferred and mapped to Math blocks, according to the guidelines explained in Inferring Math Blocks for Wide Multipliers, on page 22.

- Example 8: Unsigned 20x17-Bit Multiplier (One Wide Input), on page 26
- Example 9: 21x18-Bit Signed Multiplier (One Wide Input), on page 28
- Example 10: Unsigned 26x26-Bit Multiplier (Two Wide Inputs), on page 29
- Example 11: 35x35-Bit Signed Multiplier (Two Wide Inputs), on page 30
- Example 12: 69x53-Bit Signed Multiplier, on page 31
- Example 13: 35x35-Bit Signed Mult with 2 Pipelined Register Stages, on page 33
- Example 14: FIR 4 Tap Filter, on page 35

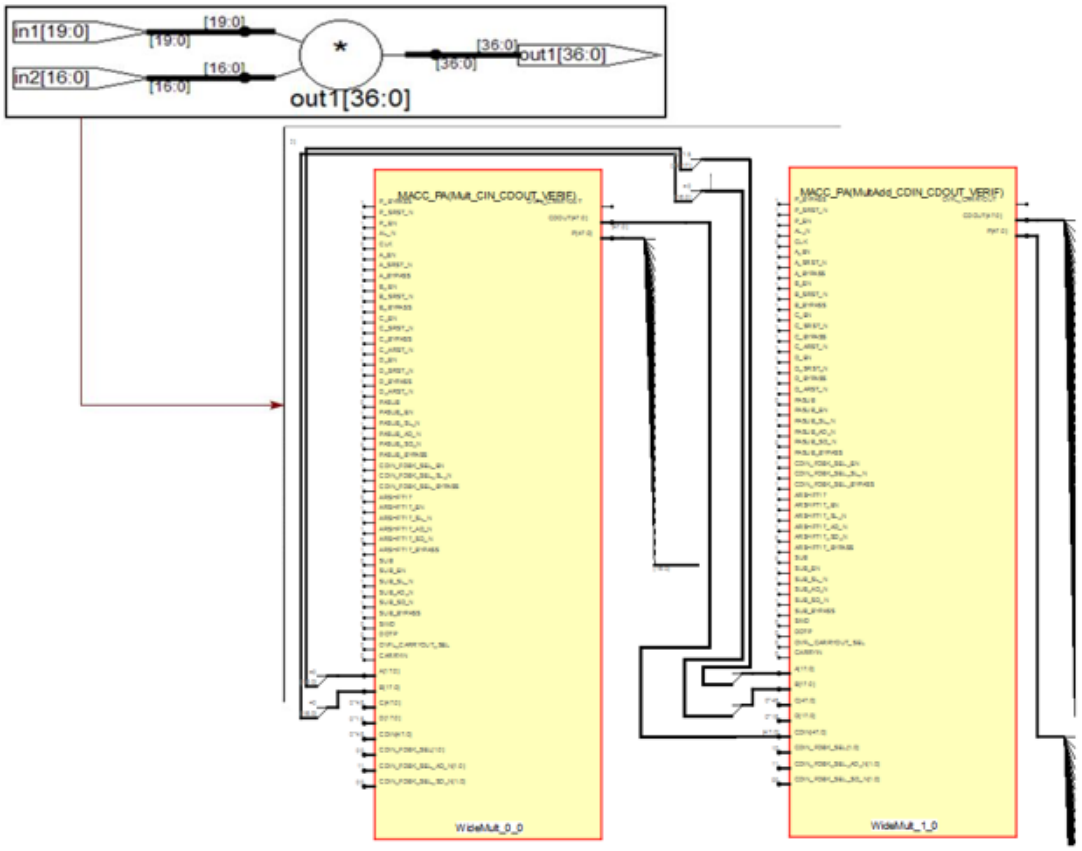
Example 8: Unsigned 20x17-Bit Multiplier (One Wide Input)

This multiplier is split and mapped to two Math blocks.

RTL

```
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;
```

SRS (RTL) and SRM (Technology) View



Resource Usage

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

Mapping to part: pa5m300fbga896std

Sequential Cells:
SLE0 uses

DSP Blocks:2
MACC_PA :1 Mult
MACC_PA :1 MultAdd

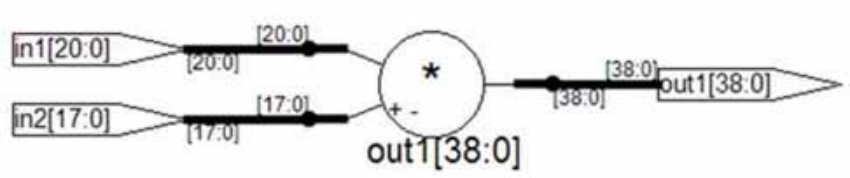
Total LUTs:0

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

RTL

```
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
```

SRS (RTL) View



Resource Usage

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

Mapping to part : pa5m300fbga896std

Sequential Cells:

SLE0 uses

DSP Blocks:2
MACC_PA :1 Mult
MACC_PA :1 MultAdd

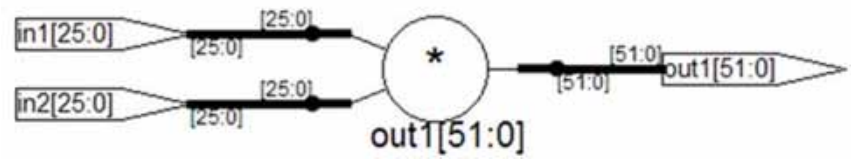
Total LUTs:0

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

RTL

```
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;
```

SRS (RTL) View



Resource Usage

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

Mapping to part: pa5m300fbga896std

Sequential Cells:
SLE0 uses

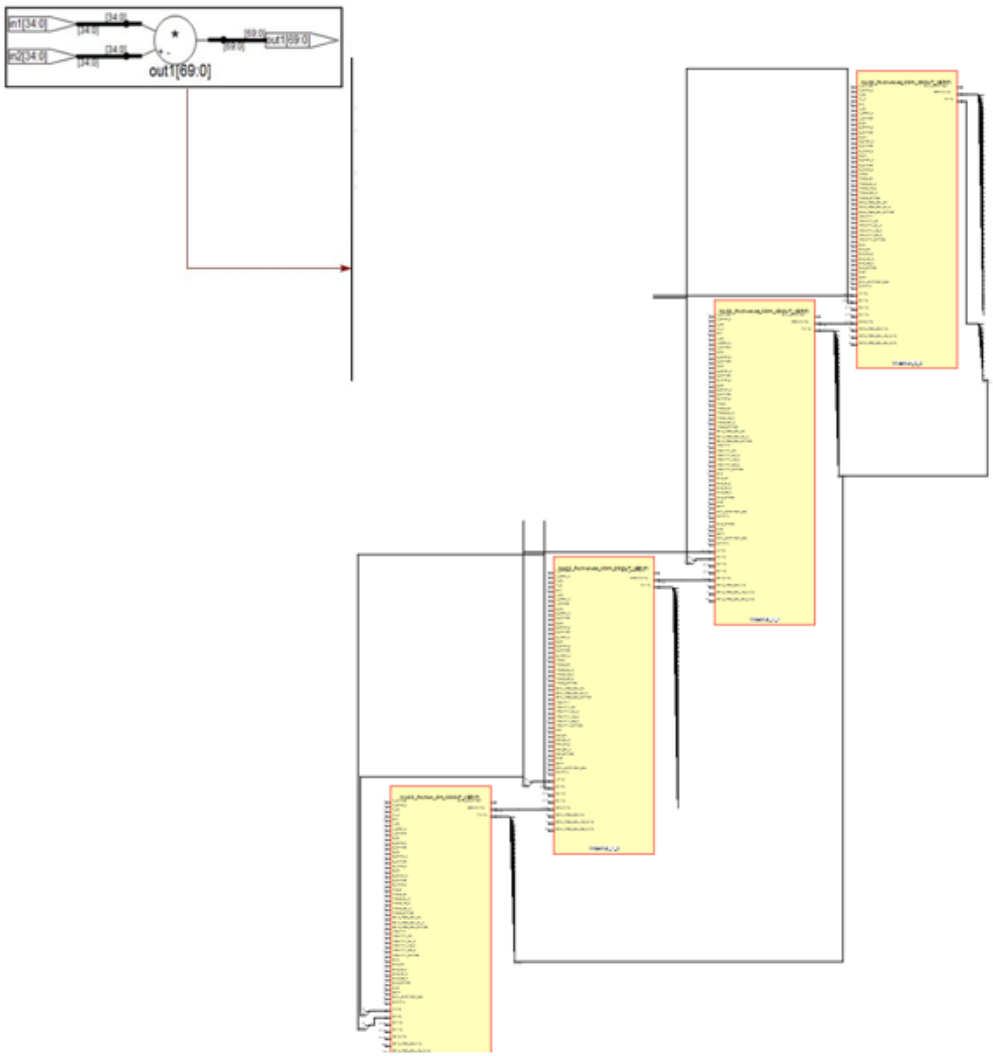
DSP Blocks:4
MACC_PA :1 Mult
MACC_PA :3 MultAdd

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

RTL

```
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
```

SRS (RTL) and SRM (Technology) View



Resource Usage

The synthesis tool infers 1 mult and 3 multAdd blocks.

Mapping to part: pa5m300fbga896std

Sequential Cells:
SLE0 uses

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

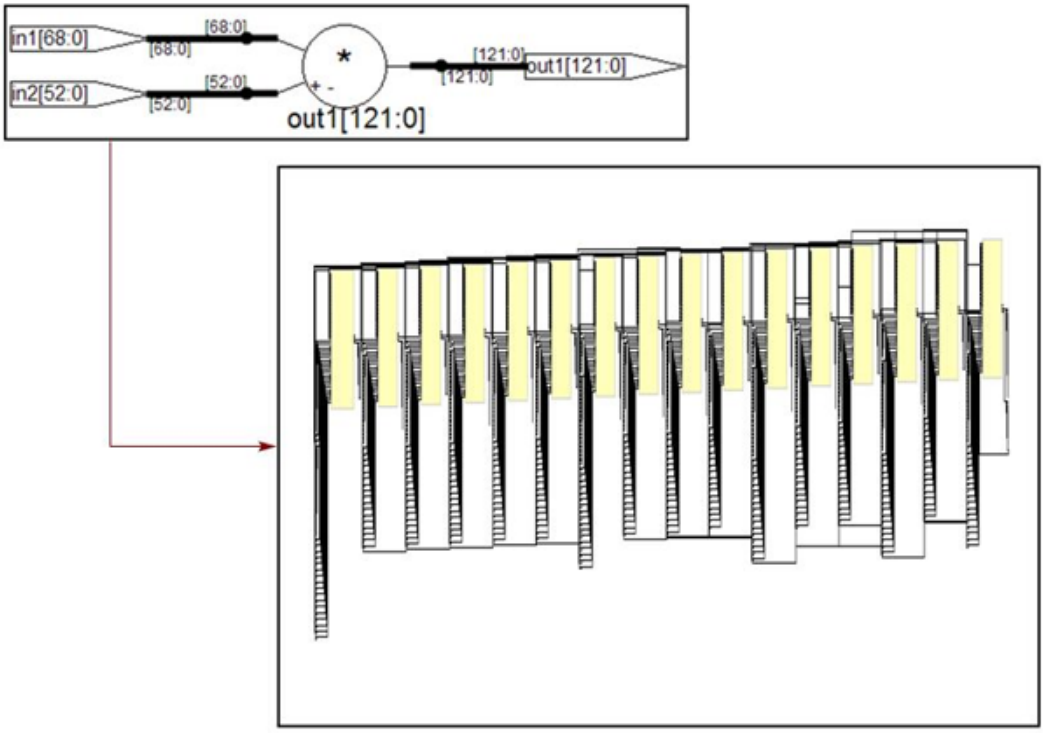
Total LUTs:0

Example 12: 69x53-Bit Signed Multiplier

RTL

```
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
```

SRS (RTL) and SRM (Technology) View



Resource Usage

The synthesis tool fractures the 69x53 multiplier into 1 mult and 15 multAdds.

Mapping to part: pa5m300fbga896std

Sequential Cells:
SLE0 uses

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

Total LUTs:0

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

RTL

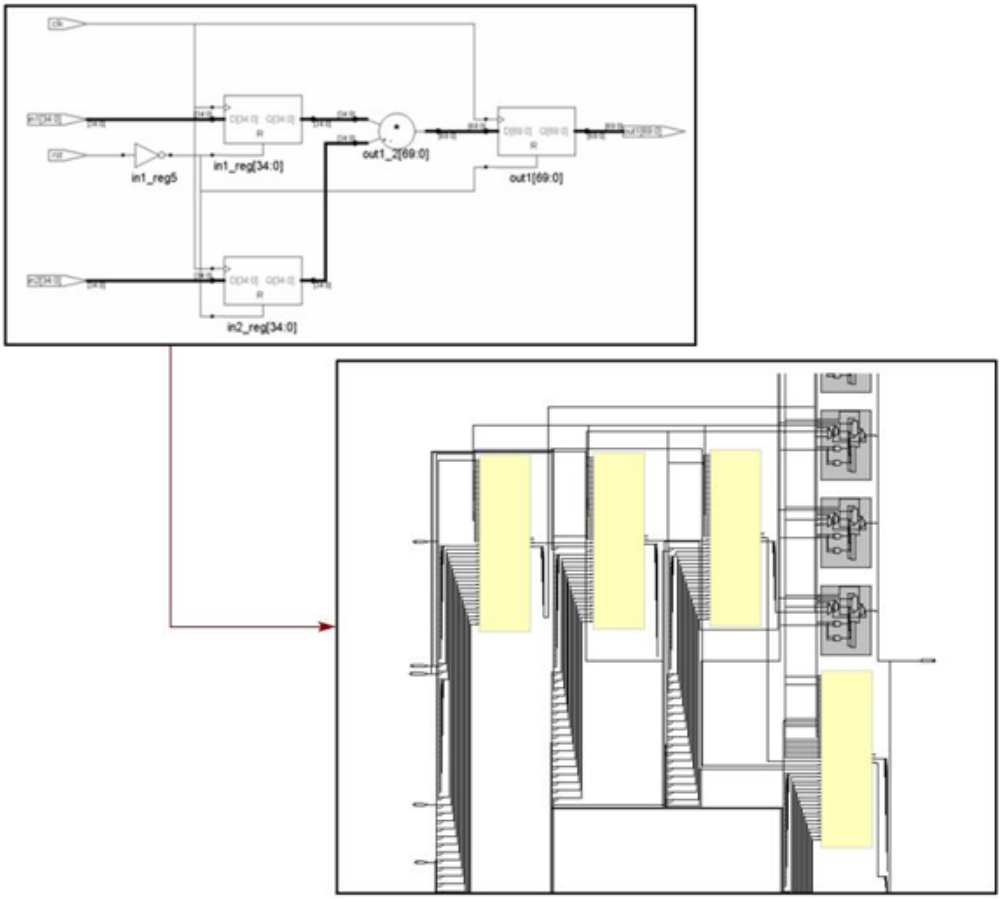
```
module sign35x35_mult ( in1, in2, clk, rst, out1 );
input signed [34:0] in1, in2;
input clk;
input rst;
output signed [69:0] out1;
reg signed [69:0] out1;
reg signed [34:0] in1_reg, in2_reg;

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 Math 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 Math block as required.

This 35x35 signed multiplier requires four Math 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 Math blocks and infers sequential primitives at the output of the Math blocks for register balancing.

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

SRS (RTL) and SRM (Technology) View



Resource Usage

The synthesis tool infers 1 mult and 3 multAdd blocks.

Mapping to part: pa5m300fbga896std
Cell usage:

Sequential Cells:
SLE 34 uses

DSP Blocks:4
MACC_PA :1 Mult

MACC_PA :3 MultAdds

Global Clock Buffers:1

Total LUTs:0

Example 14: FIR 4 Tap Filter

RTL

```
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 33 the register pipelining algorithm first pipelines registers at the output of the Math 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 Math 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 Math 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
```

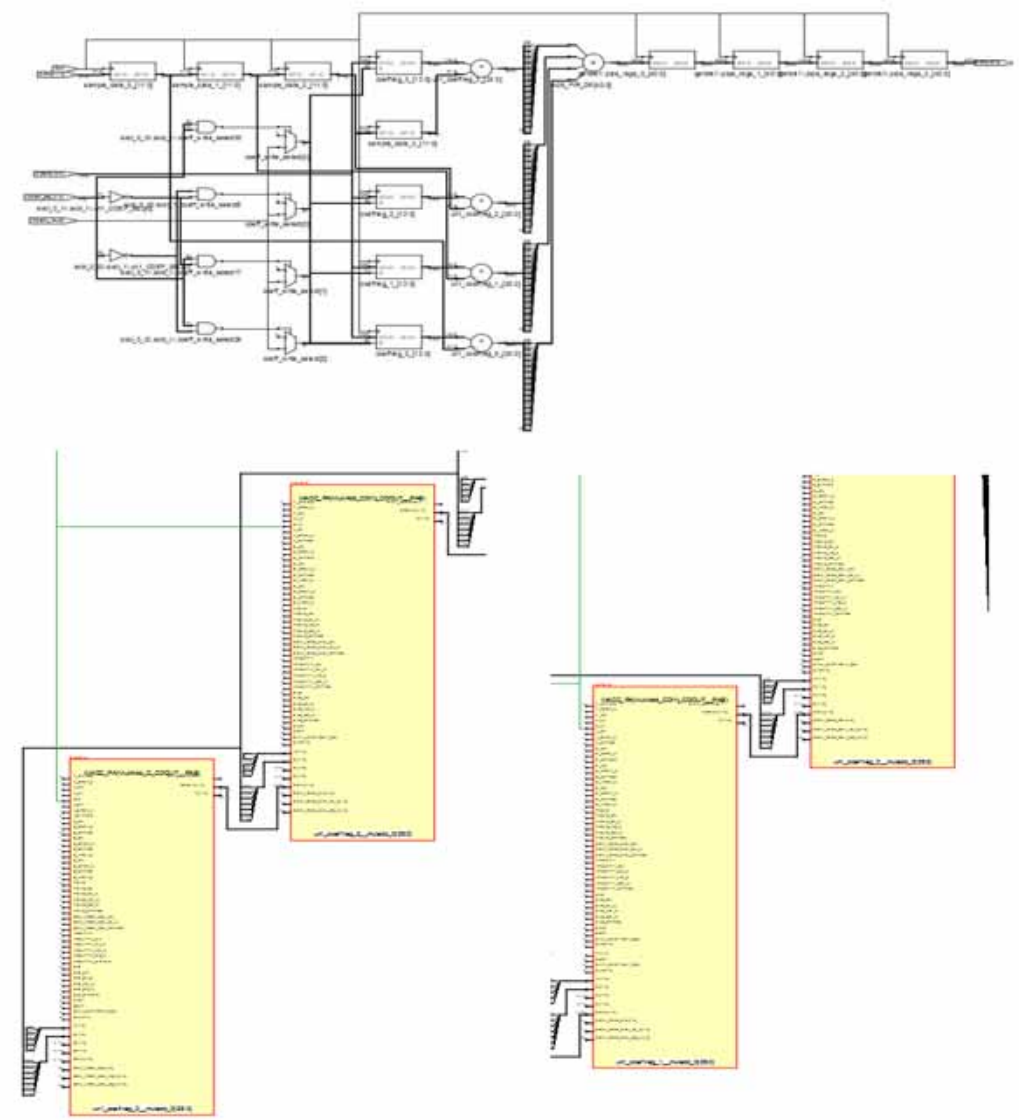
There is a register chain at the sample data input and only one register at the COEFI input. During synthesis, all output registers are pushed to the input side of the multipliers during pipelining. A warning message in the log file informs you that the tool is removing sequential instance *coeffreg* 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.

SRS (RTL) and SRM (Technology) View



Resource Usage

Mapping to part: pa5m300fbga896std
Cell usage:
CFG34 uses

Sequential Cells:
SLE150 uses

DSP Blocks:4
MACC_PA :4 MultAdds

Total LUTs:4

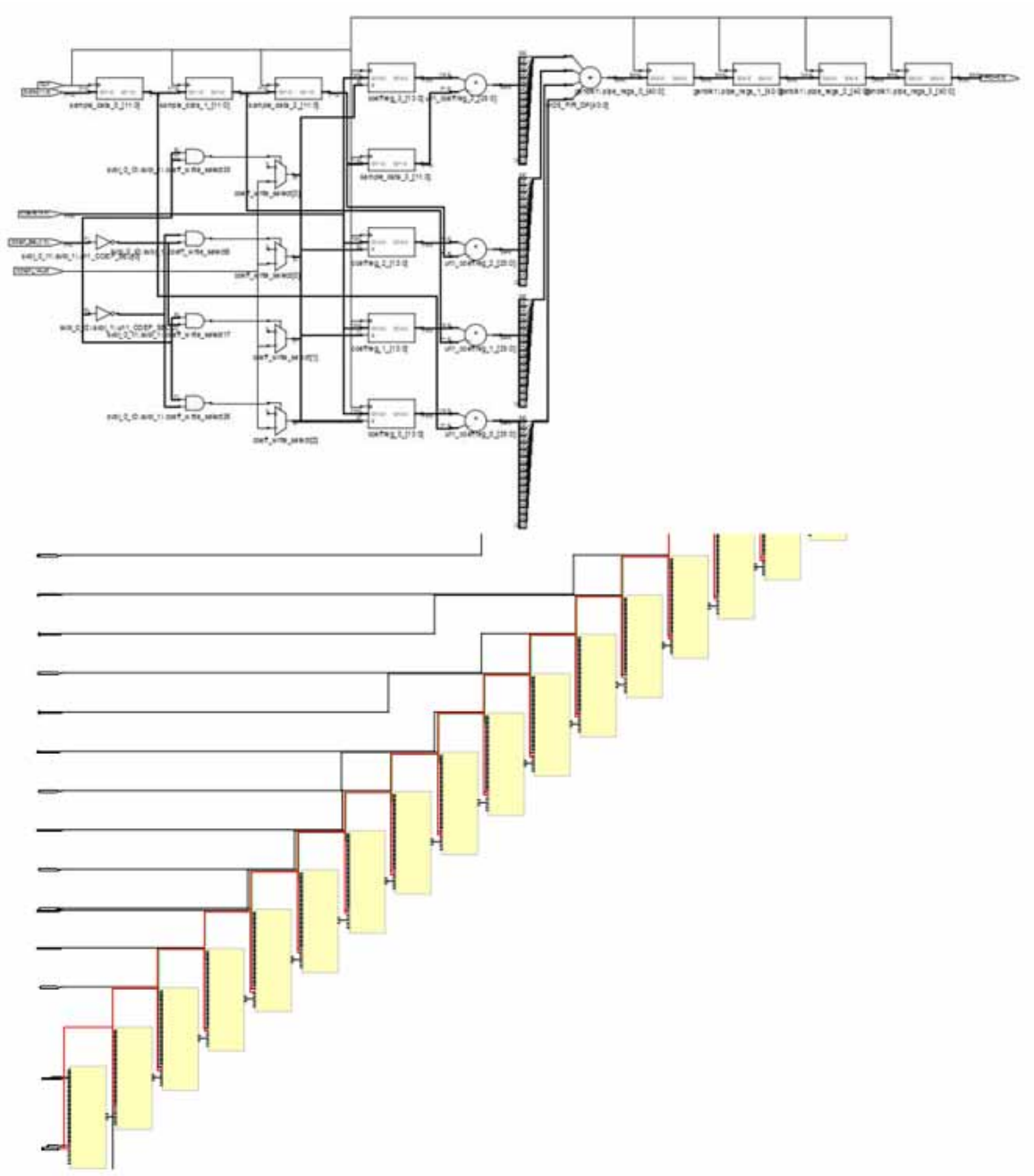
Example 15: FIR 132 Tap Filter

RTL

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

parameter TAPS = 132;// number of filter taps
parameter DATA_WIDTH = 18;
parameter COEF_WIDTH = 1;
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
```

SRS (RTL) and SRM (Technology) View



Resource Usage

Mapping to part: pa5m300fbga896std
Cell usage:
CFG2 7 uses
CFG3 1 use

CFG4 140 uses
DSP Blocks:132
MACC_PA :132 MultAdds

Inferring Math Block for Multi-Input Mult-Adds/Mult-Subs

The Math block cascade feature supports multi-input Mult-Add and Mult-Sub implementations for devices with Math blocks. The tool packs logic into Math 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 16: VHDL Test for 8 Mult-Add

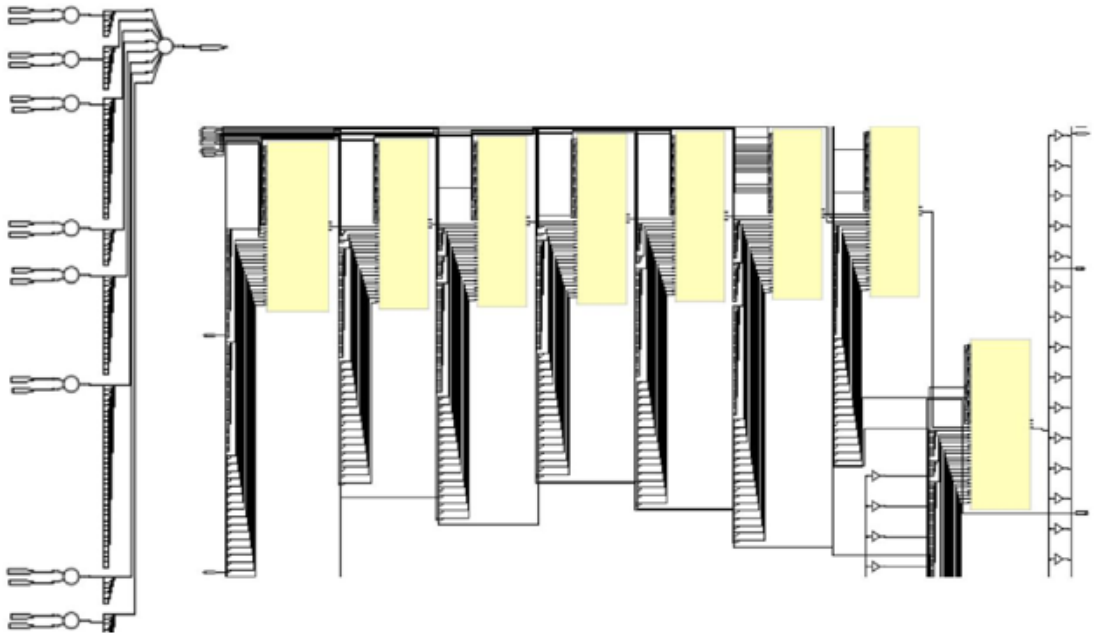
RTL

```
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;
         widthl : integer := 3;
         widthk : 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;
```

SRS (RTL) and SRM (Technology) View



Resource Usage

The synthesis tool infers 8 multAdd blocks.

Mapping to part: pa5m300fbga896std

Sequential Cells:
SLE0 uses

DSP Blocks:8
MACC_PA :8 MultAdds

Total LUTs:0

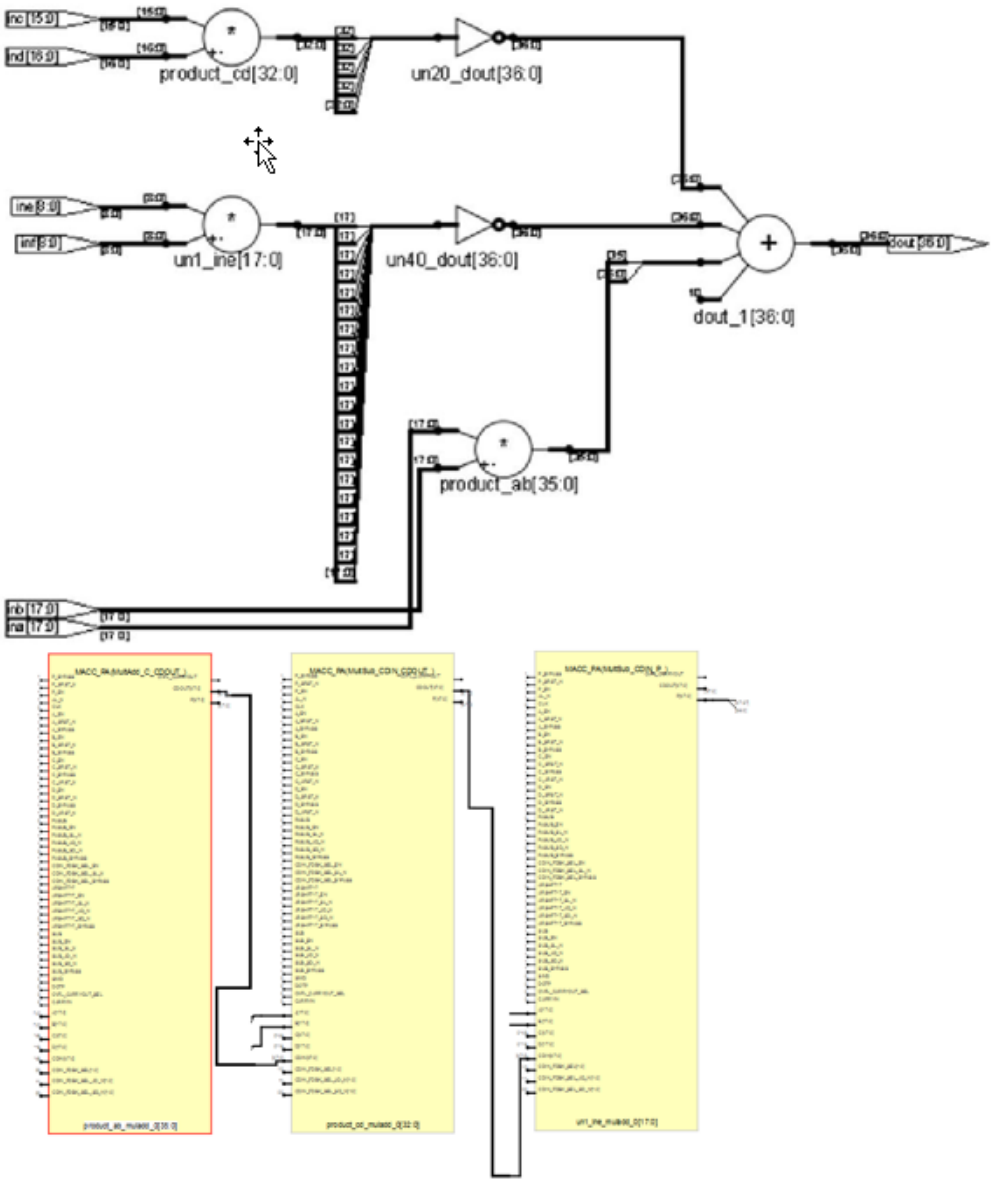
Example 17: Verilog Test for 3 Mult-Sub

RTL

```
`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
```

SRS (RTL) and SRM (Technology) View



Resource Usage

The synthesis tool infers 1 multAdd and 2 multSub blocks.

Mapping to part : pa5m300fbga896std
Sequential Cells:
SLE0 uses

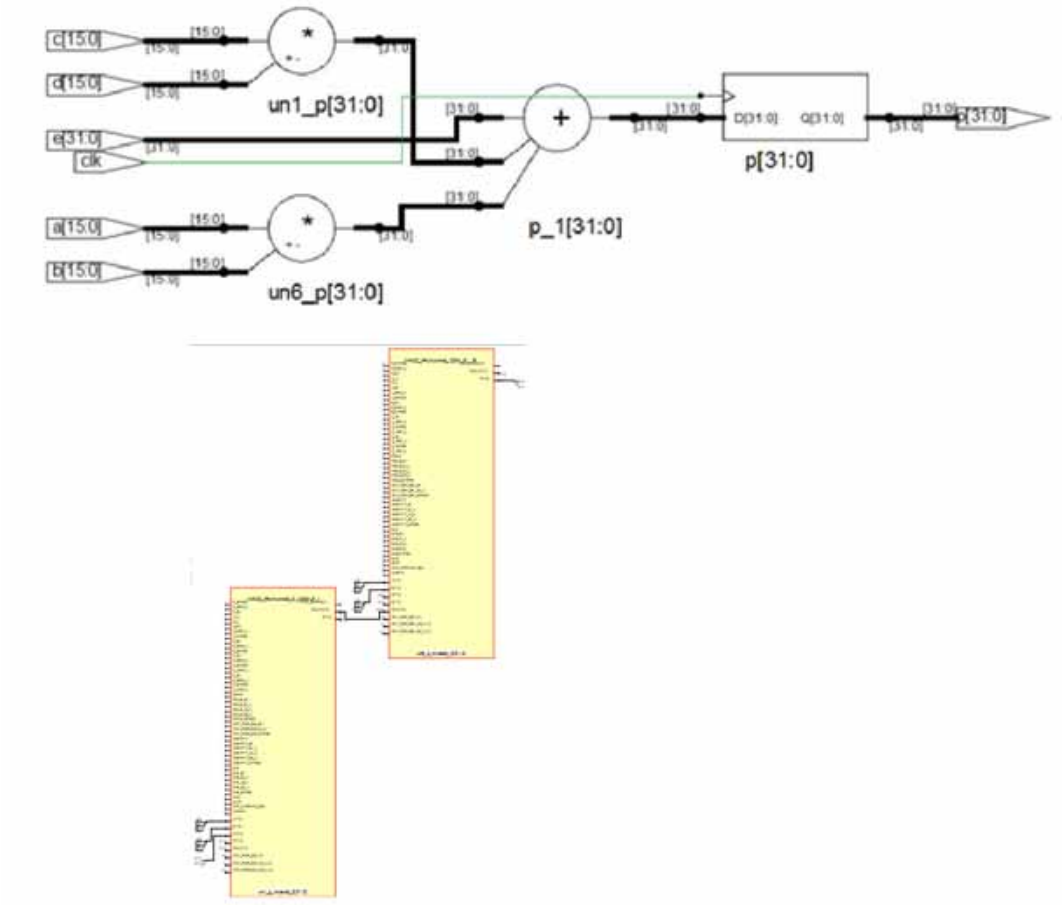
DSP Blocks: 3
MACC_PA: 1 MultAdd
MACC_PA : 2 MultSubs

Example 18: Complex Expression Example

RTL

```
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
```

SRS (RTL) and SRM (Technology) View



Resource Usage

The synthesis tool infers two multAdd blocks.

Mapping to part : pa5m300fbga896std
Cell usage:
CLKINT 1 use

Sequential Cells:
SLE 0 uses

DSP Blocks: 2
MACC_PA : 2 MultAdds

Inferring Math Blocks for Multiplier-AddSub

The Math block supports dynamic additions and subtractions. It uses the sub input of the Math 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)
  ```

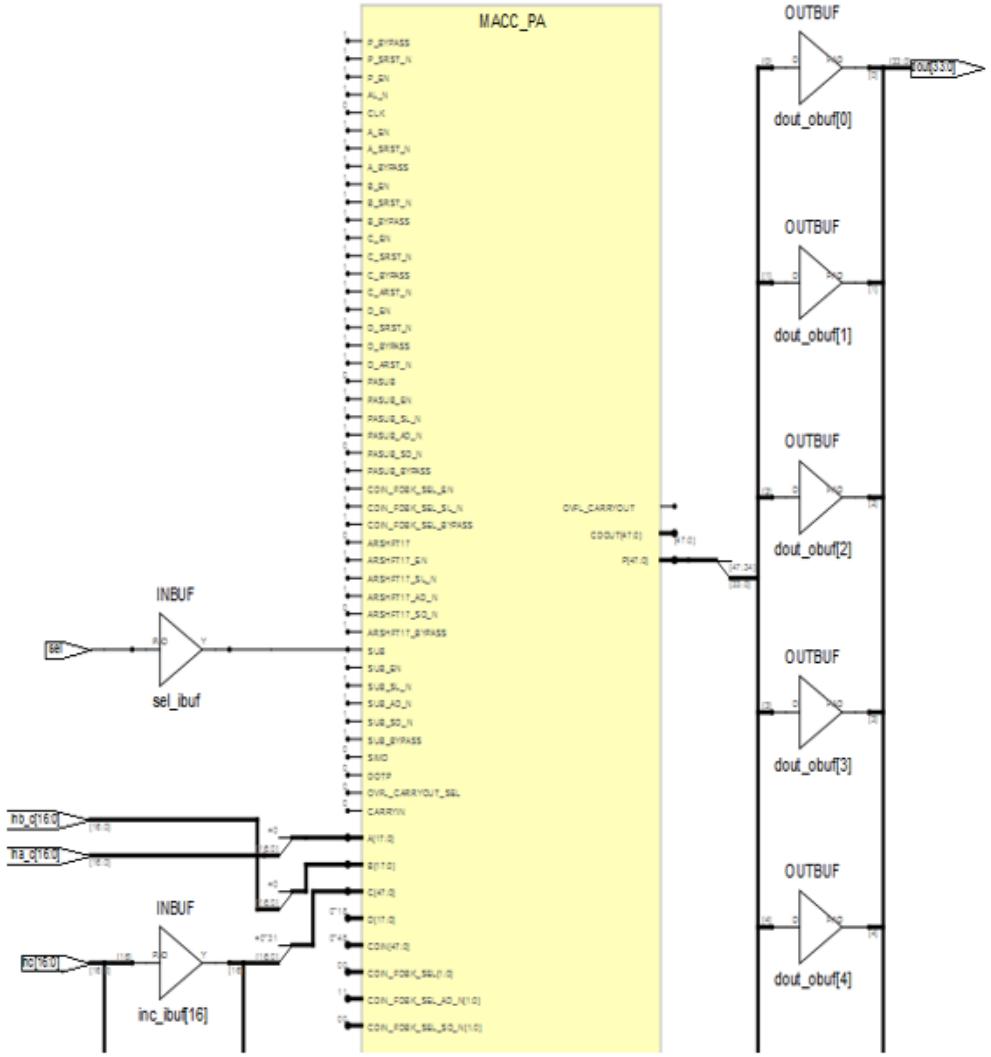
- Multiplier inputs and outputs can be registered or unregistered.

Example 19: One MultAddSub (Verilog)

RTL

```
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
```

SRS (RTL) and SRM (Technology) View



Resource Usage

The synthesis tool infers 1 MultAddSub block.

Mapping to part : pa5m300fbga896std

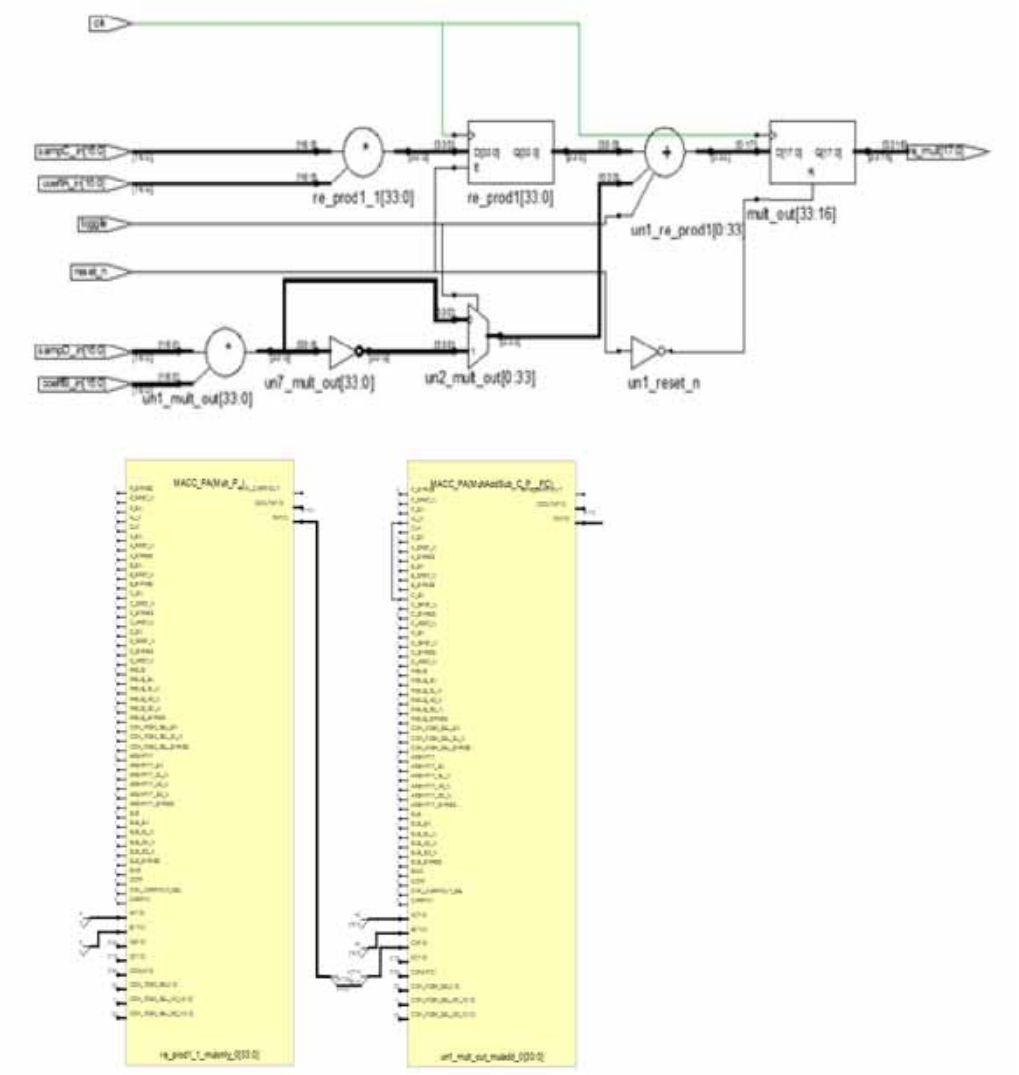
Sequential Cells:
SLE 0 uses
DSP Blocks: 1
MACC_PA : 1 MultAddSub

Example 20: One MultAddSub (VHDL)

RTL

```
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;
```

SRS (RTL) and SRM (Technology) View



Resource Usage

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

Mapping to part : pa5m300fbga896std
Cell usage:

Sequential Cells:
SLE 0 uses

DSP Blocks: 2
MACC_PA: 1 MultAddSub
MACC_PA: 1 Mult

Inferring Math Blocks for Multiplier-Accumulators

The multiplier-accumulator structures use internal paths for adder feedback loops inside the Math 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 Math block, because the Math 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 21: Verilog Test for 18X18 MultAcc with Load

RTL

```
`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

The synthesis tool infers 1 multAcc block.

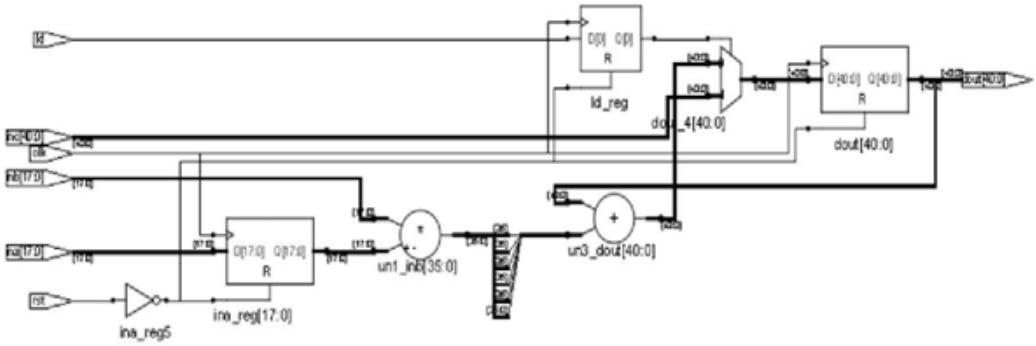
Mapping to part : pa5m300fbga896std
Cell usage:
CFG1 1 use
CFG2 41 uses

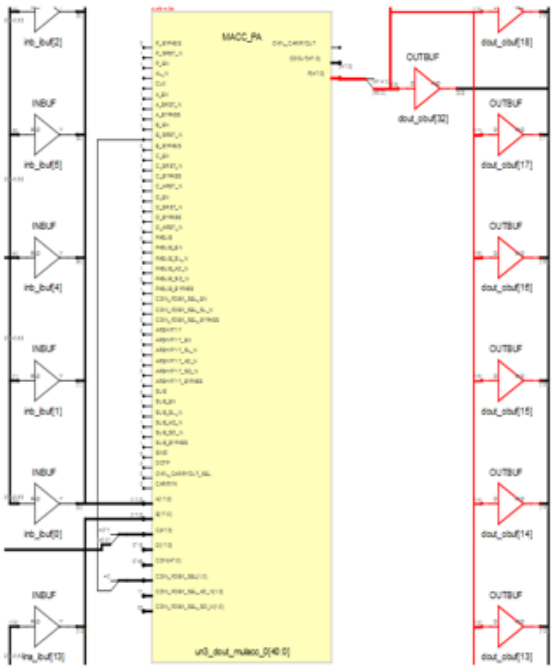
Sequential Cells:
SLE 1 use

DSP Blocks:1
MACC_PA : 1 MultAcc

Total LUTs: 42

SRS (RTL) and SRM (Technology) Views



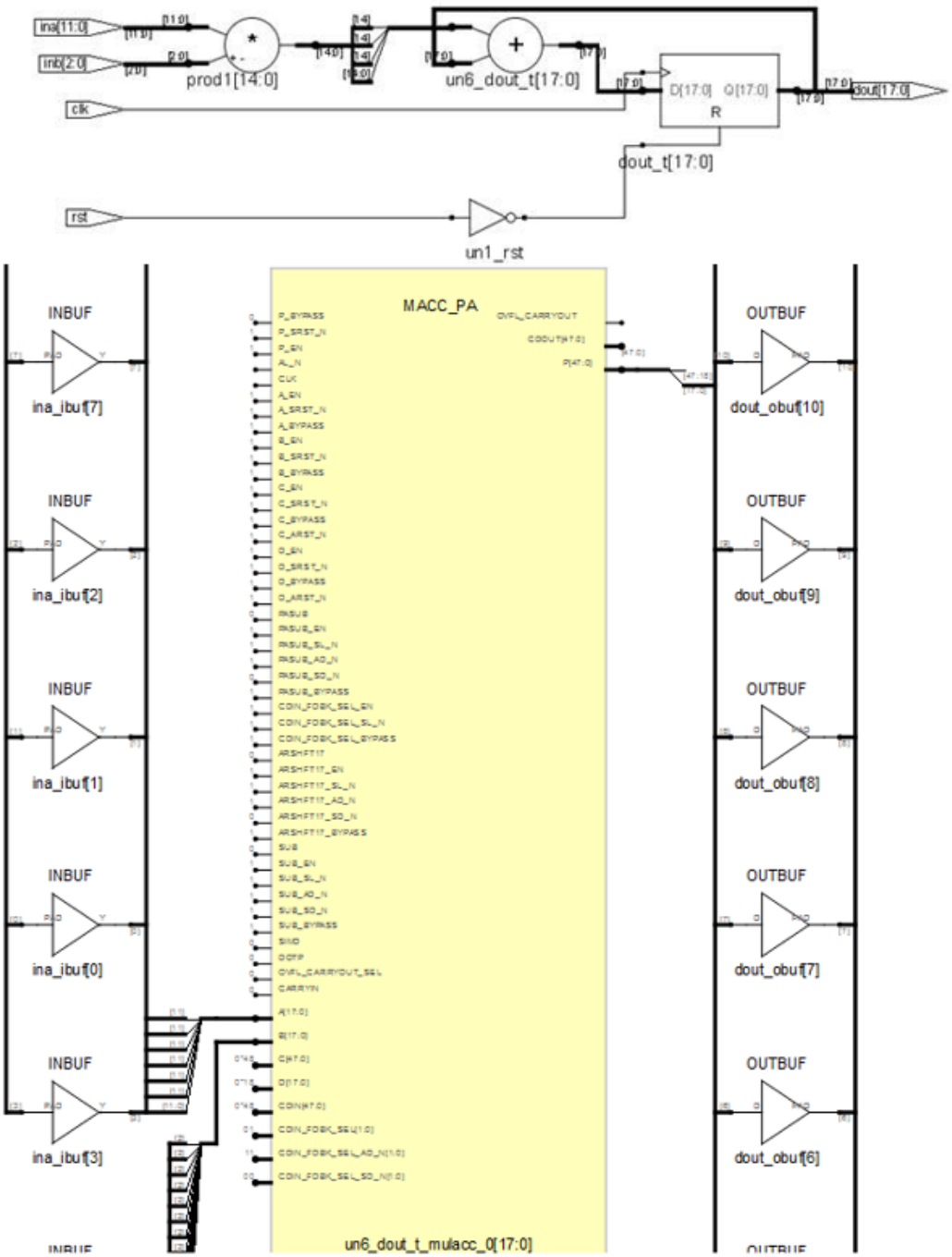


Example 22: VHDL Test for 12X3 MultAcc Without Load

RTL

```
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 and clk='1')then
         dout_t <= conv_std_logic_vector((padprod1 & signed(prod1)),width_out) +
dout_t;
   end if;
end process;

dout <= dout_t;

end arc;
```

SRS (RTL) and SRM (Technology) Views



Resource Usage

The synthesis tool infers 1 MultAcc block.

Mapping to part : pa5m300fbga896std

Sequential Cells:
SLE0 uses

DSP Blocks:1
MACC_PA :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 23: MultAdd, on page 60
- Example 24: MultAdd with Pipelined Registers, on page 63

Example 23: MultAdd

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

RTL

```
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

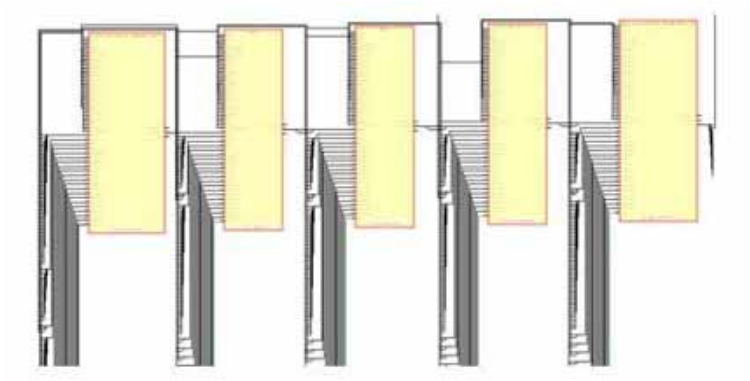
The synthesis tool infers five multAdd blocks with 32 SLE's.

Mapping to part : pa5m300fbga896std
Cell usage:
CLKINT 1 use

Sequential Cells:
SLE 32 uses

DSP Blocks: 5
MACC_PA : 5 MultAdds

SRS (RTL) and SRM (Technology) View



Example 24: MultAdd with Pipelined Registers

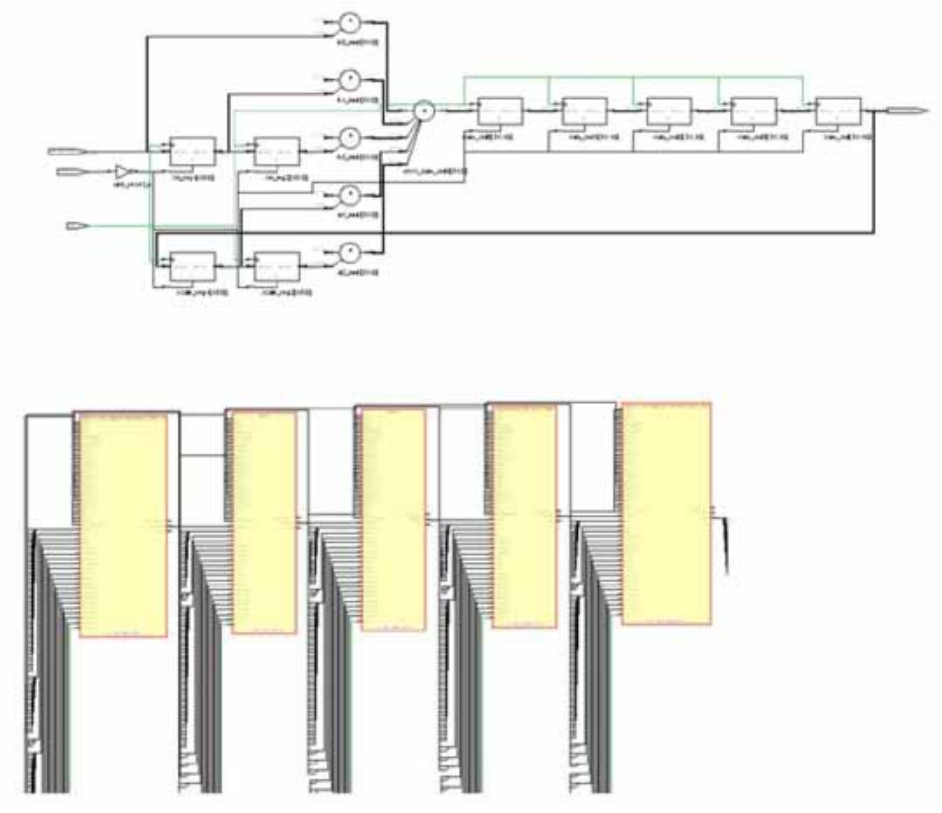
This example has the same functionality as Example 22, but the coding style is changed to pipelined registers. With pipeline registers, the synthesis tool does pipeline register retiming and then inserts registers at the input side to improve timing. The timing performance is improved to ~400MHz. The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

RTL

```
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
```

SRS (RTL) and SRM (Technology) Views



Resource Usage

Mapping to part : pa5m300fbga896std
Cell usage:
CLKINT 1 use

Sequential Cells:
SLE 96 uses

DSP Blocks: 5
MACC_PA: 5 MultAdds

Global Clock Buffers: 2

Total LUTs: 0

Inferring Math block in DOTP mode

The Math 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 25: Unsigned MultAdd Computation

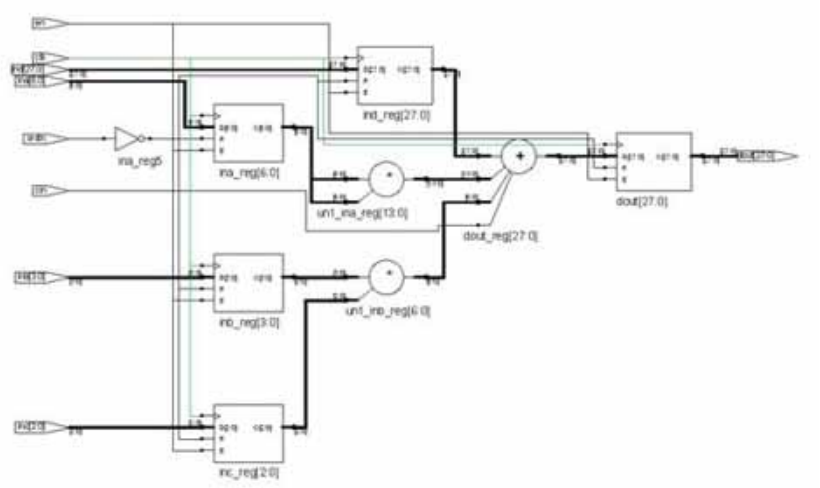
The RTL is for DOTP computation of sqr(a) + bc + d + cin. All the inputs and outputs are registered with asynchronous active-low resets and active-high enable signals. The synthesis tool infers a single Math block in DOTP mode with MultAdd configuration and packs the adder input registers in SLEs.

RTL

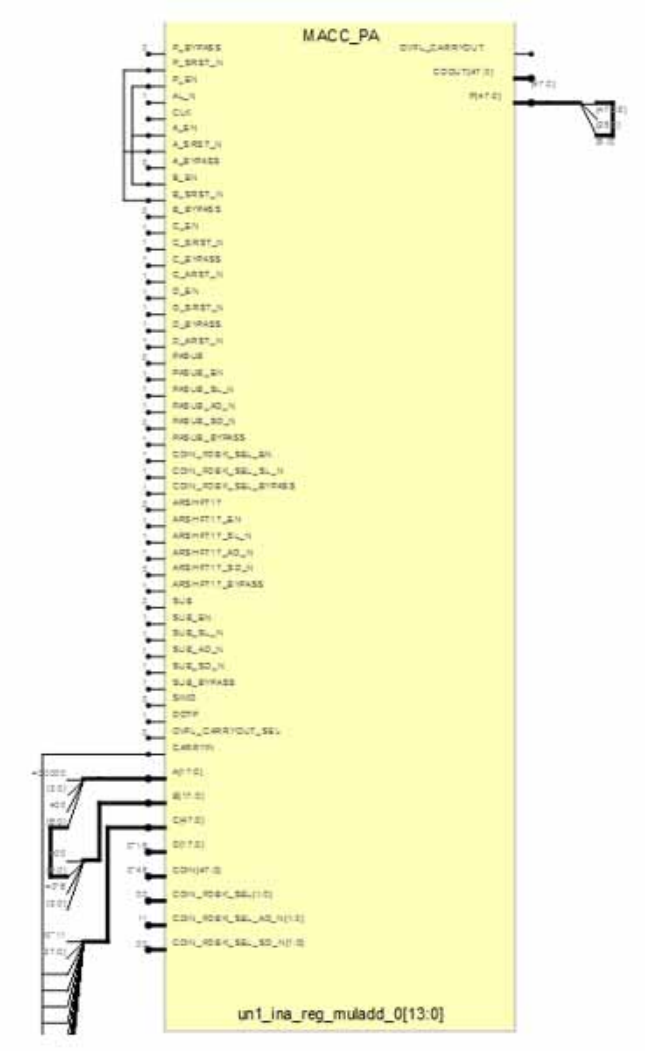
```
module dotp_add_ioreg_unsign_srstn_en (clk, srstn, en, ina, inb, inc, ind, cin,
dout);
input clk, srstn, en;
input cin;
input [6:0] ina;
input [3:0] inb;
input [2:0] inc;
input [27 : 0] ind;
output reg [30:0] dout;

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
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Mapping to part : pa5m300fbga896std
Cell usage:

CLKINT 1 use
CFG2 1 use

Sequential Cells:
SLE 28 uses

DSP Blocks: 1
MACC_PA :1 MultAdd

Global Clock Buffers: 1

Total LUTs: 1

Example 26: Direct-Form 8-tap Finite Impulse Filter

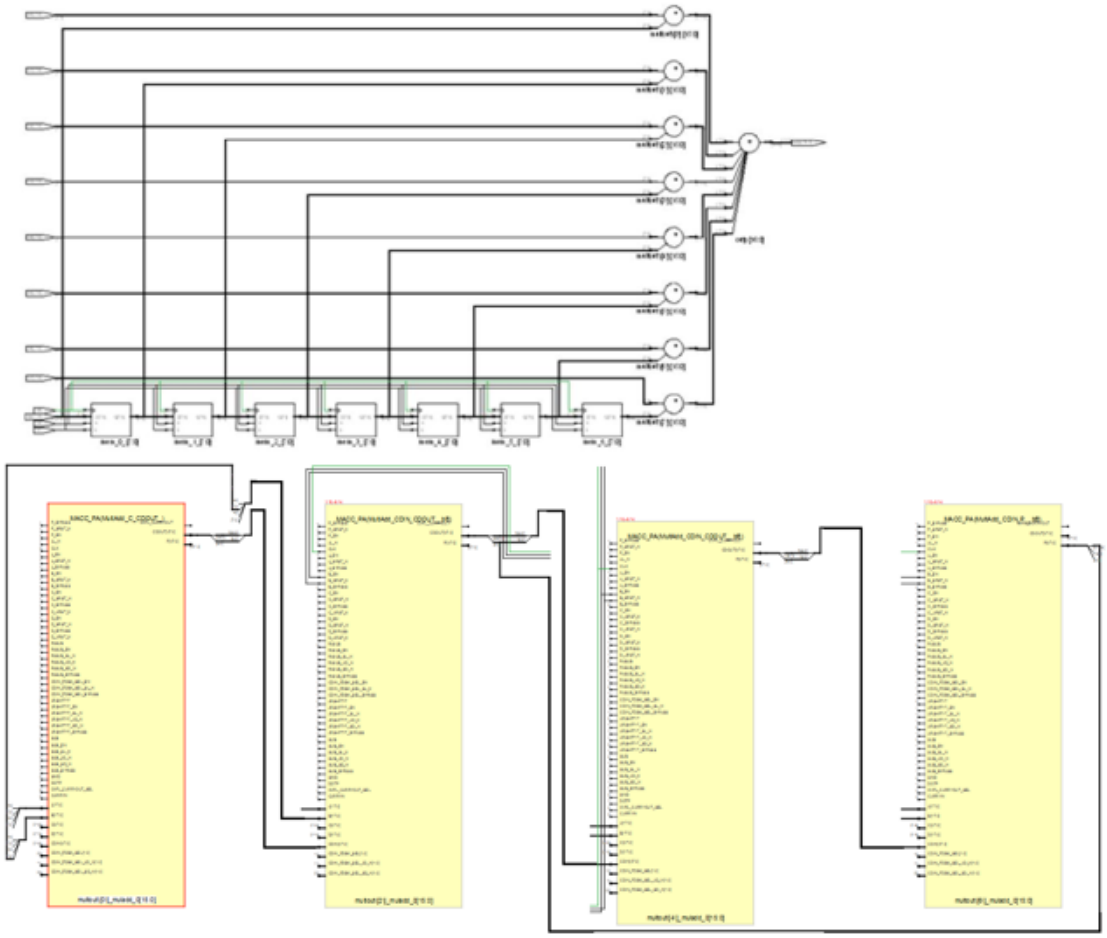
The RTL is for DOTP computation of (ab + bc) +/- d. All the inputs and outputs are registered with asynchronous active-low resets and active-high enable signals.

RTL

```
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
```

SRS (RTL) and SRM (Technology) Views



Resource Usage

Mapping to part: pa5m300fbga896std
Cell usage:
CLKINT 1 use
CFG11 use
CFG2 1 use

Sequential Cells:
SLE 48 uses

DSP Blocks: 4
MACC_PA : 4 MultAdds

Global Clock Buffers: 1

Total LUTs: 2

Example 27: DOTP with multiple clocks

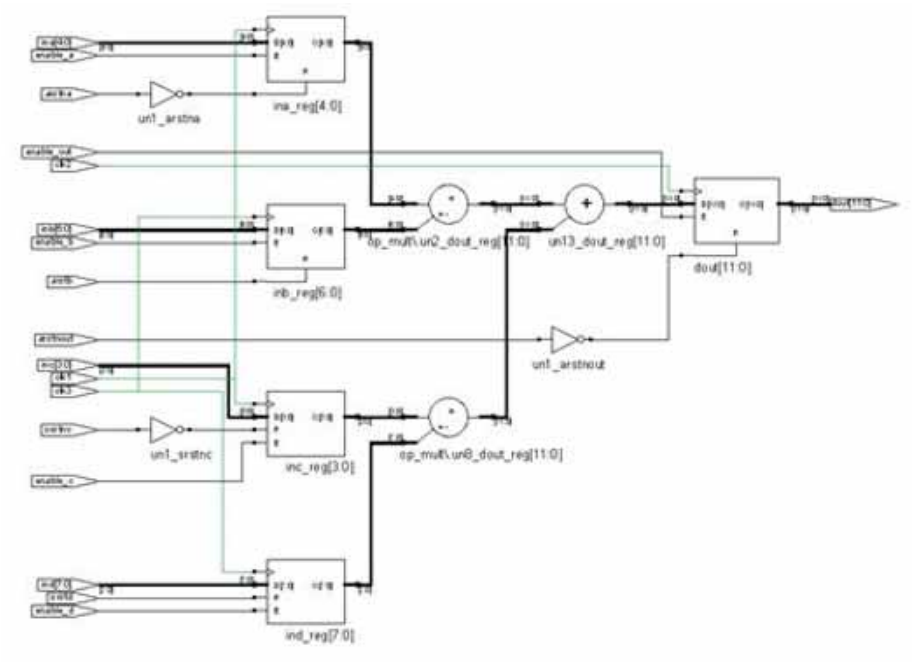
The RTL is for DOTP computation of (ab + cd). The A and B inputs are registered with asynchronous active-low resets and active-high enable signals, the C and D inputs are registered with synchronous active-low resets and synchronous active-high resets but with active-high enable signals. The output is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

RTL

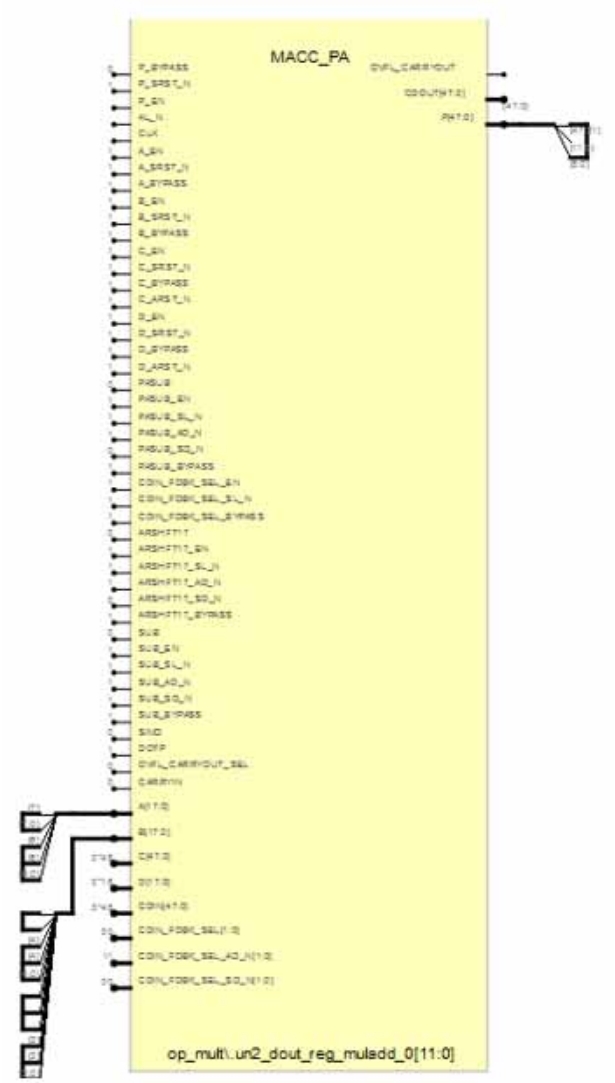
```
library ieee;

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;
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Mapping to part: pa5m300fbga896std

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

Sequential Cells:
SLE 26 uses

DSP Blocks: 1
MACC_PA 1 MultAdd

Global Clock Buffers: 3

Total LUTs: 4

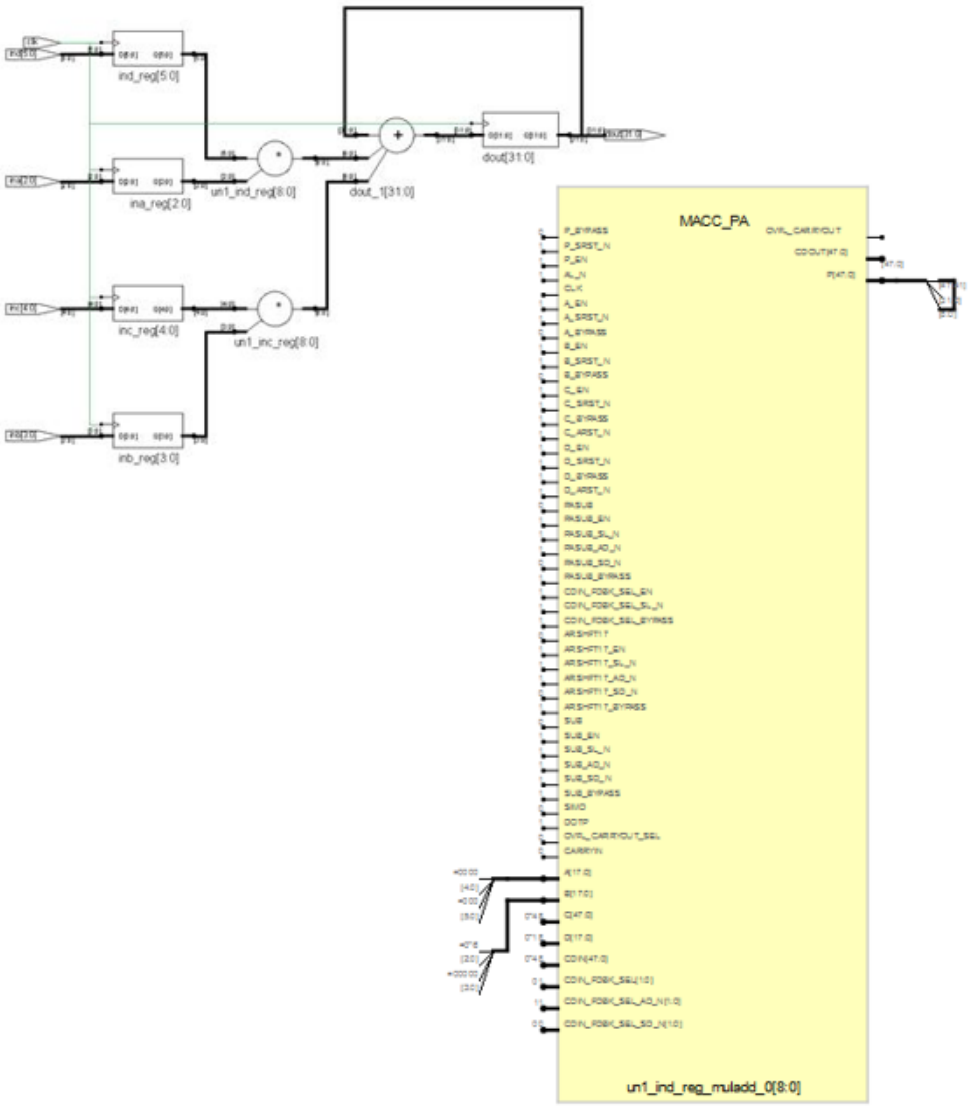
Example 28: DOTP with MultACC

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

RTL

```
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
```

SRS (RTL) and SRM (Technology) Views



Resource Usage

Mapping to part : pa5m300fbga896std
Cell usage:

Sequential Cells:
SLE 0 uses

DSP Blocks: 1
MACC_PA: 1 MultAcc

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

Total LUTs: 0

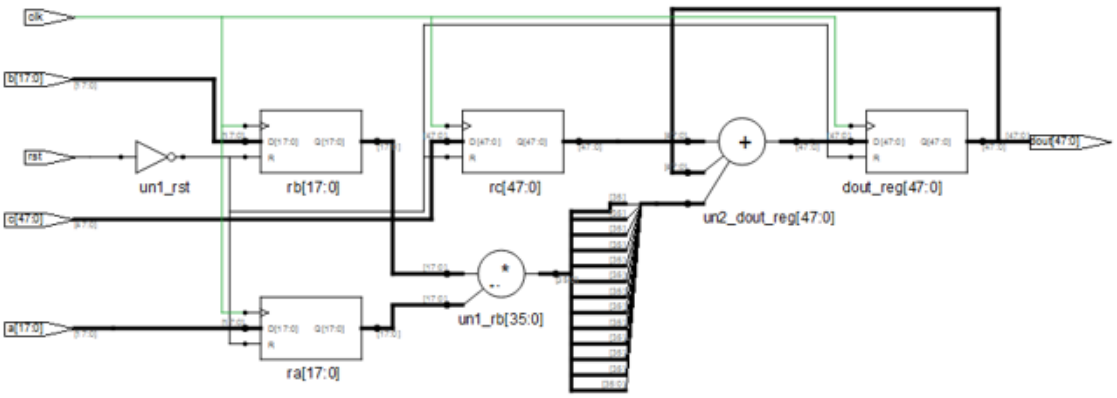
Example 29: MultAcc with C input

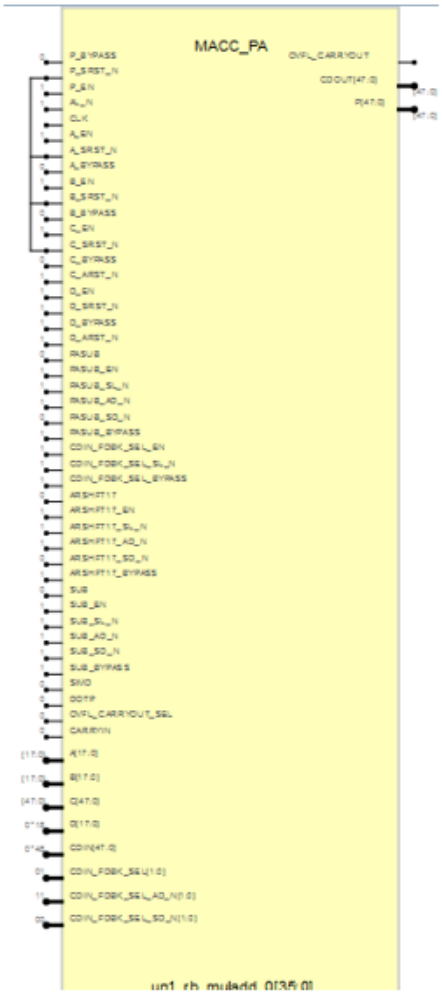
The RTL below is for MultAcc with C input. After synthesis, Math block is inferred with C input packing.

RTL

```
`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 [47:0] c;
output signed [47:0] dout;

reg signed [17:0] ra, rb;
reg signed [47:0] dout_reg, rc;

always @(posedge clk)
begin
   if (~rst) begin
   dout_reg <= 48'b0;
   ra <= 17'b0;
rb <= 17'b0;
   rc <= 48'b0;
end
else begin
   ra <= a;
rb <= b;
   rc <= c;
   dout_reg <= dout_reg + rc + ra * rb;
   end
end

assign dout = dout_reg;
endmodule
```

SRS (RTL) and SRM (Technology) Views





Resource Usage

Mapping to part: pa5m300fbga896std
Cell usage:

Sequential Cells: pa5m300fbga896std
SLE 0 uses

DSP Blocks: 1
MACC_PA: 1 MultAdd

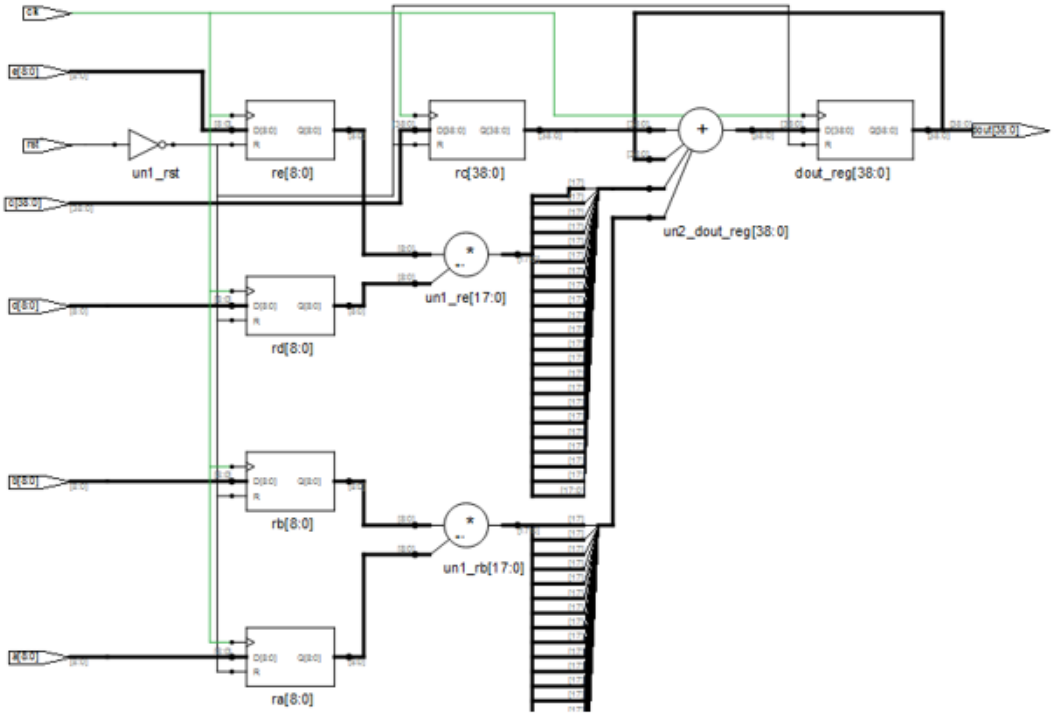
Example 30: DOTP MultAcc with C input

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

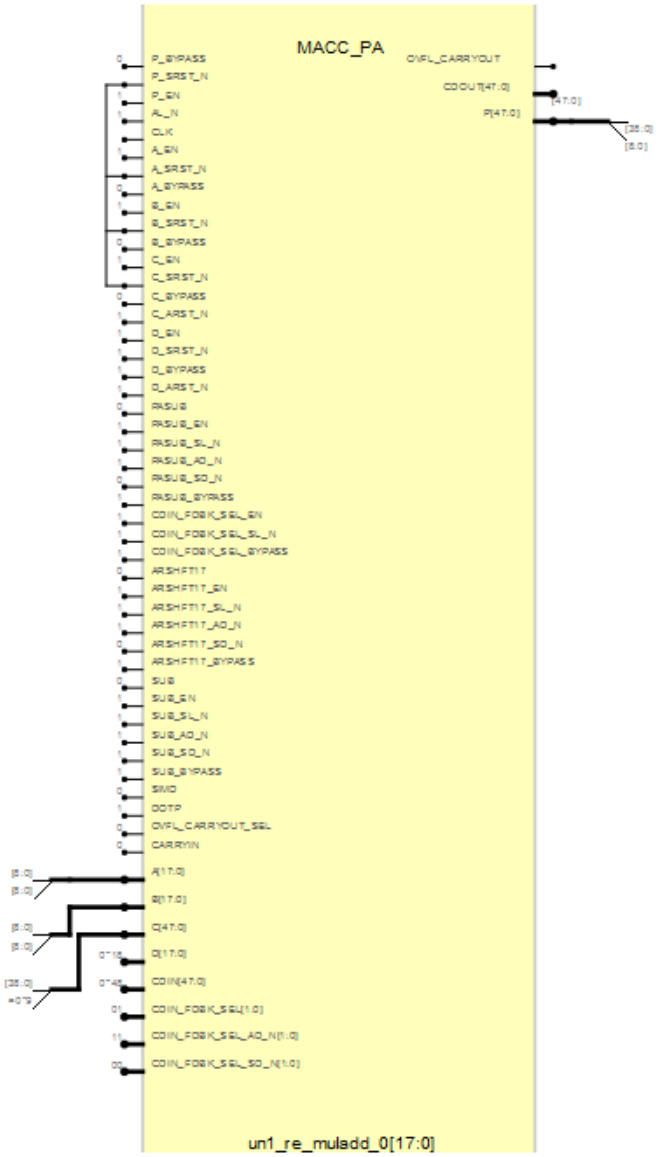
RTL

```
module test (clk, rst, a, b, c, d, e, dout);
input clk, rst;
input signed [8:0] a, b, d, e;
input signed [38:0] c;
output signed [38:0] dout;

reg signed [8:0] ra, rb, rd, re;
reg signed [38:0] dout_reg, rc;

always @(posedge clk)
begin
   if (~rst) begin
         dout_reg <= 39'b0;
         ra <= 9'b0;
      rb <= 9'b0;
      rc <= 39'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
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Mapping to part: pa5m300fbga896std
Cell usage:

Sequential Cells:
SLE 0 uses

DSP Blocks: 1
MACC_PA: 1 MultACC

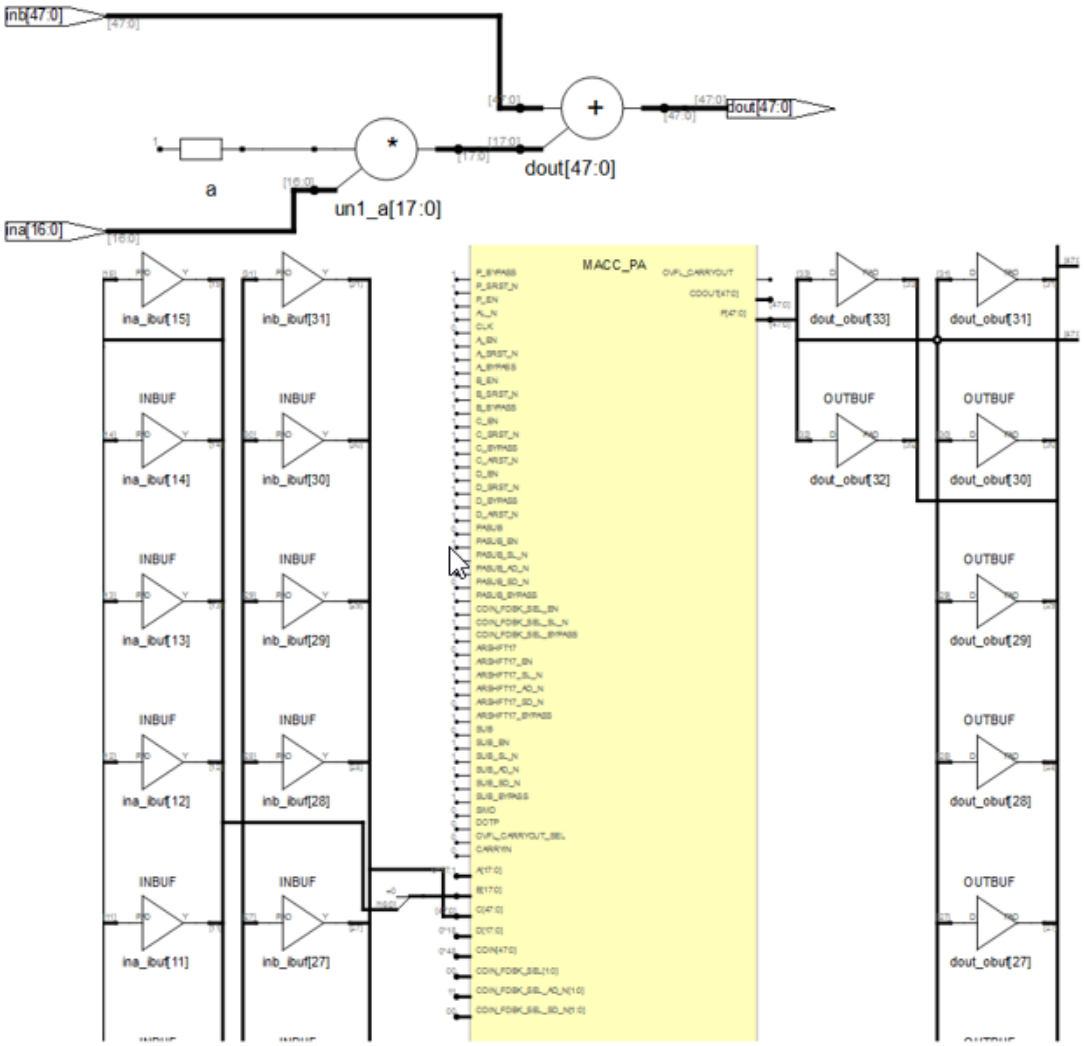
Example 31: MultAdd with constant

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

RTL

```
module test ( ina, inb, dout);
parameter widtha = 17;
parameter widthb = 47;
parameter width_out = 47;
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
```

SRS (RTL) and SRM (Technology) Views



Resource Usage

The resource usage details show that the MultAdd code was implemented in one Math block.

Mapping to part: pa5m300fbga896std

Sequential Cells:
SLE 0 uses

DSP Blocks:1
MACC_PA: 1 Mult

Total LUTs:0

Inferring Math Block for Pre-Adder

The following examples show Pre-adder packing into Math block.

- Example 32: Math Block with Preadder-Multiplier, on page 85
- Example 33: Math Block with Pre-Adder - Signed, on page 88
- Example 34: Math Block with Pre-Adder - Unsigned, on page 90
- Example 35: Math Block with Pre-Sub, on page 92
- Example 36: Math Block with Pre-Adder In DOTP Mode, on page 94
- Example 37: Math Block with Pre-Adder - VHDL, on page 96
- Example 38: Math Block with Pre-Adder with C input and dynamic Add/Sub, on page 100

Example 32: Math Block with Preadder-Multiplier

The synthesis tool packs Pre-Adder logic into one Math block.

See the following examples:

- Unsigned Pre-adderMult Verilog Example (P = {(B + D) x A), on page 85
- Signed Pre-adderMult Verilog Example (P = {(B + D) x A), on page 87

Unsigned Pre-adderMult Verilog Example (P = {(B + D) x A)

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

RTL

```
module mult_sub ( in1, in2, in3, clk, rst, out1 );
input [16:0] in1, in2;
input [16:0] in3;
input clk;
input rst;

output reg [47:0] out1;
reg [16:0] in1_reg, in2_reg, in3_reg;

always @ ( posedge clk ) begin

if (~rst) begin
in1_reg <= 17'b0;
in2_reg <= 17'b0;
in3_reg <= 17'b0;
out1 <= 48'b0;
end else
begin
```

```
in1_reg <= in1;
in2_reg <= in2;
in3_reg <= in3;
out1 <= (in1_reg + in2_reg) * in3_reg;
end
end

endmodule
```

Resource Usage

The log file resource usage report shows that everything is packed and one mult is inferred.

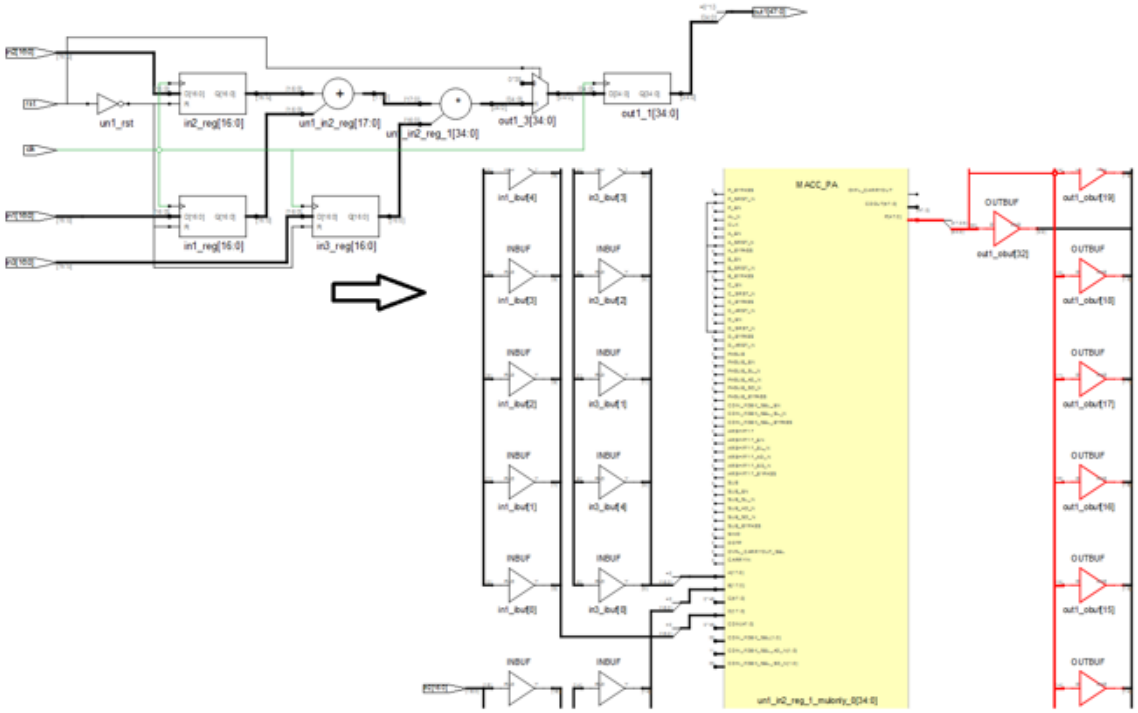
Mapping to part: pa5m300fbga896std

Sequential Cells:
SLE 0 uses

DSP Blocks:1
MACC_PA: 1 Mult

Total LUTs: 0

SRS (RTL) and SRM (Technology) Views



Signed Pre-adderMult Verilog Example (P = {(B + D) x A)

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

RTL

```
module mult_sub ( in1, in2, in3, clk, rst, out1 );
input signed [17:0] in1, in2;
input signed [17:0] in3;
input clk;
input rst;

output reg signed [47:0] out1;
reg signed [16:0] in1_reg, in2_reg, in3_reg;

always @ ( posedge clk ) begin
if (~rst) begin
in1_reg <= 17'b0;
in2_reg <= 17'b0;
in3_reg <= 17'b0;
out1 <= 48'b0;
end else
begin

in1_reg <= in1;
in2_reg <= in2;
in3_reg <= in3;
out1 <= (in1_reg + in2_reg) * in3_reg;
end
end
endmodule
```

Resource Usage

The log file resource usage report shows that everything is packed into Math block and one mult is inferred.

Mapping to part: pa5m300fbga896std

Sequential Cells:
SLE 0 uses

DSP Blocks:1
MACC_PA: 1 Mult

Total LUTs: 0

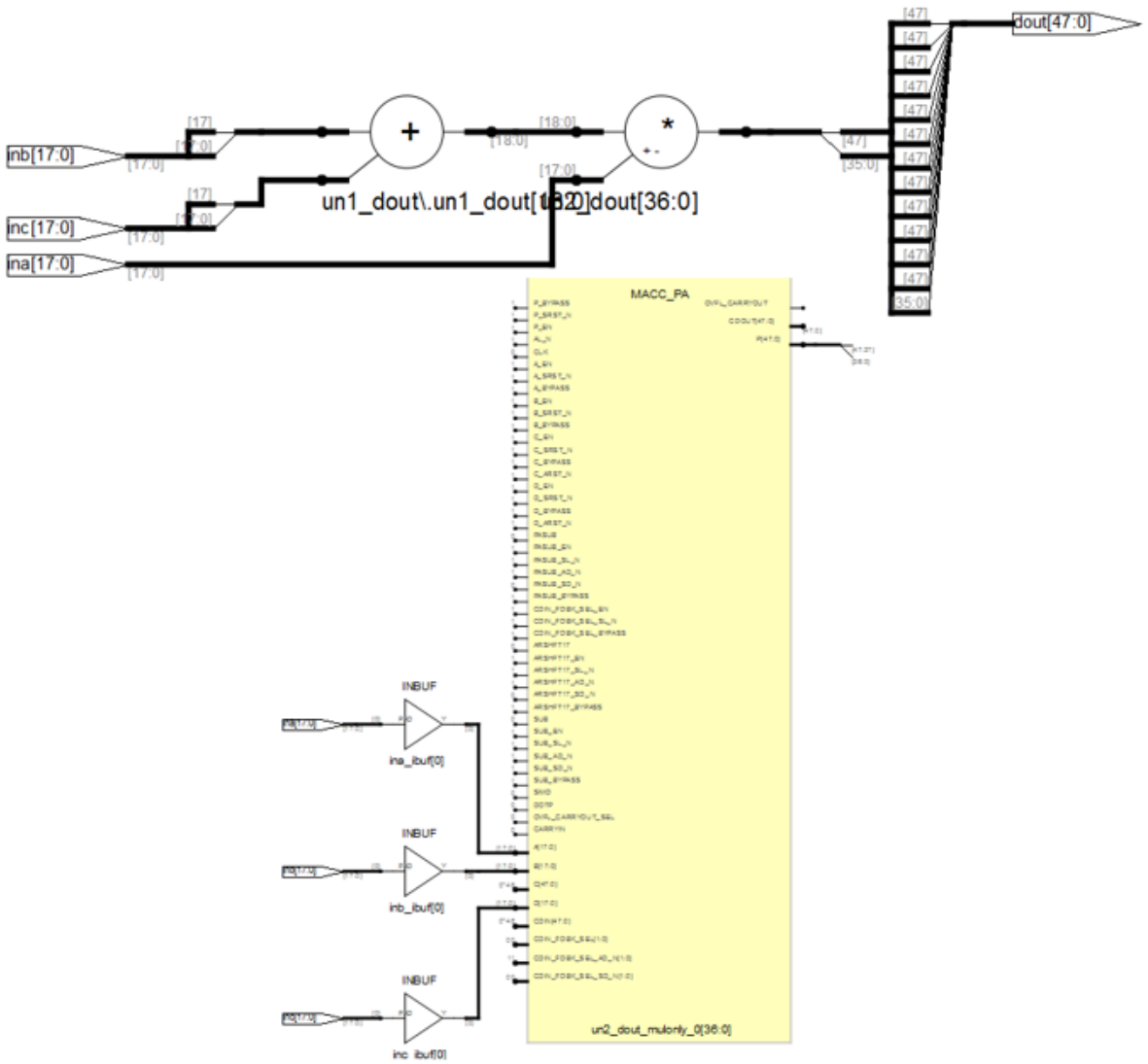
Example 33: Math Block with Pre-Adder - Signed

This example shows packing of pre-Adder followed by signed multiplier into 1 Math block for the equation p = (inb + inc) * ina.

RTL

```
`timescale 1 ns/10 ps
`ifdef synthesis
module test ( ina, inb, inc, dout);
`else
module test_rtl ( ina, inb, inc, dout);
`endif
parameter widtha = 18;
parameter widthb = 17;
parameter widthc = 17;
parameter width_out = 48;
input signed [widtha-1:0] ina;
input signed [widthb-1:0] inb;
input signed [widthc-1:0] inc;
output signed [width_out-1:0] dout;
assign dout =  (inb + inc) * ina;
endmodule
```

SRS (RTL) and SRM (Technology) Views



Resource Usage

The resource usage report shows that the logic is implemented in one Math block.

Mapping to part: pa5m300fbga896std

Sequential Cells:
SLE 0 uses

DSP Blocks:1
MACC_PA: 1 Mult

Total LUTs: 0

Example 34: Math Block with Pre-Adder - Unsigned

This example shows packing of pre-Adder followed by unsigned multiplier into 1 Math block for the equation p = (inb + inc) * ina.

RTL

```
`timescale 1 ns/10 ps
`ifdef synthesis
module test ( ina, inb, inc, dout);
`else
module test_rtl ( ina, inb, inc, dout);
`endif

parameter widtha = 17;
parameter widthb = 17;
parameter widthc = 17;
parameter width_out = 48;

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

assign dout =  (inb + inc) * ina;

endmodule
```

Resource Usage

The resource usage report shows that the logic is implemented in one Math block.

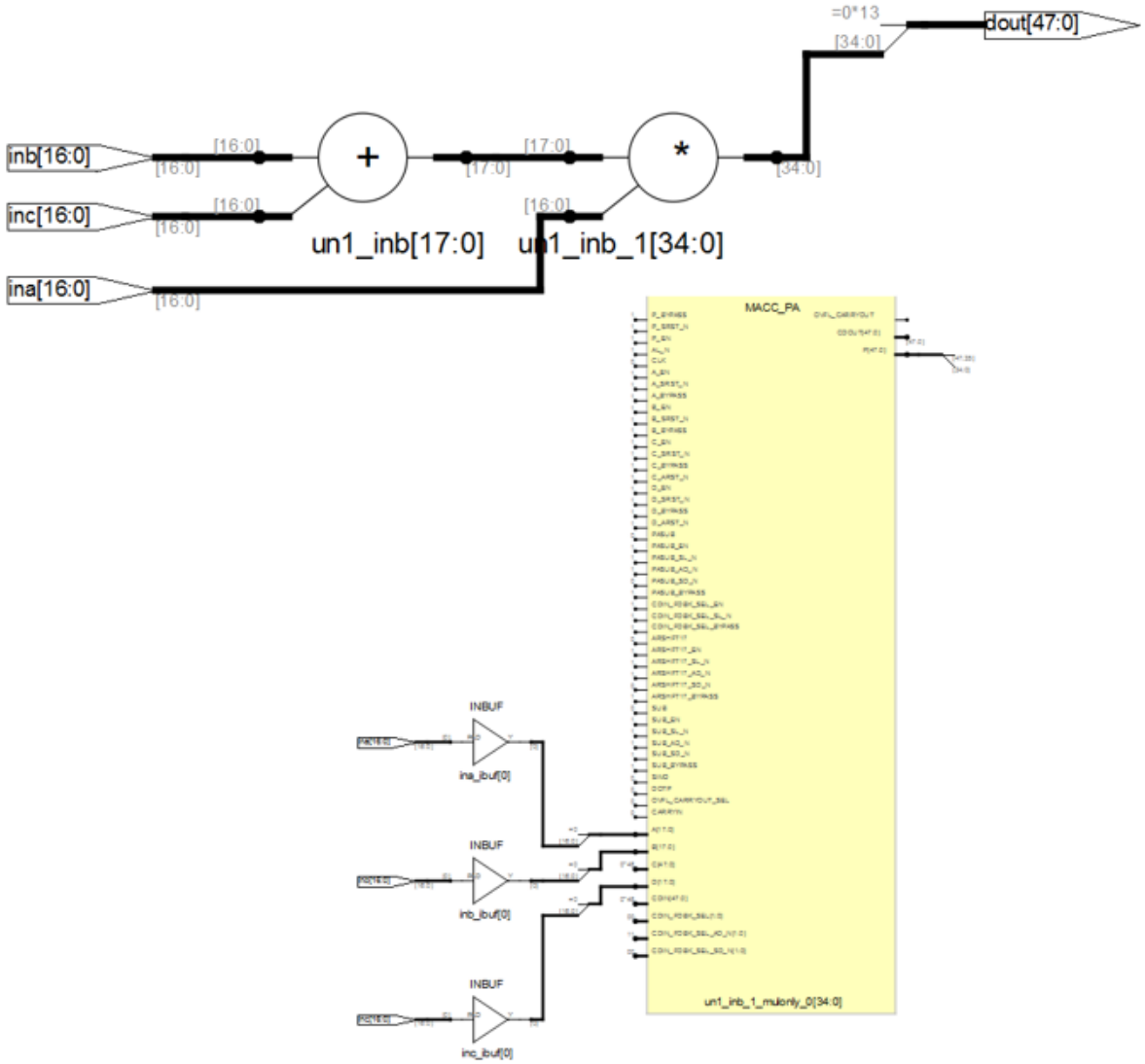
Mapping to part: pa5m300fbga896std

Sequential Cells:
SLE 0 uses

DSP Blocks:1
MACC_PA: 1 Mult

Total LUTs:0

SRS (RTL) and SRM (Technology) Views



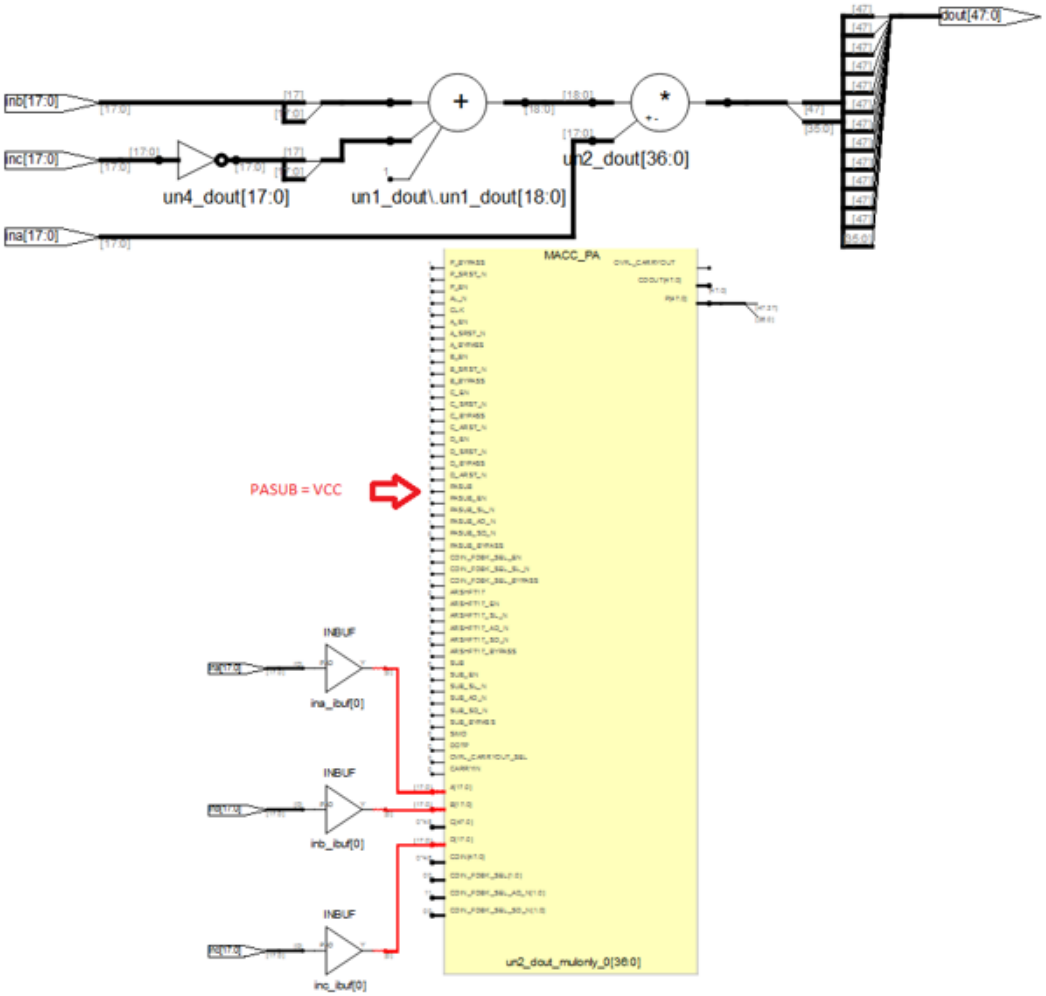
Example 35: Math Block with Pre-Sub

This example shows packing of pre-Subtractor followed by signed multiplier into 1 Math block for the equation p = (inb - inc) * ina.

RTL

```
`timescale 1 ns/10 ps
`ifdef synthesis
module test ( ina, inb, inc, dout);
`else
module test_rtl ( ina, inb, inc, dout);
`endif

parameter widtha = 18;

parameter widthb = 18;

parameter widthc = 18;
parameter width_out = 48;

input signed [widtha-1:0] ina;
input signed [widthb-1:0] inb;
input signed [widthc-1:0] inc;
output signed [width_out-1:0] dout;

assign dout =  (inb - inc) * ina;

endmodule
```

SRS (RTL) and SRM (Technology) Views



Resource Usage

The resource usage report shows that the logic is implemented in one Math block.

Mapping to part: pa5m300fbga896std

Sequential Cells:
SLE 0 uses

DSP Blocks:1
MACC_PA: 1 Mult

Total LUTs:0

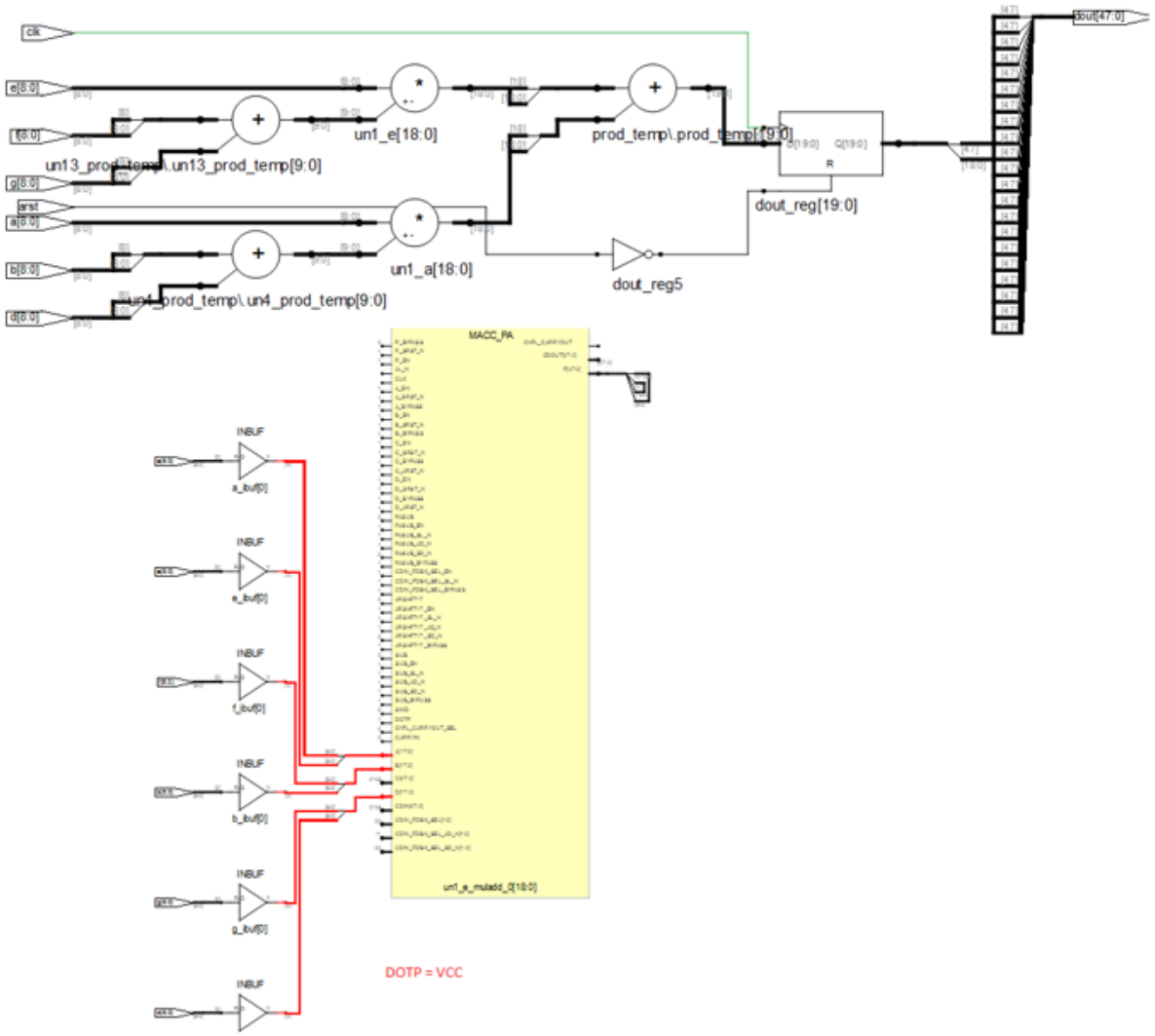
Example 36: Math Block with Pre-Adder In DOTP Mode

This example shows packing of pre-Adder followed by signed multiplier into 1 Math block in DOTP mode for the equation p = (a * (b + d)) + (e * (f + g));.

RTL

```
`ifdef synthesis
module test (clk, arst, a, b, c, d, e, f, g, dout);
`else
module test_rtl (clk, arst, a, b, c, d, e, f, g, dout);
`endif

input clk, arst;
input  signed  [8:0] a, e;
input  signed  [8:0] b, d, f, g;
input  signed  [47:0] c;
output signed [47:0] dout;

reg signed [47:0] dout_reg;

wire signed [47:0] prod_temp;
wire signed [47:0] dout_wire;

assign prod_temp = (a * (b + d)) + (e * (f + g));

always @(posedge clk or negedge arst)
begin
   if (~arst)
         dout_reg <= 48'b0;
      else
         dout_reg <= prod_temp;
   end

assign dout = dout_reg;

endmodule
```

SRS (RTL) and SRM (Technology) Views



Resource Usage

The resource usage report shows that the logic is implemented in one Math block in DOTP mode.

Mapping to part: pa5m300fbga896std

Sequential Cells:
SLE 0 uses

DSP Blocks:1
MACC_PA: 1 Mult

Total LUTs:0

Example 37: Math Block with Pre-Adder - VHDL

The following is a VHDL example which shows packing of pre-Adder followed by signed multiplier into 1 Math block using resize () function.

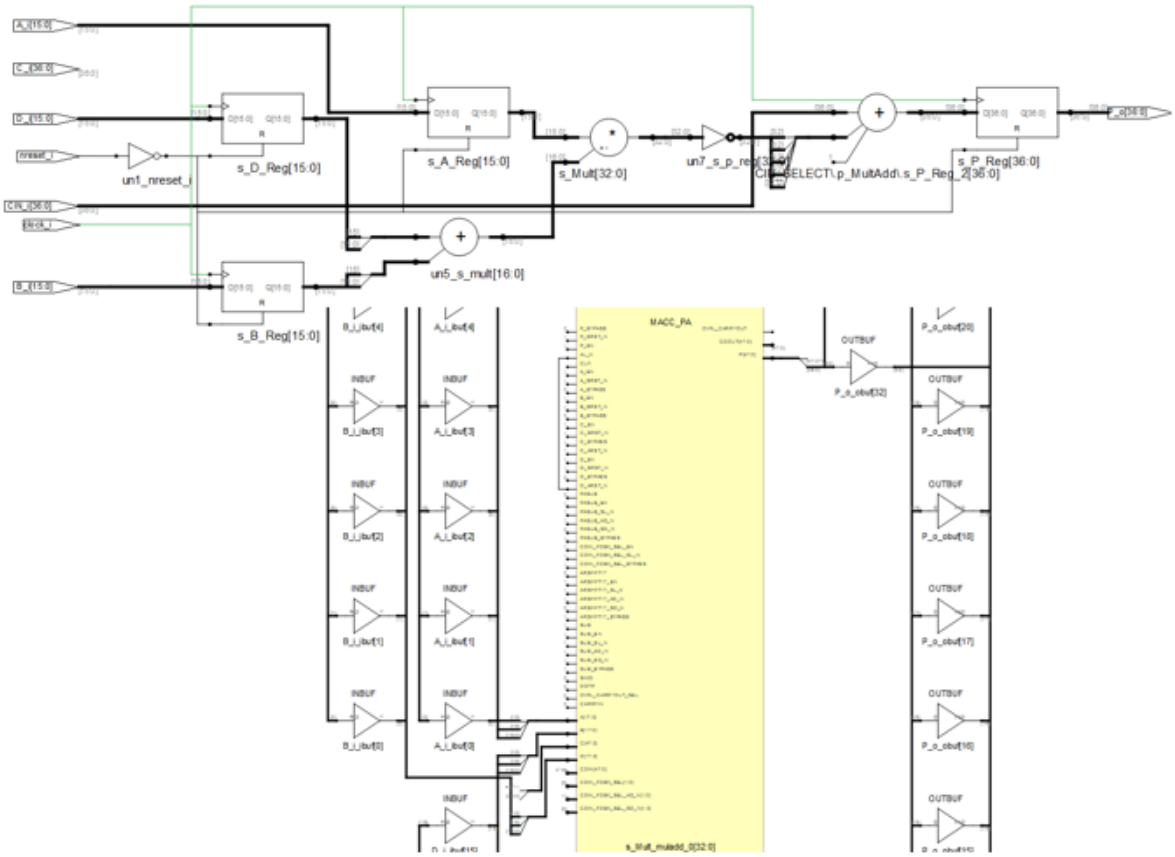
RTL

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

entity preadd_mult_adder is
      eneric (
            g_A_WIDTH : integer := 16;
            g_B_WIDTH : integer := 16;
            g_C_WIDTH : integer := 37;
            g_D_WIDTH : integer := 16;
            g_CIN_WIDTH : integer := 37;
            g_MUL_WIDTH : integer := 33;
            g_P_WIDTH : integer := 37;
            g_PERADD_WIDTH : integer := 17;
            g_PRE_ADD : integer := 1;
            g_ADD: integer := 1;-- 0=> ADD 1=> SUB
            g_C_CIN_Select : integer := 1;   --Add/sub from C port select =0,
--Add/sub from Cin port select =1,
            g_RESET_STATE : integer := 0

            );
         port (
               clock_i : in  std_logic;
               nreset_i : in  std_logic;
               A_i : in  std_logic_vector(g_A_WIDTH-1 downto 0);
               B_i : in  std_logic_vector(g_B_WIDTH-1 downto 0);
               C_i : in  std_logic_vector(g_C_WIDTH-1 downto 0);
               D_i : in  std_logic_vector(g_D_WIDTH-1 downto 0);
               CIN_i: in  std_logic_vector(g_CIN_WIDTH-1 downto 0);
               P_o : out std_logic_vector(g_P_WIDTH-1 downto 0)
               );
end preadd_mult_adder;
architecture architecture_preadd_mult_adder of preadd_mult_adder is
   signal s_reset_state      : std_logic;

   signal s_A_Reg : std_logic_vector(g_A_WIDTH-1 downto 0);
   signal s_B_Reg : std_logic_vector(g_B_WIDTH-1 downto 0);
   signal s_C_Reg : std_logic_vector(g_C_WIDTH-1 downto 0);
   signal s_D_Reg : std_logic_vector(g_D_WIDTH-1 downto 0);
   signal s_CIN_Reg : std_logic_vector(g_CIN_WIDTH-1 downto 0);
   signal s_Mult : std_logic_vector(g_MUL_WIDTH-1 downto 0);
   signal s_P_Reg : std_logic_vector(g_P_WIDTH-1 downto 0);

begin

Reset_gen1: if g_RESET_STATE = 0 generate
         s_reset_state <= '0';
         end generate;
```

```
Reset_gen2: if g_RESET_STATE = 1 generate
         s_reset_state <= '1';
      end generate;

 --
*******************************************************************************
******

   --** process name  : p_InReg

   --** Description   : This process registers inputs A,B & C

   --

   --
*******************************************************************************
******

 p_InReg : process (nreset_i, clock_i)
      begin
         if(nreset_i = s_reset_state) then
            s_A_Reg    <= (others => '0') ;
            s_B_Reg    <= (others => '0') ;
               s_C_Reg    <= (others => '0') ;
            s_D_Reg    <= (others => '0');
         elsif(rising_edge(clock_i)) then
            s_A_Reg    <=  A_i ;
            s_B_Reg    <=  B_i ;
               s_C_Reg    <=  C_i ;
            s_D_Reg    <=  D_i ;
            end if;
      end process p_InReg ;

   s_CIN_Reg <=  CIN_i;

 --
*******************************************************************************
******

   --** process name  : p_MultAddSub

   --** Description   : This process implements  C + (A*B)  or C-(A*B)

   --

   --
*******************************************************************************
******

C_PREADD: if (g_PRE_ADD = 1) generate
   --s_Mult  <= std_logic_vector(resize ((signed(s_B_Reg) + signed(s_D_Reg))*
signed(s_A_Reg), g_MUL_WIDTH) );

      s_Mult  <= std_logic_vector( resize ((resize(signed(s_B_Reg),g_PERADD_WIDTH)
+ resize(signed(s_D_Reg),g_PERADD_WIDTH))*signed(s_A_Reg), g_MUL_WIDTH)  );

end generate;

C_PRESUB: if (g_PRE_ADD = 0) generate
   s_Mult  <= std_logic_vector(resize ((signed(s_B_Reg) - signed(s_D_Reg))*
signed(s_A_Reg), g_MUL_WIDTH) );
end generate;
```

```
C_SELECT: if (g_C_CIN_Select = 0 ) generate
         p_MultAdd: process (nreset_i, clock_i)
            begin
               if (nreset_i =  s_reset_state) then
               s_P_Reg       <= (others=>'0');
            elsif(rising_edge(clock_i)) then
               if (g_ADD = 1) then
               s_P_Reg <= std_logic_vector(  resize (signed(s_C_Reg),g_P_WIDTH) -
(resize (signed(s_Mult), g_P_WIDTH) )    )  ;
               else
               s_P_Reg <= std_logic_vector(   resize (signed(s_C_Reg),g_P_WIDTH) +
(resize (signed(s_Mult), g_P_WIDTH) )    )  ;
               end if;
            end if;
         end process p_MultAdd;
end generate;

 --
**************************************************************************
******

   --** process name  : p_MultAddSub

   --** Description   : This process implements  Cin + (A*B)  or Cin -(A*B)

   --

   --
**************************************************************************
******

CIN_SELECT: if (g_C_CIN_Select = 1 and g_ADD = 1 ) generate
      p_MultAdd: process (nreset_i, clock_i)
            begin
               if(nreset_i =  s_reset_state) then
               s_P_Reg       <= (others=>'0');
            elsif(rising_edge(clock_i)) then
               s_P_Reg <= std_logic_vector(  resize (signed(s_CIN_Reg),g_P_WIDTH) -
(resize (signed(s_Mult), g_P_WIDTH) )    )  ;
               end if;
            end process p_MultAdd;
end generate;

CIN_SELECT1: if (g_C_CIN_Select = 1 and g_ADD = 0 ) generate
      p_MultAdd: process (nreset_i, clock_i)
            begin
               if(nreset_i =  s_reset_state) then
                s_P_Reg       <= (others=>'0');
               elsif(rising_edge(clock_i)) then
               s_P_Reg <= std_logic_vector((resize(signed(s_CIN_Reg),g_P_WIDTH)) +
(resize(signed(s_Mult), g_P_WIDTH)));
               end if;
            end process p_MultAdd;
end generate;

 P_o <= s_P_Reg;

end architecture_preadd_mult_adder;
```

SRS (RTL) and SRM (Technology) Views



Resource Usage

The resource usage report shows that the logic is implemented in one Math block.

Mapping to part: pa5m300fbga896std

Sequential Cells:
SLE 0 uses

DSP Blocks:1
MACC_PA: 1 Mult

Total LUTs:0

Example 38: Math Block with Pre-Adder with C input and dynamic Add/Sub

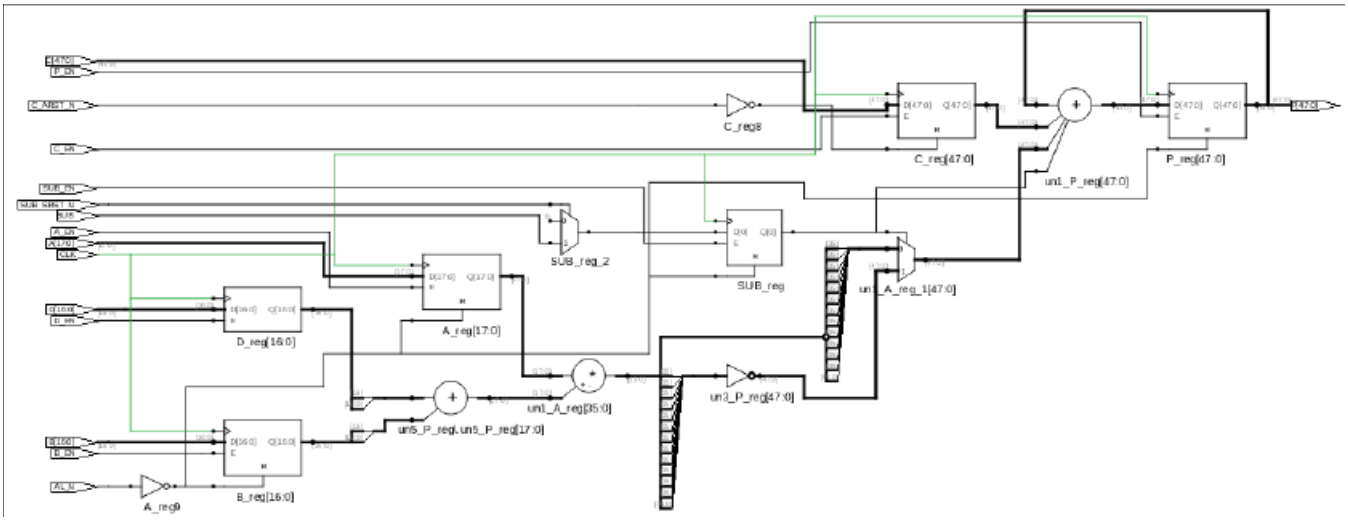
This example shows packing of pre-Adder with C input and dynamic Add/Sub control.

RTL

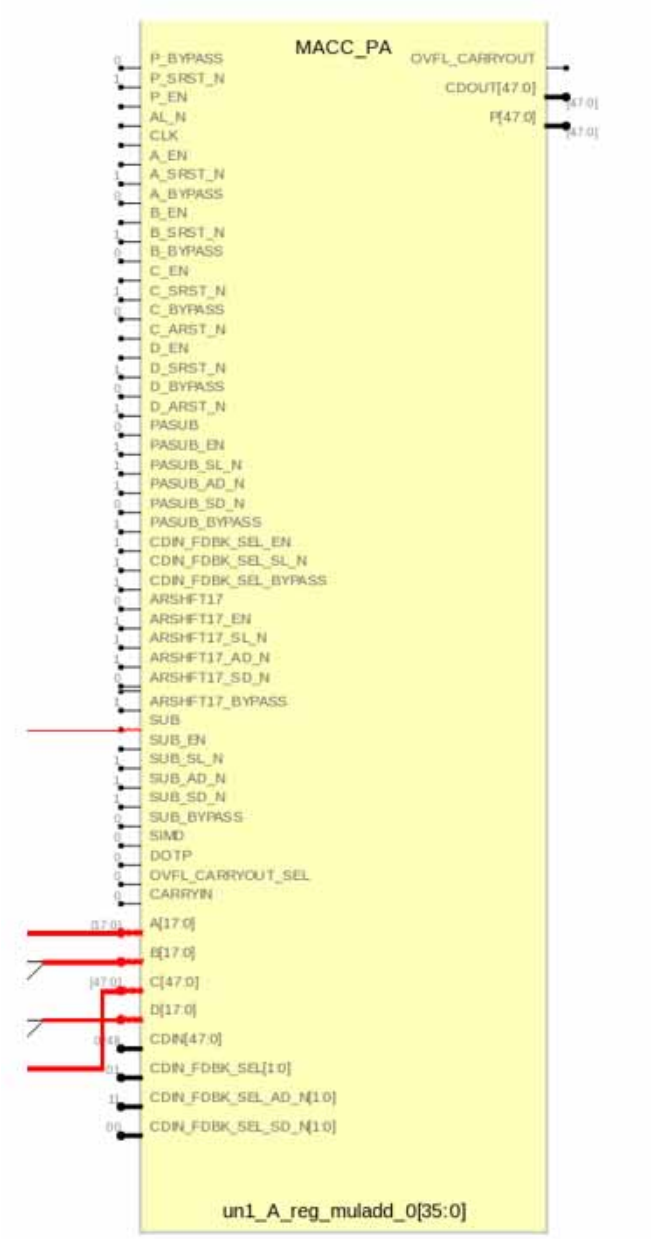
```
module maccpa_arst_en_subregctrl(
   input                CLK,
   input                AL_N,
   input  signed [17:0] A,
   input                A_EN,
   input  signed [16:0] B,
   input                B_EN,
   input  signed [47:0] C,
   input                C_ARST_N,
   input                C_EN,
   input  signed [16:0] D,
   input                D_EN,
   input                SUB,
   input                SUB_EN,
   input                SUB_SRST_N,
   input                P_EN,
   output signed [47:0] P
);

reg signed [17:0] A_reg;
reg signed [16:0] B_reg, D_reg;
reg signed [47:0] C_reg, P_reg;
reg SUB_reg;

always @( posedge CLK or negedge AL_N )
begin
   if ( ~AL_N )
   begin
      A_reg <= 18'b0;
      B_reg <= 17'b0;
      P_reg <= 48'b0;
   end
   else
   begin
      if ( A_EN )
         A_reg <= A;

      if ( B_EN )
      B_reg <= B;

      if ( P_EN )
      begin
      if ( SUB_reg )
         P_reg <= P_reg + C_reg - ((B_reg + D_reg) * A_reg);
      else
         P_reg <= P_reg + C_reg + ((B_reg + D_reg) * A_reg);
      end
   end
end
```

```
always @( posedge CLK or negedge C_ARST_N )
begin
   if ( ~C_ARST_N )
      C_reg <= 48'b0;
      else if ( C_EN )
         C_reg <= C;
end

always @( posedge CLK )
begin
   if ( D_EN )
      D_reg <= D;
end

always @( posedge CLK or negedge AL_N )
begin
   if ( ~AL_N )
      SUB_reg <= 1'b0;
   else if ( SUB_EN )
   SUB_reg <= SUB_SRST_N ? SUB : 1'b0;
end

assign P = P_reg;

endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

The resource usage report shows that the logic is implemented in 1 Math block.

Mapping to part: pa5m300fbga896std

Sequential Cells:
SLE 0 uses

DSP Blocks:1
MACC_PA: 1 Mult

Total LUTs: 0

Inferring MACC_PA_BC_ROM Block for Multiplier

MACC_PA_BC_ROM primitive supports packaging of the Coefficient 16x18 ROM logic structure connected as A input to a MACC block. MACC_PA_BC_ROM block supports 17x17 unsigned operations and 18x18 signed operations.

The following examples show how to infer MACC_PA_BC_ROM blocks for simple multiplier test cases:

- Example 39: 17x17 Unsigned Multiplier, on page 103
- Example 40: 18x18 Signed Multiplier, on page 105
- Example 41: 43x17 Wide Unsigned Multiplier with Output Registered, on page 106
- Example 42: 43x18 Wide Signed Multiplier, on page 108
- Example 43: RTL Coding Style for Simple MULT, on page 109
- Example 44: VHDL RTL Coding Style for Simple MULT, on page 112
- Example 45: RTL Coding Style with Case Statement for Simple MULT with 18-bit Signed Input, on page 114

Example 39: 17x17 Unsigned Multiplier

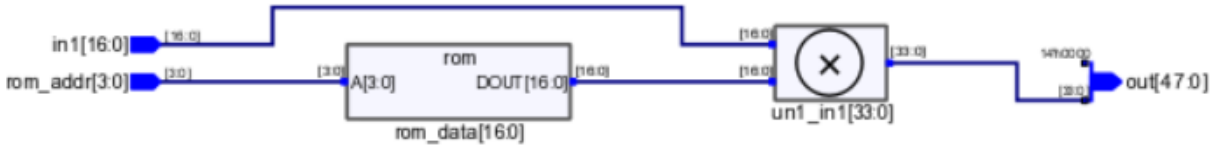
The following design is a simple 17x17-bit unsigned multiplier that the tool maps to Coeff ROM Math block as shown in the subsequent figure.

RTL

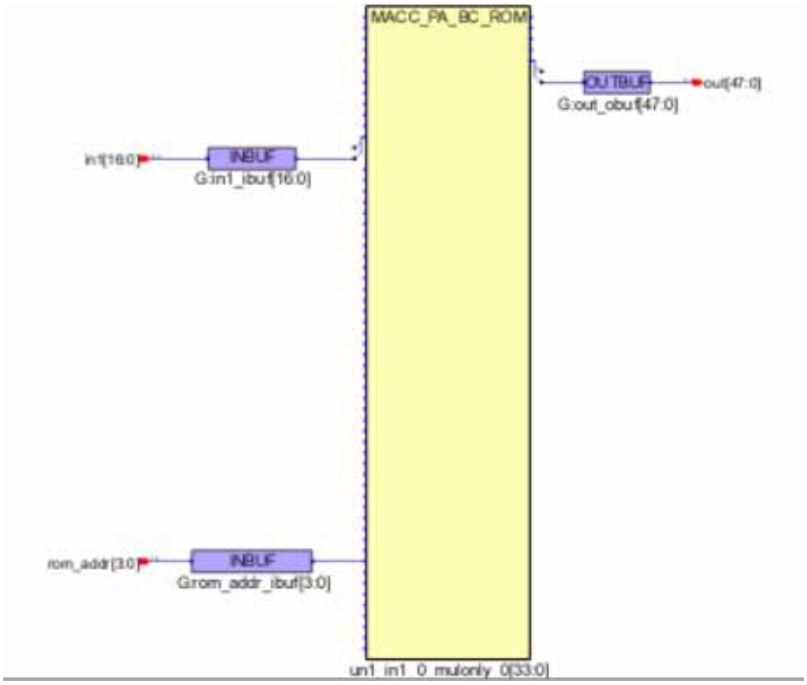
```
module test(in1, rom_addr, out);

input [16:0] in1;
input [3:0] rom_addr;
output [47:0] out;

reg [16:0] in2 [0:15];
initial

begin
   $readmemb("mem.dat",in2);
end

wire [16:0] rom_data;
assign rom_data = in2[rom_addr];
assign out =  rom_data * in1;
```

```
endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

The resource usage report shows that the logic is implemented in 1 Math block.

Sequential Cells:
SLE 0 uses
DSP Blocks: 1 of 336 (0%)
MACC_PA_BC_ROM: 1 Mult

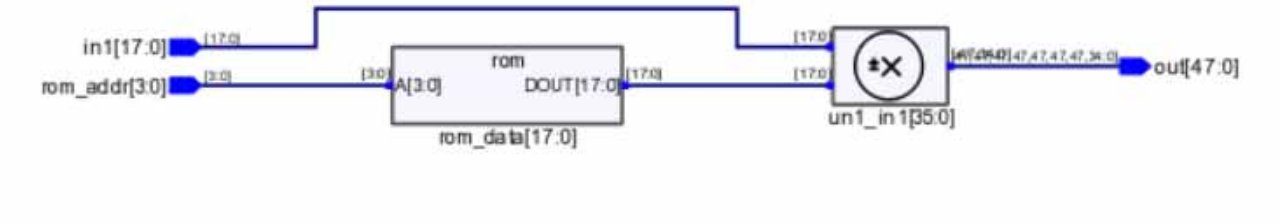
Example 40: 18x18 Signed Multiplier

The following design is a simple 18x18-bit signed multiplier that the tool maps to Coeff ROM Math block as shown in the subsequent figure.

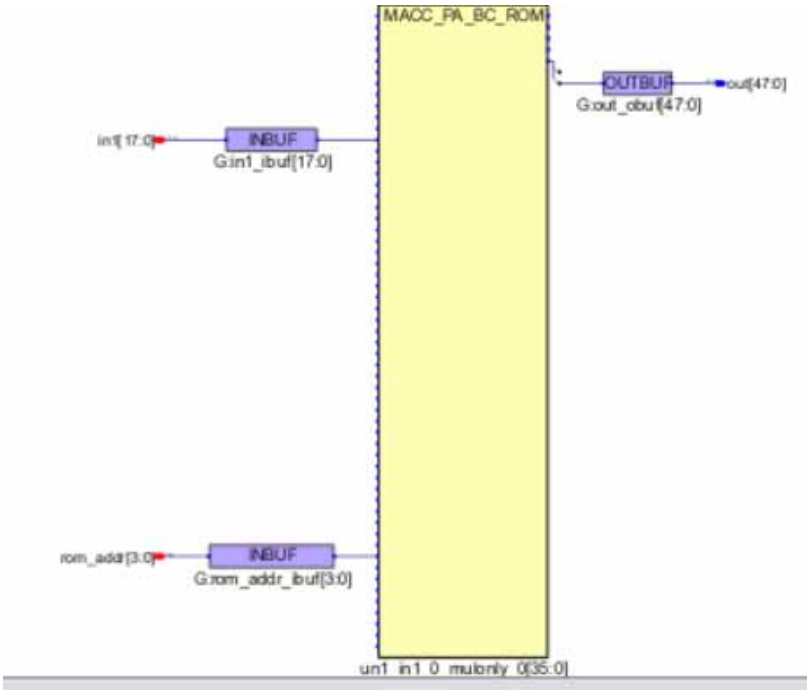
RTL

```
module test(in1, rom_addr, out);

input signed [17:0] in1;
input [3:0] rom_addr;
output signed [47:0] out;

reg signed [17:0] in2 [0:15];

initial
begin
   $readmemb("mem.dat",in2);
end

wire signed [17:0] rom_data;
assign rom_data = in2[rom_addr];
assign out =  rom_data * in1;

endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

The resource usage report shows that the logic is implemented in 1 Math block.

Sequential Cells:
SLE 0 uses
DSP Blocks: 1 of 336 (0%)
MACC_PA_BC_ROM: 1 Mult

Example 41: 43x17 Wide Unsigned Multiplier with Output Registered

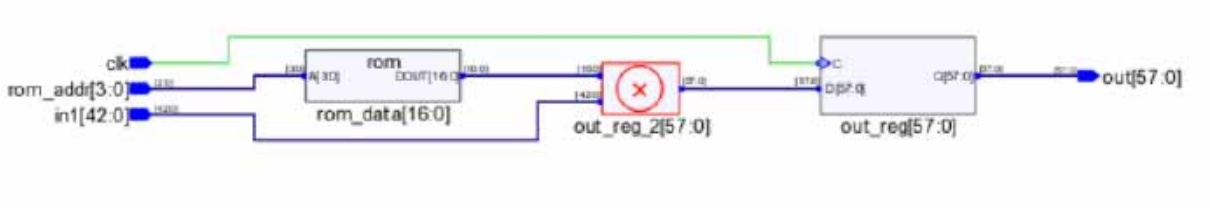
The following design is a simple 43x17-bit unsigned multiplier that the tool maps to Coeff ROM Math blocks as shown in the subsequent figure.

RTL

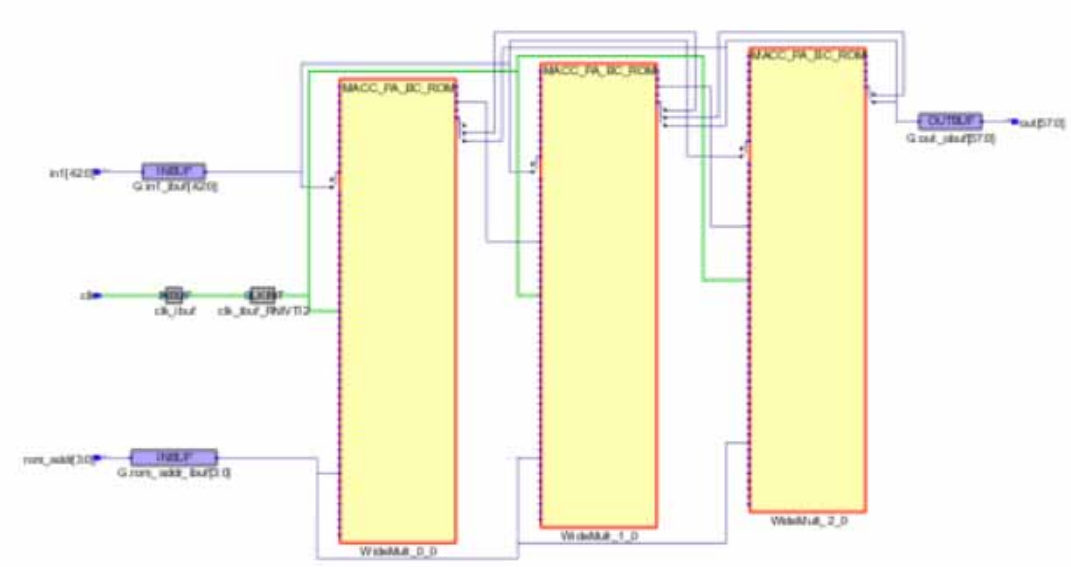
```
module test(clk, in1, rom_addr, out);

input clk;
input [42:0] in1;
input [3:0] rom_addr;
output [57:0] out;

reg [57:0] out_reg;
reg [16:0] mem [0:2**4 -1];

wire [16:0] rom_data;
```

```
initial
begin
   $readmemb("mem.dat",mem);
end

always@(posedge clk)

begin
   out_reg <=  rom_data * in1;
end

assign rom_data = mem[rom_addr];
assign out = out_reg;

endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

The resource usage report shows that the logic is implemented in 3 Math blocks.

Sequential Cells:
SLE 0 uses
DSP Blocks: 3 of 336 (0%)
MACC_PA_BC_ROM: 1 Mult
MACC_PA_BC_ROM: 2 MultAdds

Example 42: 43x18 Wide Signed Multiplier

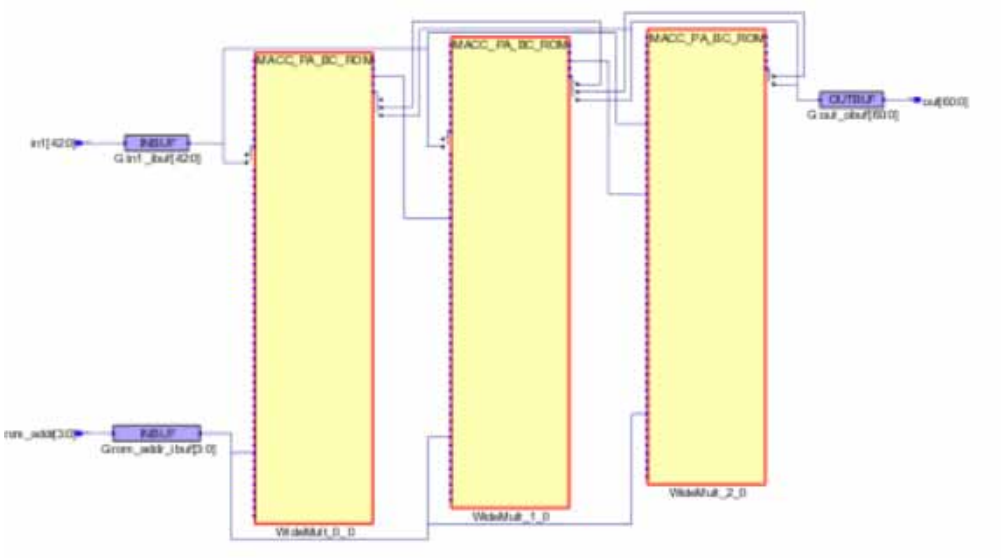
The following design is a simple 43x18-bit signed multiplier that the tool maps to Coeff ROM Math blocks as shown in the subsequent figure.

RTL

```
module test(in1, rom_addr, out);

input signed [42:0] in1;
input [3:0] rom_addr;
output signed [60:0] out;

reg signed [17:0] in2 [0:15];

initial
begin
   $readmemb("mem.dat",in2);
end

wire signed [17:0] rom_data;

assign rom_data = in2[rom_addr];
assign out =  rom_data * in1;

endmodule
```

SRS (RTL) View

SRM (Technology) View



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 3 of 336 (0%)
MACC_PA_BC_ROM: 1 Mult
MACC_PA_BC_ROM: 2 MultAdds

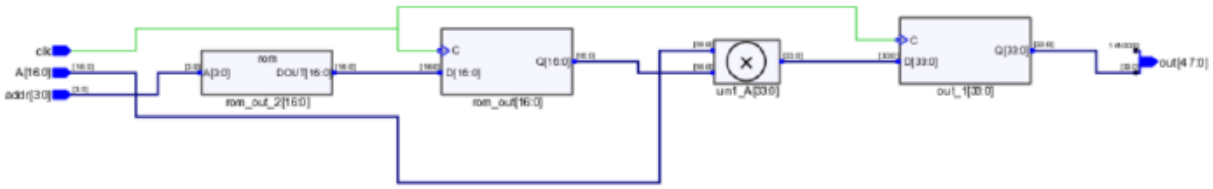
Example 43: RTL Coding Style for Simple MULT

RTL

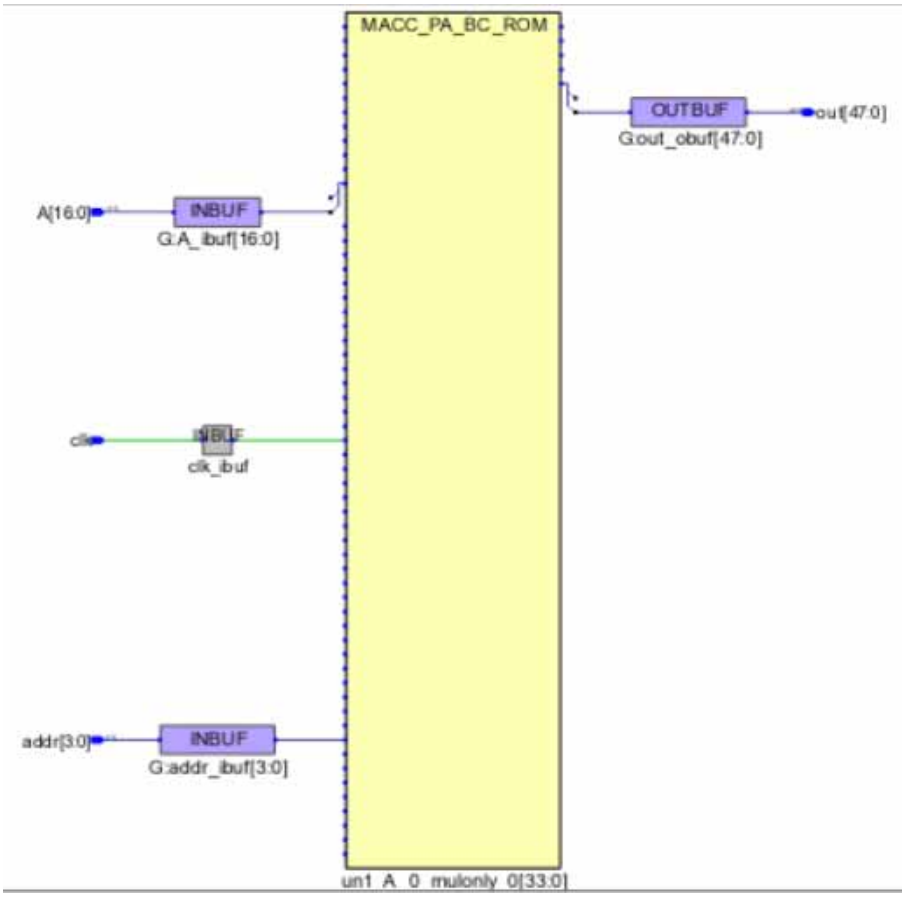
```
module simple_mult(clk,A,out,addr );

input clk;
input [16:0] A;
input [3:0] addr;
output [33:0] out;

reg [16:0] rom_out;
reg [33:0] out;

always@(posedge clk)

begin
case (addr)
   4'd0 : rom_out <= 17'b01100000100011001;
   4'd1 : rom_out <= 17'b00001101100111001;
   4'd2 : rom_out <= 17'b01011011110110011;
   4'd3 : rom_out <= 17'b10100110000000001;
   4'd4 : rom_out <= 17'b01101001111111001;
   4'd5 : rom_out <= 17'b01010000100010101;
```

```
      4'd6 : rom_out <= 17'b10101101010001101;
      4'd7 : rom_out <= 17'b01001001010010101;
      4'd8 : rom_out <= 17'b01110111000111111;
      4'd9 : rom_out <= 17'b10010101111110110;
      4'd10 : rom_out <= 17'b01000110000010001;
      4'd11 : rom_out <= 17'b11100000110110011;
      4'd12 : rom_out <= 17'b10110101101111011;
      4'd13 : rom_out <= 17'b01111010011000001;
      4'd14 : rom_out <= 17'b00110110100111110;
      4'd15 : rom_out <= 17'b11110101000010001;
   default : rom_out <= 17'b00010011011010101;
   endcase

   out <= rom_out * A;

   end
   endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 Mult
Total LUTs: 0

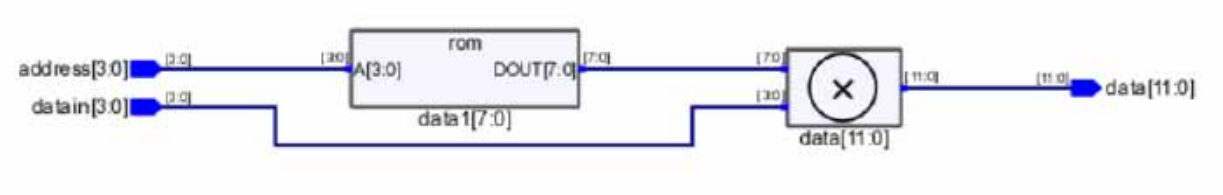
Example 44: VHDL RTL Coding Style for Simple MULT

RTL

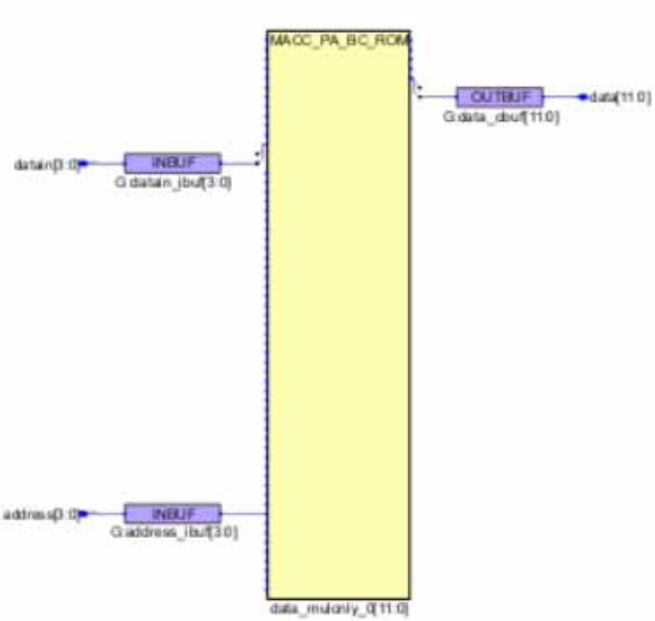
```
library ieee;

use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mult is
port ( address : in std_logic_vector(3 downto 0);
         datain : in std_logic_vector(3 downto 0);
         data : out std_logic_vector(11 downto 0) );
end entity mult;

architecture behavioral of mult is
   signal data1 : std_logic_vector(7 downto 0);
   type mem is array ( 0 to 2**4 - 1) of std_logic_vector(7 downto 0);
   constant my_Rom : mem := (
      0  => "00000000",
      1  => "00000001",
      2  => "00000010",
      3  => "00000011",
      4  => "00000100",
      5  => "11010000",
      6  => "01110000",
      7  => "10110000",
      8  => "01110000",
      9  => "11110000",
      10 => "11110100",
      11 => "11110010",
      12 => "11110001",
      13 => "11111000",
      14 => "11110100",
      15 => "11110010");
begin
   process (address)
   begin
      case address is
         when "0000" => data1 <= my_rom(0);
         when "0001" => data1 <= my_rom(1);
         when "0010" => data1 <= my_rom(2);
         when "0011" => data1 <= my_rom(3);
         when "0100" => data1 <= my_rom(4);
         when "0101" => data1 <= my_rom(5);
         when "0110" => data1 <= my_rom(6);
         when "0111" => data1 <= my_rom(7);
         when "1000" => data1 <= my_rom(8);
         when "1001" => data1 <= my_rom(9);
         when "1010" => data1 <= my_rom(10);
         when "1011" => data1 <= my_rom(11);
         when "1100" => data1 <= my_rom(12);
         when "1101" => data1 <= my_rom(13);
         when "1110" => data1 <= my_rom(14);
         when "1111" => data1 <= my_rom(15);
```

```
            when others => data1 <= "00000000";
      end case;
   end process;
      data <= std_logic_vector(data1 * datain);
   end architecture behavioral;
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

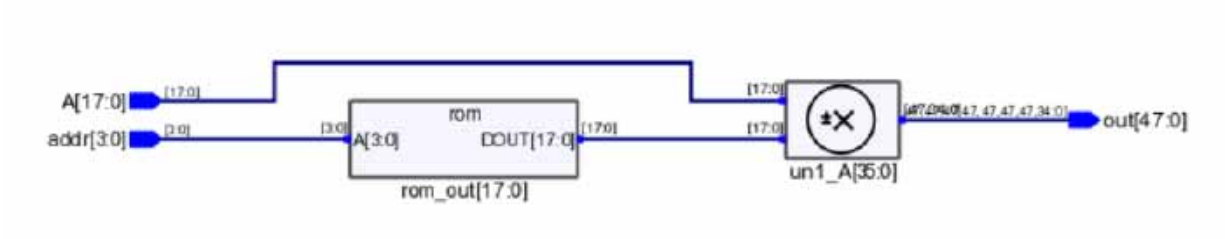
SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 Mult
Total LUTs: 0

Example 45: RTL Coding Style with Case Statement for Simple MULT with 18-bit Signed Input

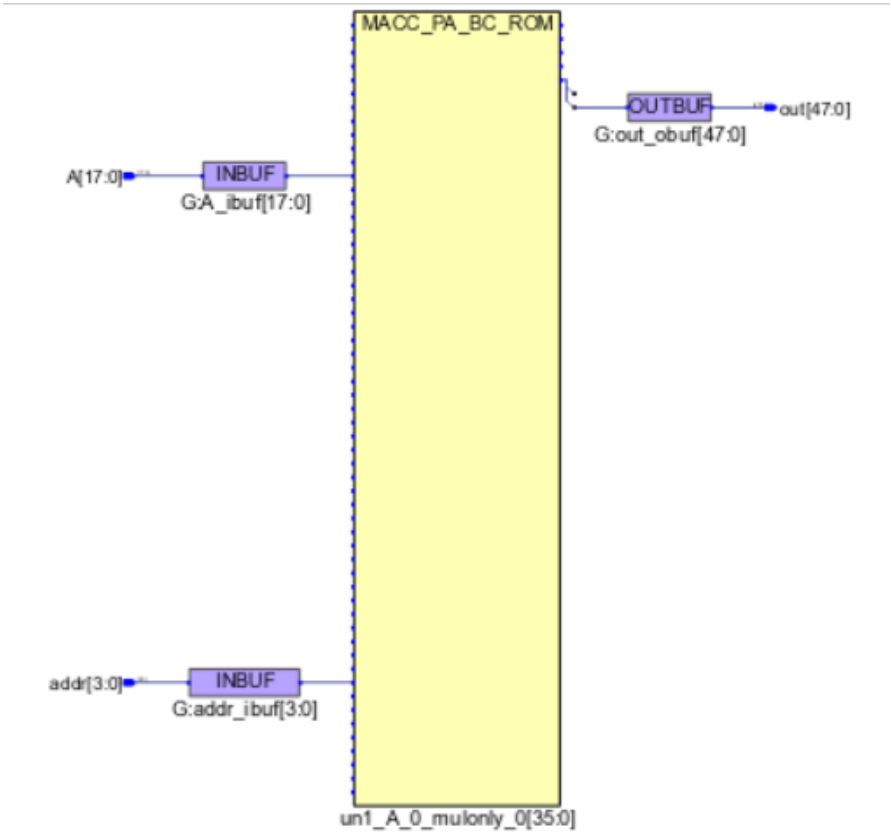
RTL

```
module mult_signed(A,out,addr );

input signed [17:0] A;
input signed [3:0] addr;
output signed [47:0] out;

reg signed [17:0] rom_out;
reg signed [47:0] out;

always@(addr or A)

begin
case (addr)

   4'd0 : rom_out <=   18'b011000001000110010;
   4'd1 : rom_out <=   18'b000011011001110011;
   4'd2 : rom_out <=   18'b010110111101100111;
   4'd3 : rom_out <=   18'b101001100000000010;
   4'd4 : rom_out <=   18'b011010011111110011;
   4'd5 : rom_out <=   18'b001010000100010101;
   4'd6 : rom_out <=   18'b010101101010001101;
   4'd7 : rom_out <=   18'b001001001010010101;
   4'd8 : rom_out <=   18'b001110111000111111;
   4'd9 : rom_out <=   18'b010010101111110110;
   4'd10 : rom_out <=  18'b001000110000010001;
   4'd11 : rom_out <=  18'b011100000110110011;
   4'd12 : rom_out <=  18'b010110101101111011;
   4'd13 : rom_out <=  18'b001111010011000001;
   4'd14 : rom_out <=  18'b100110110100111110;
   4'd15 : rom_out <=  18'b111110101000010001;
   default:rom_out <=  18'b000100101101010111;

endcase

out <= rom_out * A;

end
endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 Mult
Total LUTs: 0

Inferring MACC_PA_BC_ROM Block for Mult-Add/Mult-Sub/Mult-Acc

The following examples show how to infer MACC_PA_BC_ROM blocks for simple multiplier followed by adder/subtractor/accumulator test cases.

- Example 46: 17x17 Unsigned Mult-Add, on page 116
- Example 47: 17x7 Signed Mult-Sub, on page 118
- Example 48: 17x11 Unsigned Mult-Acc, on page 119
- Example 49: 17x17 Signed Mult-Acc-Add, on page 121
- Example 50: RTL Coding Style with Case Statement for Simple Mult-Add, on page 123
- Example 51: VHDL RTL Coding Style for Simple Mult-Add, on page 126
- Example 52: RTL Coding Style with Case Statement for Simple Mult-Sub, on page 128
- Example 53: RTL Coding Style with Case Statement for Simple Mult-Adder with Registered Inputs with Synchronous Reset, on page 130

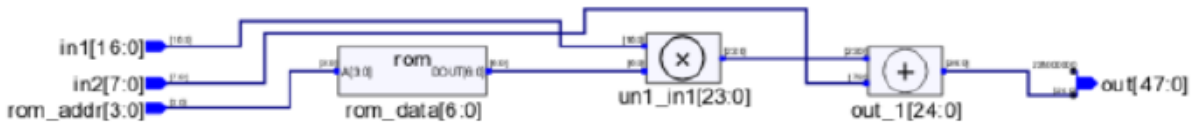
Example 46: 17x17 Unsigned Mult-Add

This example shows how the output of a multiplier is added to another input.

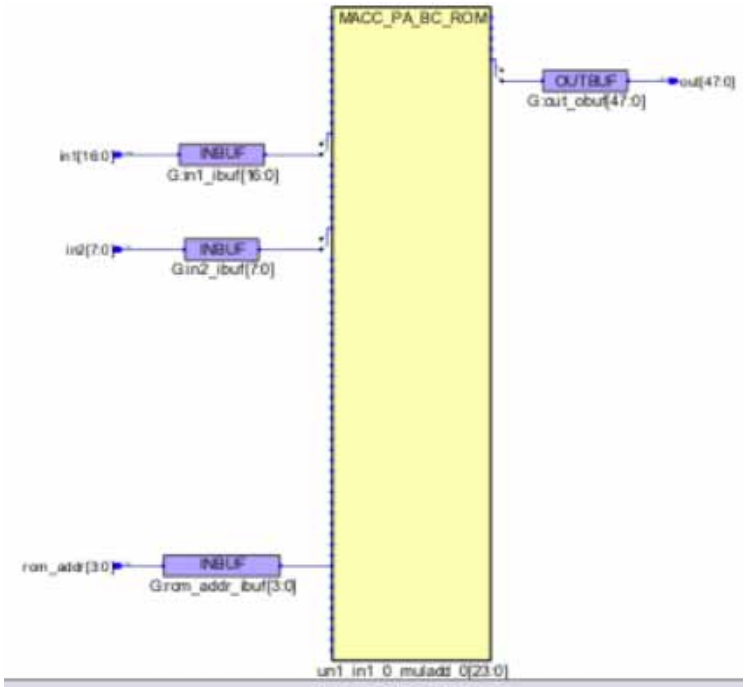
RTL

```
module test(in1,in2, rom_addr, out);

input [16:0] in1;
input [7:0] in2;
input [3:0] rom_addr;
output  [47:0] out;

wire [49:0] out1;
reg [6:0] mem [0:2**4 -1];

initial

begin
   $readmemb("mem.dat",mem);
end

wire [6:0] rom_data;

assign rom_data = mem[rom_addr];
assign out1 =  rom_data * in1;
assign out = out1 + in2;

endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 MultAdd

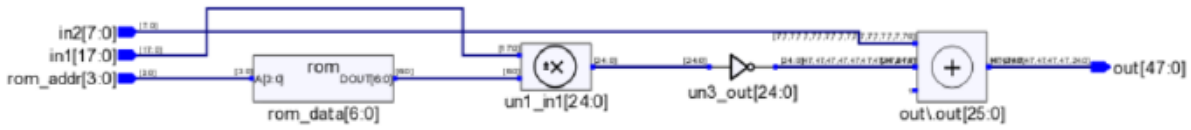
Example 47: 17x7 Signed Mult-Sub

This example shows how the output of a multiplier is subtracted from another input.

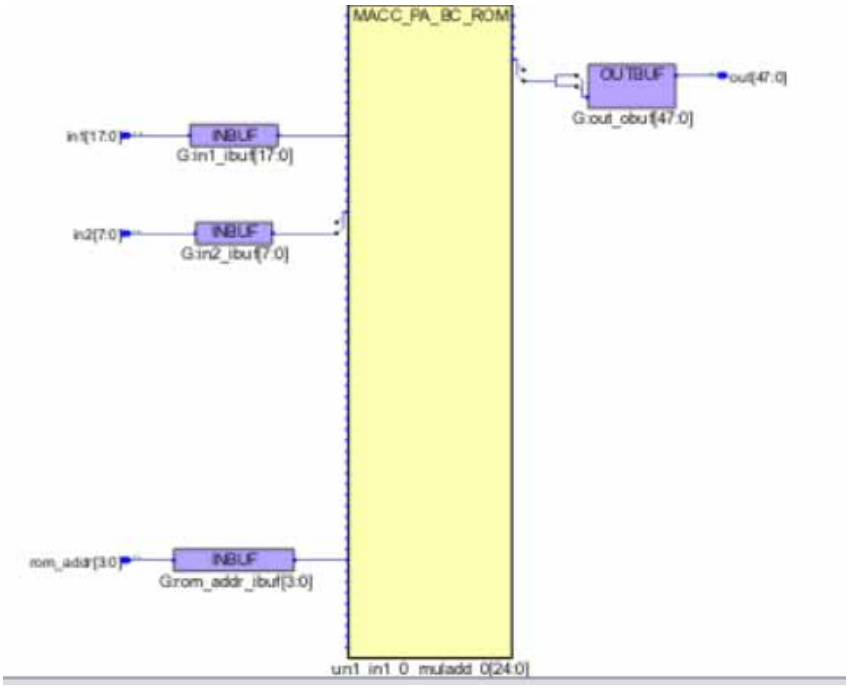
RTL

```
module test(in1, in2, rom_addr, out);

input signed [17:0] in1;
input signed [7:0] in2;
input [3:0] rom_addr;
output signed [47:0] out;

wire signed [49:0] out1;
reg [6:0] mem [0:2**4 -1];

initial
begin
   $readmemb("mem.dat",mem);
end

wire signed [6:0] rom_data;

assign rom_data = mem[rom_addr];
assign out1 =  rom_data * in1;
assign out = in2 - out1;

endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 MultSub

Example 48: 17x11 Unsigned Mult-Acc

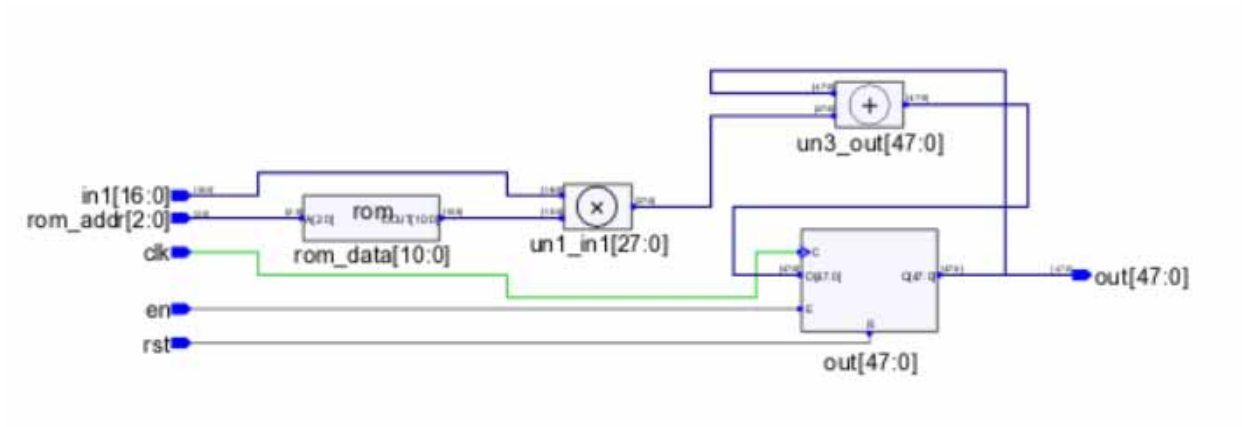
This example shows a Mult-Acc that the tool maps to Coeff ROM MATH block.

RTL

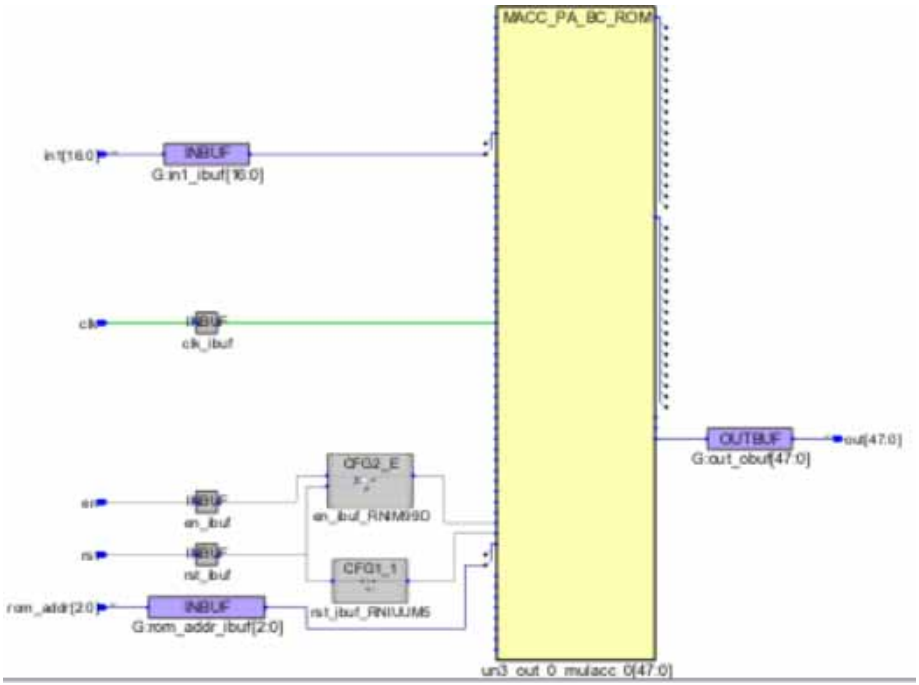
```
module test(clk, rst, en, in1, rom_addr, out);

input [16:0] in1;
input [2:0] rom_addr;
input clk,rst,en;
output reg [47:0] out;

wire  [47:0] out1;

reg [10:0] mem [0:2**3 -1];

initial
begin
   $readmemb("mem.dat",mem);
end
```

```
wire [10:0] rom_data;

assign rom_data = mem[rom_addr];
assign out1 =  rom_data * in1;

always@(posedge clk)
begin
   if(rst)
      out <= 0;
   else if(en)
      out <= out1 + out;

end
endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 1 of 336 (0%)
MACC_PA_BC_ROM: 1 MultAcc

Example 49: 17x17 Signed Mult-Acc-Add

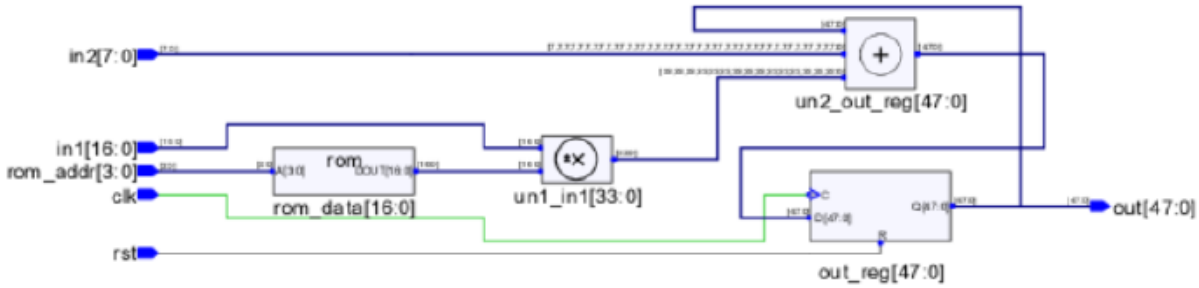
This example shows a Mult-Acc that the tool maps to Coeff ROM MATH block.

RTL

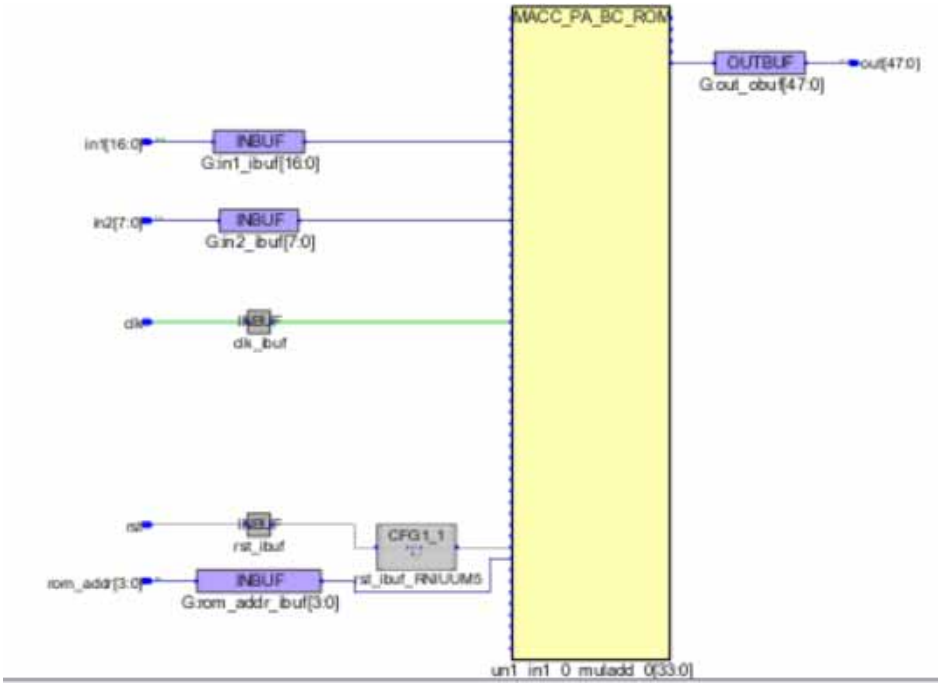
```
module test(clk, rst, in1, in2, rom_addr, out);

input signed [16:0] in1;
input signed [7:0] in2;
input [3:0] rom_addr;
input clk,rst;

output signed [47:0] out;
reg signed [47:0] out_reg;

wire signed  [47:0] out1;
reg [16:0] mem [0:2**4 -1];
```

```
wire signed [16:0] rom_data;
initial
begin
   $readmemb("mem.dat",mem);
end

assign rom_data = mem[rom_addr];
assign out1 =  rom_data * in1;
always@(posedge clk)
begin
   if(rst)
      out_reg <= 0;
   else
      out_reg <= out1 + out_reg + in2;

end
assign out = out_reg;
endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 1 of 336 (0%)
MACC_PA_BC_ROM: 1 MultAdd

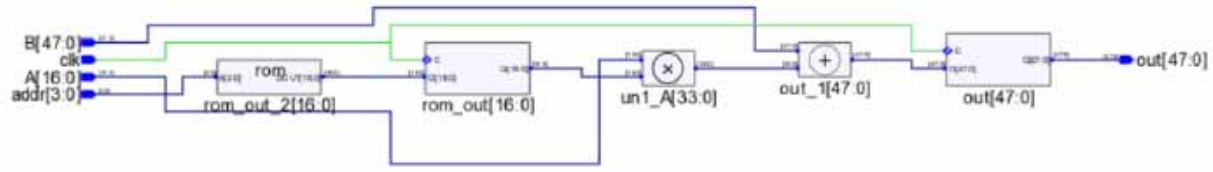
Example 50: RTL Coding Style with Case Statement for Simple Mult-Add

RTL

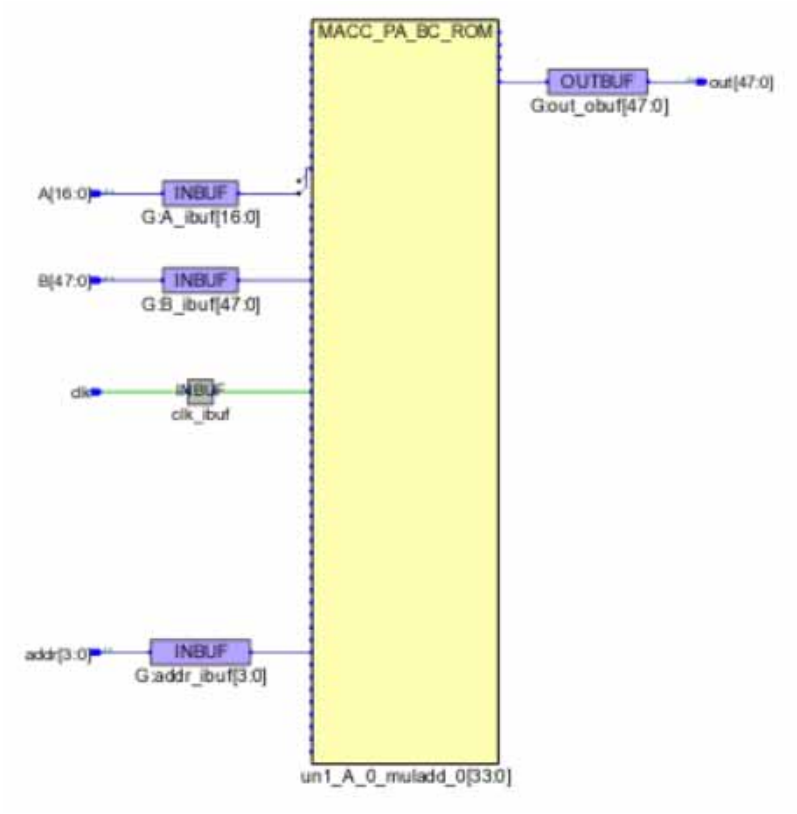
```
module multadd(clk,A,B,out,addr );

input clk;
input [16:0] A;
input [47:0] B;
input [3:0] addr;
output [47:0] out;

reg [16:0] rom_out;
reg [47:0] out;
```

```
always@(posedge clk)
begin
case (addr)
   4'd0 : rom_out <= 17'b01100000100011001;
   4'd1 : rom_out <= 17'b00001101100111001;
   4'd2 : rom_out <= 17'b01011011110110011;
   4'd3 : rom_out <= 17'b10100110000000001;
   4'd4 : rom_out <= 17'b01101001111111001;
   4'd5 : rom_out <= 17'b01010000100010101;
   4'd6 : rom_out <= 17'b10101101010001101;
   4'd7 : rom_out <= 17'b01001001010010101;
   4'd8 : rom_out <= 17'b01110111000111111;
   4'd9 : rom_out <= 17'b10010101111110110;
   4'd10 : rom_out <= 17'b01000110000010001;
   4'd11 : rom_out <= 17'b11100000110110011;
   4'd12 : rom_out <= 17'b10110101101111011;
   4'd13 : rom_out <= 17'b01111010011000001;
   4'd14 : rom_out <= 17'b00110110100111110;
   4'd15 : rom_out <= 17'b11110101000010001;
   default : rom_out <= 17'b00010010011010101;

endcase
out <= (rom_out * A)+ B ;
end
endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 MultAdd
Total LUTs: 0

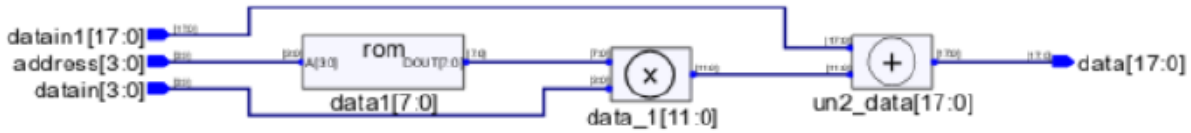
Example 51: VHDL RTL Coding Style for Simple Mult-Add

RTL

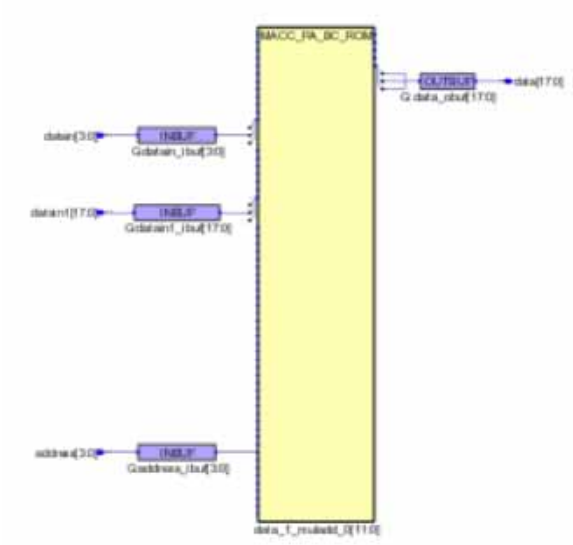
```
library ieee;

use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mult is
port ( address : in std_logic_vector(3 downto 0);
       datain : in std_logic_vector(3 downto 0);
       datain1 : in std_logic_vector(17 downto 0);
data : out std_logic_vector(17 downto 0) );
end entity mult;

architecture behavioral of mult is
signal data1 : std_logic_vector(7 downto 0);
type mem is array ( 0 to 2**4 - 1) of std_logic_vector(7 downto 0);
constant my_Rom : mem := (
   0  => "00000000",
   1  => "00000001",
   2  => "00000010",
   3  => "00000011",
   4  => "00000100",
   5  => "11010000",
   6  => "01110000",
   7  => "10110000",
   8  => "01110000",
   9  => "11110000",
   10 => "11110100",
   11 => "11110010",
   12 => "11110001",
   13 => "11111000",
   14 => "11110100",
   15 => "11110010");

begin
   process (address)
      begin
         case address is
            when "0000" => data1 <= my_rom(0);
            when "0001" => data1 <= my_rom(1);
            when "0010" => data1 <= my_rom(2);
            when "0011" => data1 <= my_rom(3);
            when "0100" => data1 <= my_rom(4);
            when "0101" => data1 <= my_rom(5);
            when "0110" => data1 <= my_rom(6);
            when "0111" => data1 <= my_rom(7);
            when "1000" => data1 <= my_rom(8);
            when "1001" => data1 <= my_rom(9);
            when "1010" => data1 <= my_rom(10);
            when "1011" => data1 <= my_rom(11);
            when "1100" => data1 <= my_rom(12);
            when "1101" => data1 <= my_rom(13);
```

```
            when "1110" => data1 <= my_rom(14);
            when "1111" => data1 <= my_rom(15);
            when others => data1 <= "00000000";
         end case;
end process;

data <= std_logic_vector(data1 * datain + datain1);
end architecture behavioral;
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

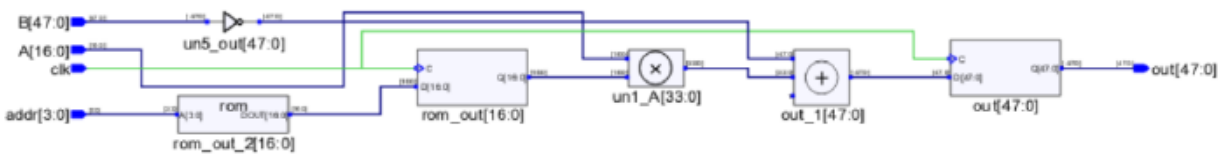
SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 MultAdd
Total LUTs: 0

Example 52: RTL Coding Style with Case Statement for Simple Mult-Sub

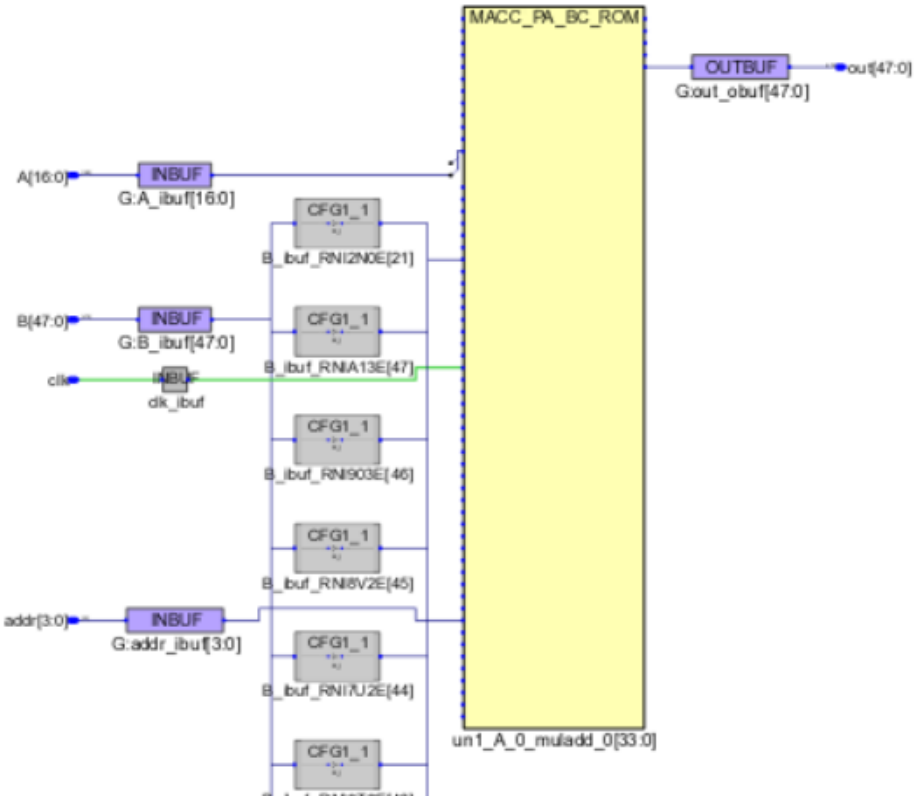
RTL

```
module multsub(clk,A,B,out,addr );

input clk;
input [16:0] A;
input [47:0] B;
input [3:0] addr;
output [47:0] out;

reg [16:0] rom_out;
reg [47:0] out;

always@(posedge clk)
   begin
      case (addr)
         4'd0 : rom_out <= 17'b01100000100011001;
         4'd1 : rom_out <= 17'b00001101100111001;
         4'd2 : rom_out <= 17'b01011011110110011;
         4'd3 : rom_out <= 17'b10100110000000001;
         4'd4 : rom_out <= 17'b01101001111111001;
         4'd5 : rom_out <= 17'b01010000100010101;
         4'd6 : rom_out <= 17'b10101101010001101;
         4'd7 : rom_out <= 17'b01001001010010101;
         4'd8 : rom_out <= 17'b01110111000111111;
         4'd9 : rom_out <= 17'b10010101111110110;
         4'd10 : rom_out <= 17'b01000110000010001;
         4'd11 : rom_out <= 17'b11100000110110011;
         4'd12 : rom_out <= 17'b10110101101111011;
         4'd13 : rom_out <= 17'b01111010011000001;
         4'd14 : rom_out <= 17'b00110110100111110;
         4'd15 : rom_out <= 17'b11110101000010001;
         default : rom_out <= 17'b00010010111010101;
      endcase
out <= (rom_out * A) - B ;
end
endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 MultAdd
Total LUTs: 48

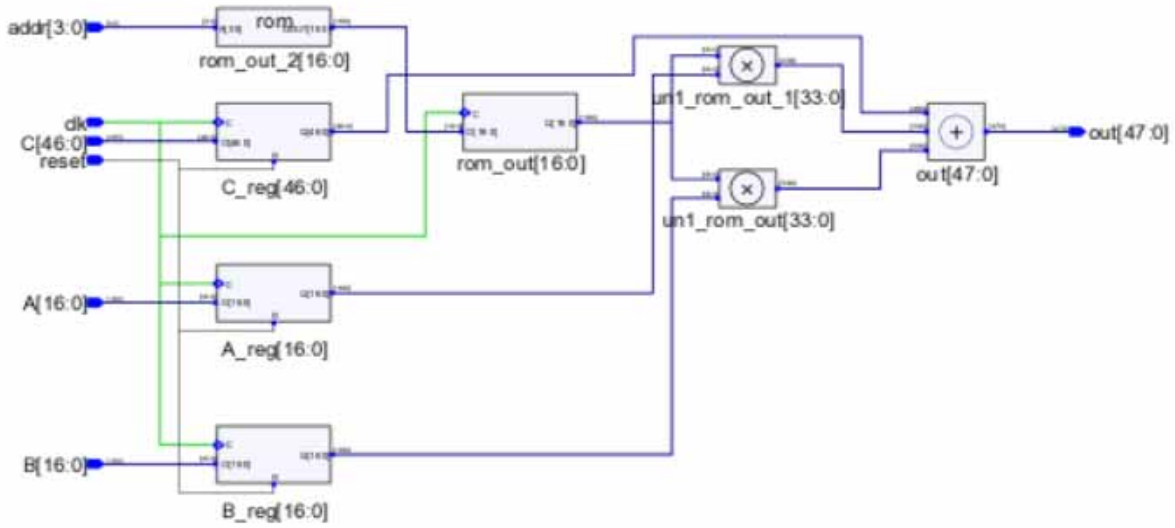
Example 53: RTL Coding Style with Case Statement for Simple Mult-Adder with Registered Inputs with Synchronous Reset

RTL

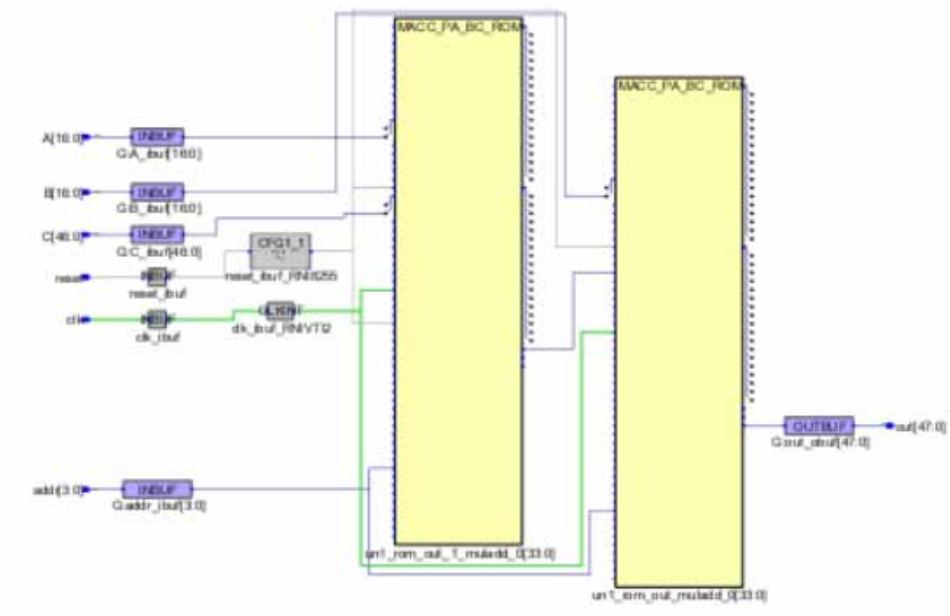
```
module multadd_synrst(clk,A,B,C,out,addr,reset);
input clk,reset;
input unsigned [16:0] A;
input unsigned   [16:0] B;
input unsigned [46:0] C;
input [3:0] addr;
output [47:0] out;

reg [16:0] rom_out;
wire [47:0] out;

reg unsigned [17:0] A_reg;
reg unsigned [17:0] B_reg;
reg unsigned [46:0] C_reg;

always@(posedge clk)
   begin
      case (addr)
         4'd0 : rom_out <= 17'b011000001000110010;
         4'd1 : rom_out <= 17'b000011011001110010;
         4'd2 : rom_out <= 17'b010110111101100110;
         4'd3 : rom_out <= 17'b101001100000000010;
         4'd4 : rom_out <= 17'b011010011111110010;
         4'd5 : rom_out <= 17'b010100001000101010;
         4'd6 : rom_out <= 17'b101011010100011010;
         4'd7 : rom_out <= 17'b010010010100101010;
         4'd8 : rom_out <= 17'b011101110001111111;
         4'd9 : rom_out <= 17'b100101011111101101;
         4'd10 : rom_out <= 17'b010001100000100011;
         4'd11 : rom_out <= 17'b111000001101100111;
         4'd12 : rom_out <= 17'b101101011011110111;
         4'd13 : rom_out <= 17'b011110100110000011;
         4'd14 : rom_out <= 17'b001101101001111101;
         4'd15 : rom_out <= 17'b111101010000100010;
         default : rom_out <= 17'b000100101101010;
      endcase
end

always@(posedge clk)
begin
   if (reset)
   begin
     A_reg <= 0;
     B_reg <= 0;
     C_reg <= 0;
   end
   else
   begin
     A_reg <= A;
     B_reg <= B;
```

```
      C_reg <= C;
end
end
   assign out = (rom_out * A_reg) + (rom_out * B_reg) + C_reg  ;
endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

SLE 1 uses
DSP Blocks: 2 of 924 (0%)
MACC_PA_BC_ROM: 2 MultAdds
Total LUTs: 1

Inferring MACC_PA_BC_ROM Block for Multiplier with Pre-adder

The following examples show how to infer MACC_PA_BC_ROM blocks for multiplier with pre-adder/pre-sub test cases:

- Example 54: 17x18 Unsigned Multiplier with Pre-adder, on page 132
- Example 55: 16x9 Multiplier with Pre-sub, on page 134
- Example 56: 14x11 Unsigned Mult-Add with Pre-adder, on page 135
- Example 57: 10x17 Unsigned Mult-Acc with Pre-adder, on page 137
- Example 58: 10x17 Unsigned Mult-Acc-Add with Pre-adder, on page 139
- Example 59: RTL Coding Style with Case Statement for Simple Pre-adder Multiplier, on page 141
- Example 60: RTL Coding Style with Case Statement for Simple Pre-Adder Mult Acc with Asynchronous Reset on Output, on page 143

Example 54: 17x18 Unsigned Multiplier with Pre-adder

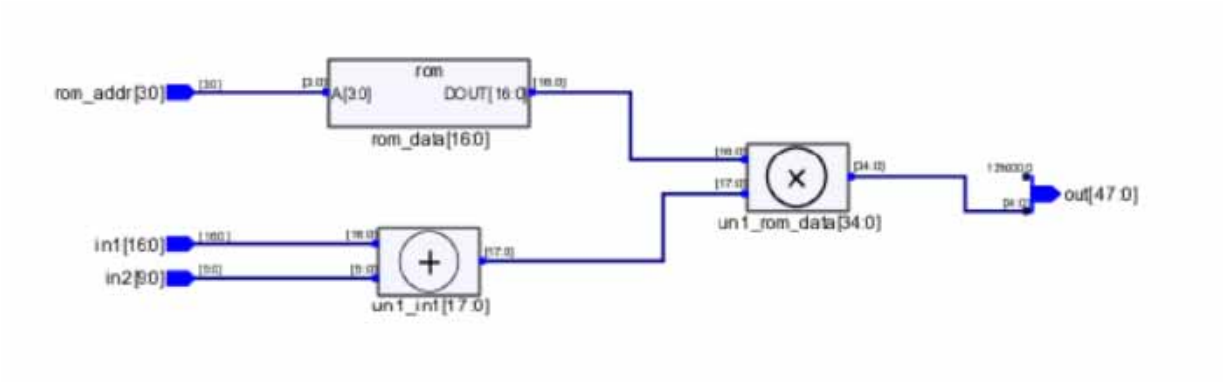
This example shows a Multiplier with Pre-adder that the tool maps to Coeff ROM MATH block.

RTL

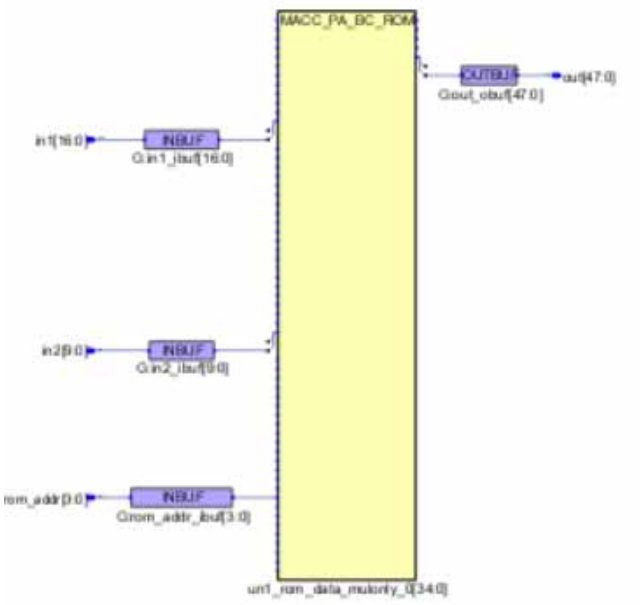
```
module test(in1,in2, rom_addr, out)

input [16:0] in1;
input [9:0] in2;
input [3:0] rom_addr;
output  [47:0] out;

reg [16:0] mem [0:2**4-1];

initial
begin
   $readmemb("mem.dat",mem);
end
```

```
wire [16:0] rom_data;

assign rom_data = mem[rom_addr];
assign out =  rom_data * (in1 + in2);

endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 Mult

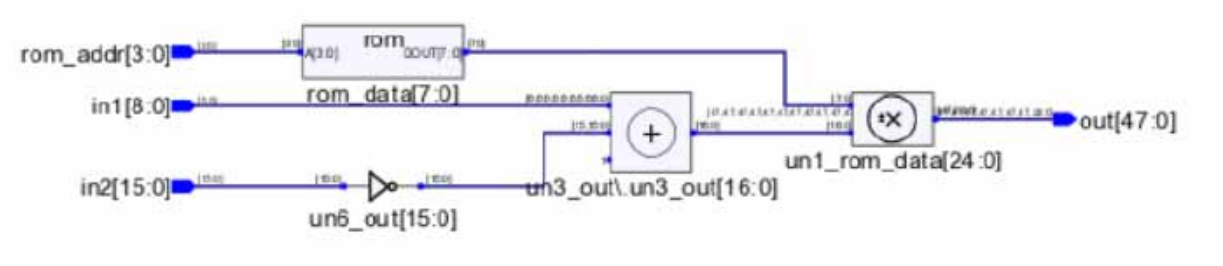
Example 55: 16x9 Multiplier with Pre-sub

This example shows a Multiplier with Pre-sub that the tool maps to Coeff ROM MATH block.

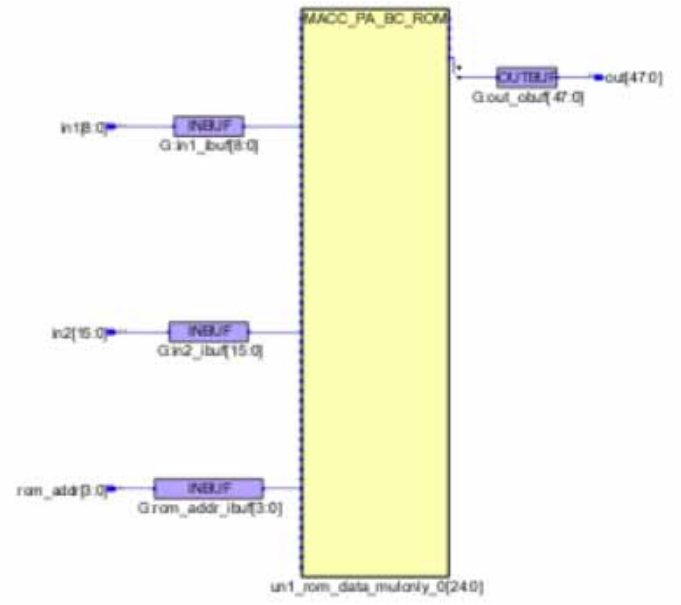
RTL

```
module test(in1,in2, rom_addr, out);

input signed [8:0] in1;
input signed [15:0] in2;
input [3:0] rom_addr;
output  [47:0] out;

reg [7:0] mem [0:2**4 -1];

initial
begin
   $readmemb("mem.dat",mem);
end

wire signed [7:0] rom_data;

assign rom_data = mem[rom_addr];
assign out =  rom_data *  (in1 - in2);

endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 Mult

Example 56: 14x11 Unsigned Mult-Add with Pre-adder

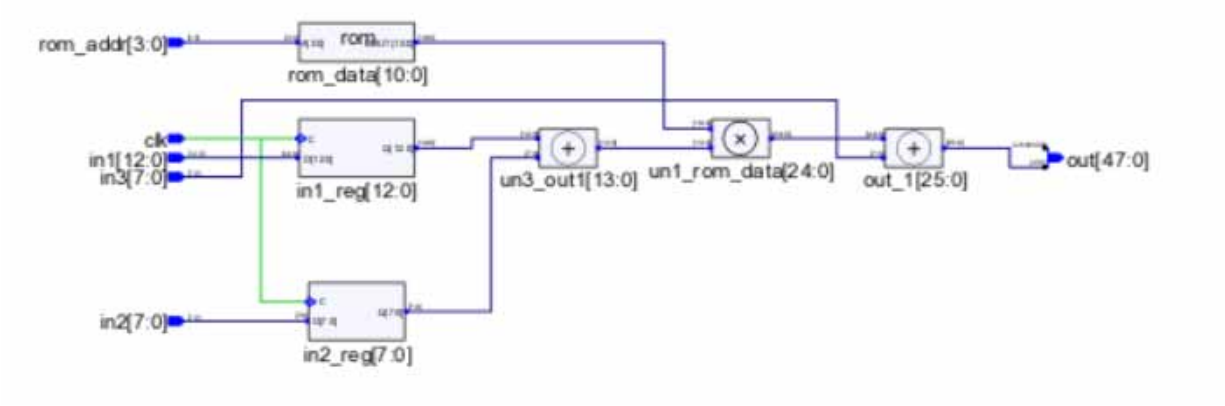
This example shows a Mult-Add with Pre-adder that the tool maps to Coeff ROM MATH block.

RTL

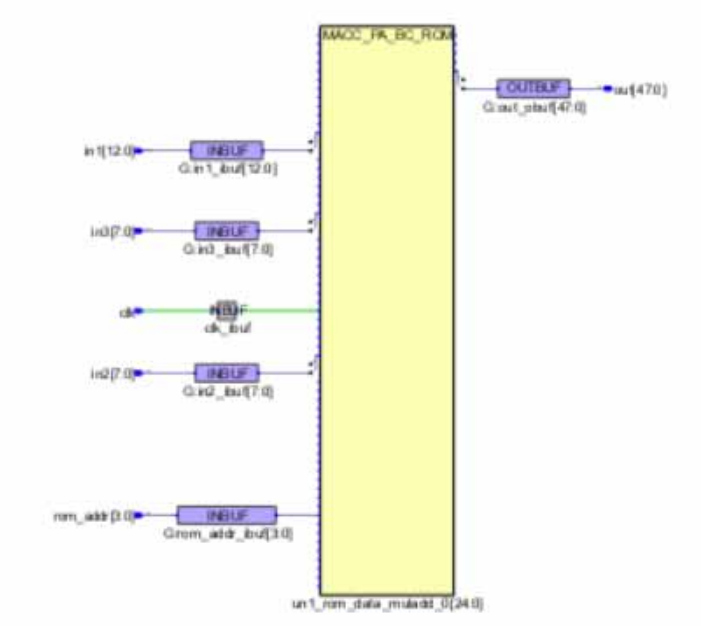
```
module test(clk, in1,,in2, in3, rom_addr, out);

input [12:0] in1;
input [7:0] in2;
input [7:0] in3;
input [3:0] rom_addr;
input clk;
output  [47:0] out;

wire [47:0] out1;
reg [10:0] mem [0:2**4 -1];
reg [12:0] in1_reg;
reg [7:0] in2_reg;
```

```
initial
begin
   $readmemb("mem.dat",mem);
end

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

wire [10:0] rom_data;
assign rom_data = mem[rom_addr];
assign out1 =  rom_data * (in1_reg + in2_reg);
assign out = out1 + in3;
endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 Mult

Example 57: 10x17 Unsigned Mult-Acc with Pre-adder

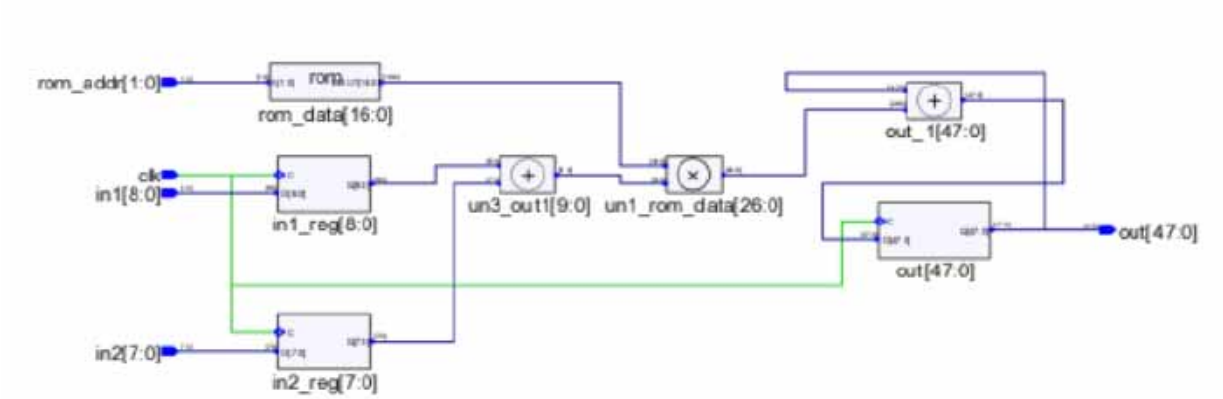
This example shows a Mult-Acc with pre-adder that the tool maps to Coeff ROM MATH block.

RTL

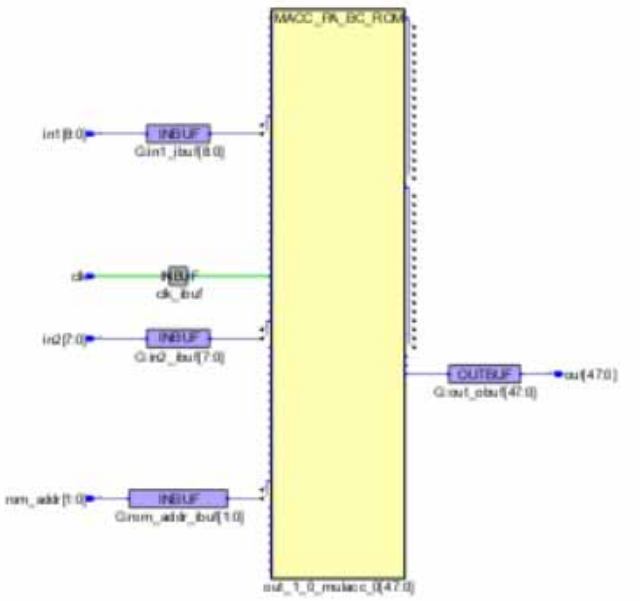
```
module test(clk, in1,,in2, rom_addr, out);

input [8:0] in1;
input [7:0] in2;
input [1:0] rom_addr;
input clk;
output reg [47:0] out;

wire [47:0] out1;
reg [16:0] mem [0:2**2 -1];
reg [8:0] in1_reg;
reg [7:0] in2_reg;
```

```
initial
begin
   $readmemb("mem.dat",mem);
end

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

wire [16:0] rom_data;
assign rom_data = mem[rom_addr];
assign out1 =  rom_data * (in1_reg + in2_reg);

always@(posedge clk)
begin
   out <= out1 + out;
end
endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 Mult

Example 58: 10x17 Unsigned Mult-Acc-Add with Pre-adder

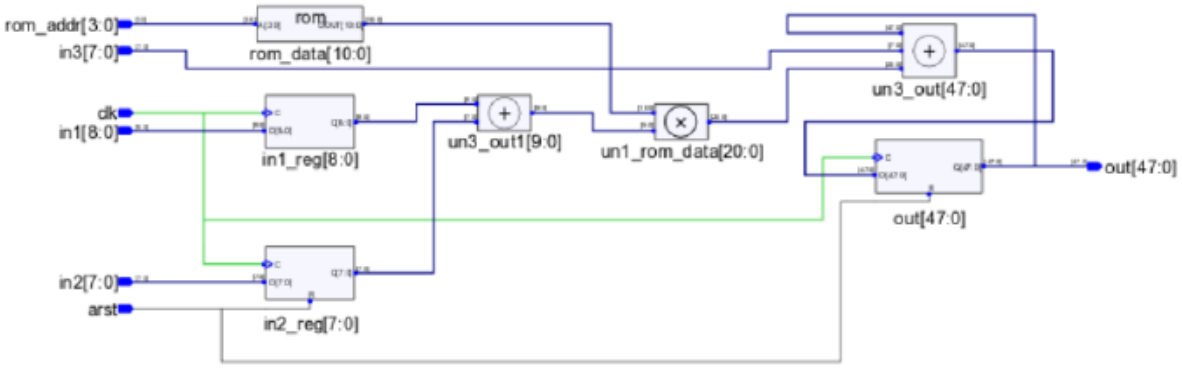
This example shows a Mult-Acc-Add with Pre-adder that the tool maps to Coeff ROM MATH block.

RTL

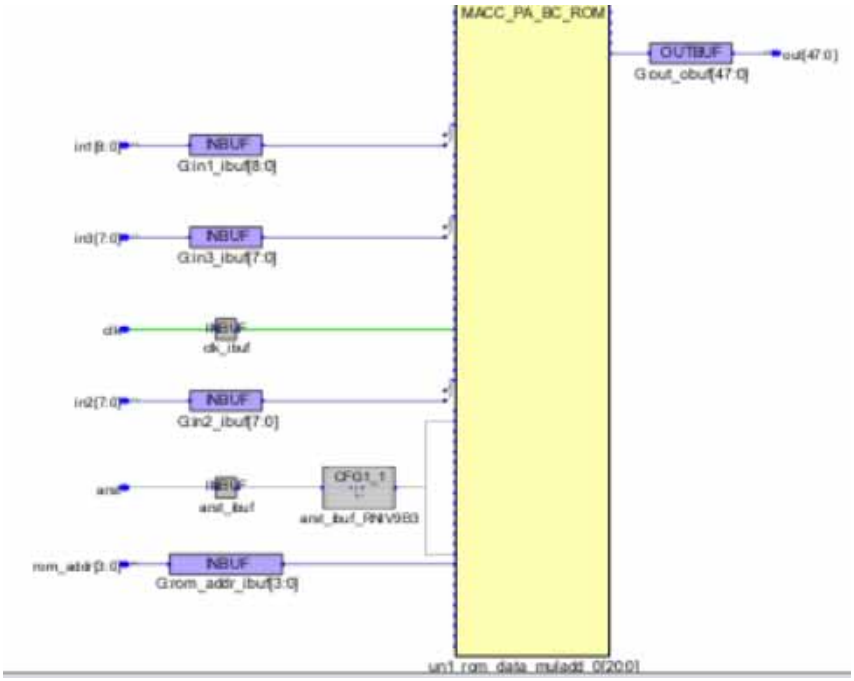
```
module test(clk, arst, in1,,in2, in3, rom_addr, out);

input [8:0] in1;
input [7:0] in2;
input [7:0] in3;
input [3:0] rom_addr;
input clk,arst;
output reg [47:0] out;

wire [47:0] out1;
reg [10:0] mem [0:2**4 -1];
reg [8:0] in1_reg;
reg [7:0] in2_reg;
```

```
initial
begin
   $readmemb("mem.dat",mem);
end

always@(posedge clk)

begin
   in1_reg <= in1;
end

always@(posedge clk or posedge arst)
begin
   if(arst)
      in2_reg <= 0;
   else
   in2_reg <= in2;
end

always@(posedge clk)
begin
   if(arst)
      out <= 0;
   else
      out <= out1 + in3 + out;
end

wire [10:0] rom_data;
assign rom_data = mem[rom_addr];
assign out1 =  rom_data * (in1_reg + in2_reg);
endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 Mult

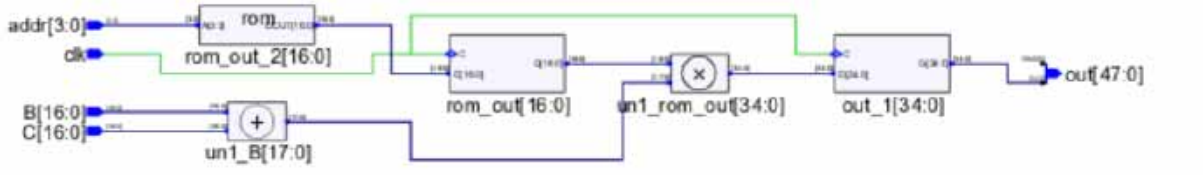
Example 59: RTL Coding Style with Case Statement for Simple Pre-adder Multiplier

RTL

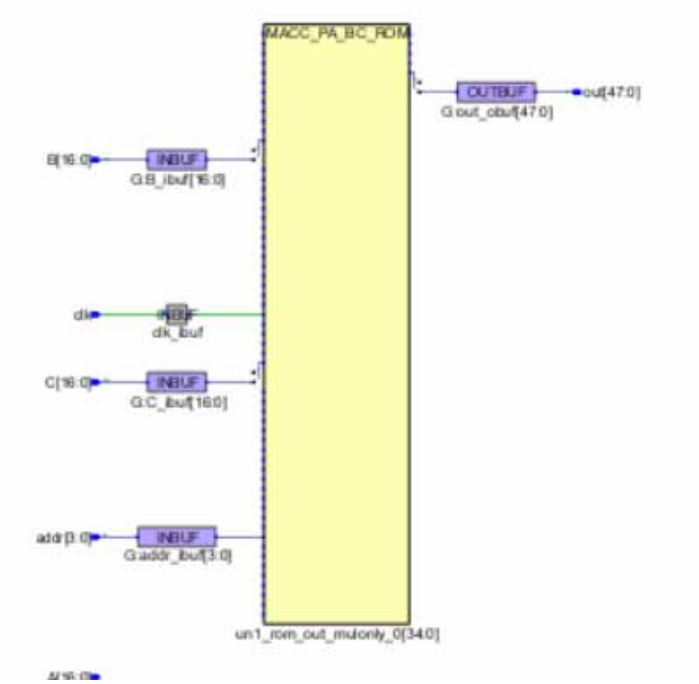
```
module preadd_mult(clk,A,B,C,out,addr );

input clk;
input unsigned [16:0] A,B,C;
input [3:0] addr;
output [33:0] out;

reg [16:0] rom_out;
reg [33:0] out;

always@(posedge clk)
   begin
      case (addr)
         4'd0 : rom_out <= 17'b01100000100011001;
         4'd1 : rom_out <= 17'b00001101100111001;
```

```
            4'd2 : rom_out <= 17'b01011011110110011;
            4'd3 : rom_out <= 17'b10100110000000001;
            4'd4 : rom_out <= 17'b01101001111111001;
            4'd5 : rom_out <= 17'b01010000100010101;
            4'd6 : rom_out <= 17'b10101101010001101;
            4'd7 : rom_out <= 17'b01001001010010101;
            4'd8 : rom_out <= 17'b01110111000111111;
            4'd9 : rom_out <= 17'b10010101111110110;
            4'd10 : rom_out <= 17'b01000110000010001;
            4'd11 : rom_out <= 17'b11100000110110011;
            4'd12 : rom_out <= 17'b10110101101111011;
            4'd13 : rom_out <= 17'b01111010011000001;
            4'd14 : rom_out <= 17'b00110110100111110;
            4'd15 : rom_out <= 17'b11110101000010001;
            default : rom_out <= 17'b00010010011010101;
endcase
   out <= rom_out * (B+ C);
end
endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 Mult
Total LUTs: 0

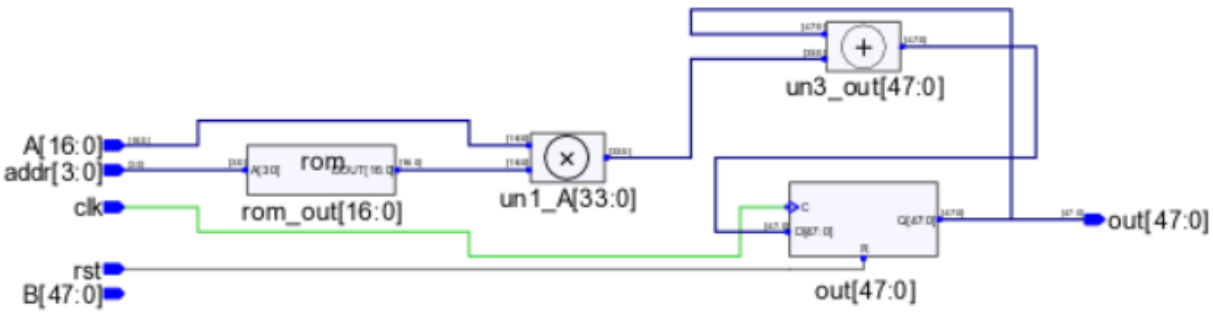
Example 60: RTL Coding Style with Case Statement for Simple Pre-Adder Mult Acc with Asynchronous Reset on Output

RTL

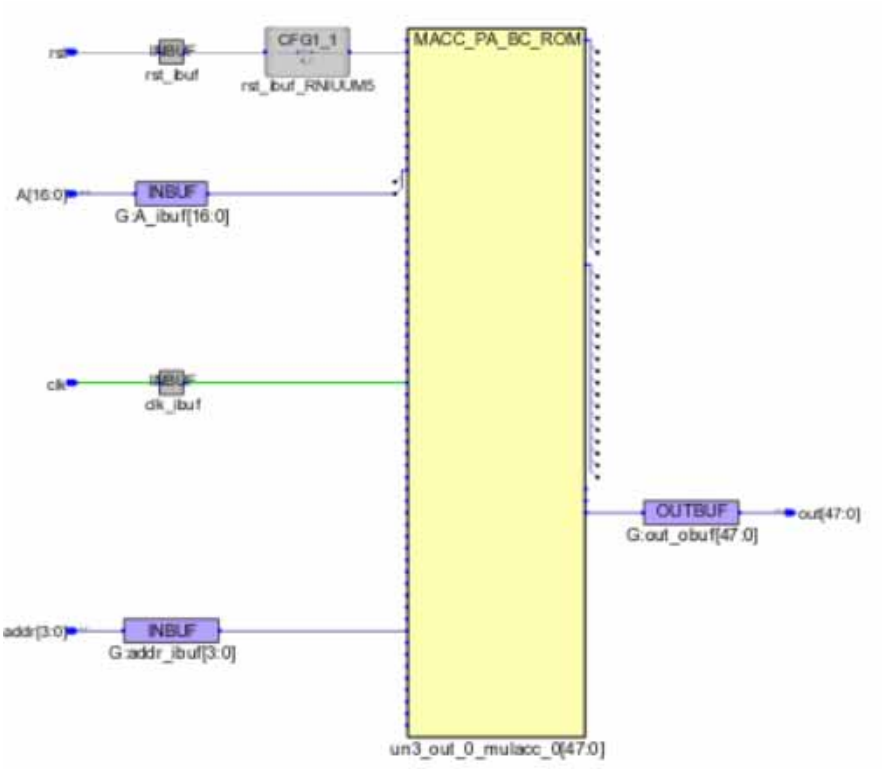
```
module preadd_mult_acc(clk,A,B,out,addr,rst );

input clk,rst;
input [16:0] A;
input [47:0] B;
input [3:0] addr;
output [47:0] out;

reg [16:0] rom_out;
reg  [47:0] out;
```

```
always@(addr)
   begin
      case (addr)
         4'd0 : rom_out <= 17'b01100000100011001;
         4'd1 : rom_out <= 17'b00001101100111001;
         4'd2 : rom_out <= 17'b01011011110110011;
         4'd3 : rom_out <= 17'b10100110000000001;
         4'd4 : rom_out <= 17'b01101001111111001;
         4'd5 : rom_out <= 17'b01010000100010101;
         4'd6 : rom_out <= 17'b10101101010001101;
         4'd7 : rom_out <= 17'b01001001010010101;
         4'd8 : rom_out <= 17'b01110111000111111;
         4'd9 : rom_out <= 17'b10010101111110110;
         4'd10 : rom_out <= 17'b01000110000010001;
         4'd11 : rom_out <= 17'b11100000110110011;
         4'd12 : rom_out <= 17'b10110101101111011;
         4'd13 : rom_out <= 17'b01111010011000001;
         4'd14 : rom_out <= 17'b00110110100111110;
         4'd15 : rom_out <= 17'b11110101000010001;
         default : rom_out <= 17'b00010010011010101;
      endcase
   end

always@(posedge clk or posedge rst)
   begin
      if(rst)
         out <= 0;
      else
         out <= out + (rom_out * A);
   end
endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

SLE 1 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 MultAcc
Total LUTs: 1

Inferring MACC_PA_BC_ROM Block in DOTP mode

The Coefficient ROM Math block, MACC_PA_BC_ROM primitive, when configured in DOTP mode has two independent signed 9x9-bit or unsigned 8x8-bit multipliers followed by addition of these two products.

The following examples show how to infer MACC_PA_BC_ROM blocks in DOTP mode:

- Example 61: Signed Mult-Add DOTP, on page 146
- Example 62: Unsigned Mult-Add DOTP, on page 148
- Example 63: Signed Mult-Add DOTP Followed by Add, on page 149
- Example 64: Unsigned Mult-Add DOTP Followed by Add, on page 151
- Example 65: Unsigned Mult-Add DOTP Followed by Acc, on page 153
- Example 66: Signed Mult-Add DOTP Followed by Acc, on page 155

Example 61: Signed Mult-Add DOTP

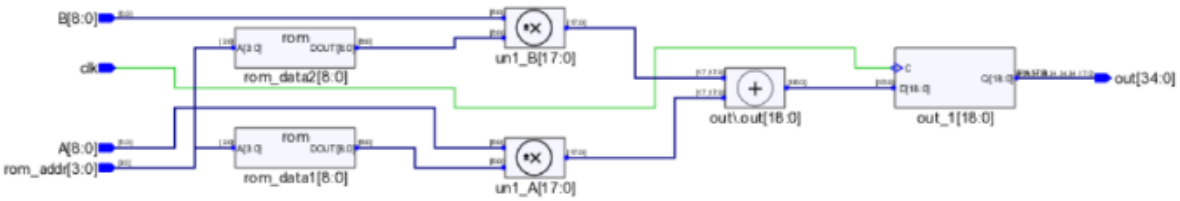
The RTL is for DOTP computation of ab + cd. The synthesis tool infers a single Coefficient ROM Math block in DOTP mode with Signed Mult-Add configuration. The ROM blocks are separate for each input.

RTL

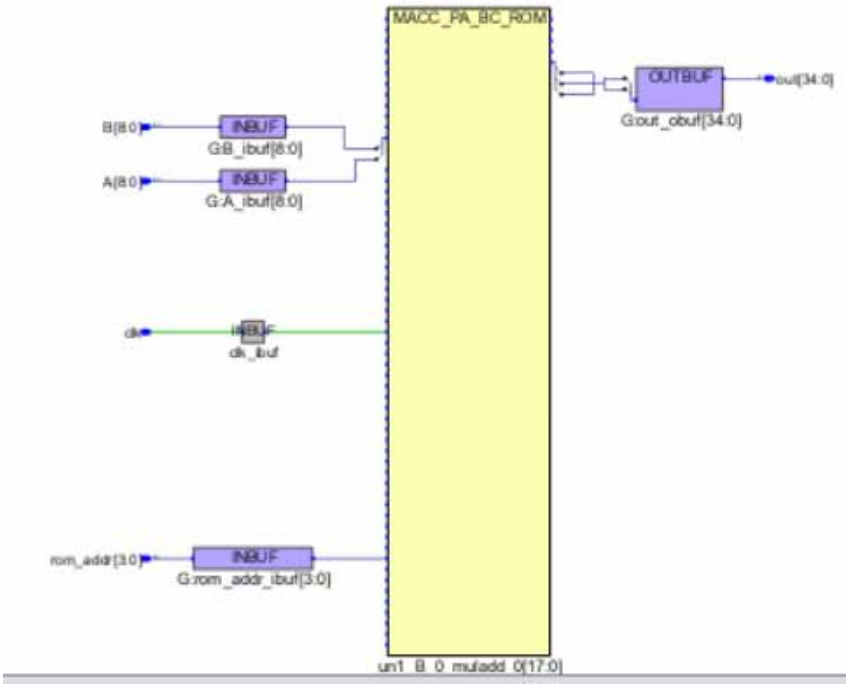
```
module test(clk,A,B,out,rom_addr);

input clk;
input signed [8:0] A;
input signed [8:0] B;
input [3:0] rom_addr;
output [17:0] out;

reg signed [17:0] out;
reg [8:0] mem1 [0:2**4 -1];
reg [8:0] mem2 [0:2**4 -1];

initial
   begin
      $readmemb("mem.dat",mem1);
   end

initial
   begin
      $readmemb("mem.dat",mem2);
   end

wire signed [8:0] rom_data1;
assign rom_data1 = mem1[rom_addr];

wire signed [8:0] rom_data2;
assign rom_data2 = mem2[rom_addr];
```

```
always@(posedge clk)
   begin
      out <= (rom_data1 * A) + (rom_data2 * B) ;
   end
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 MultAdd

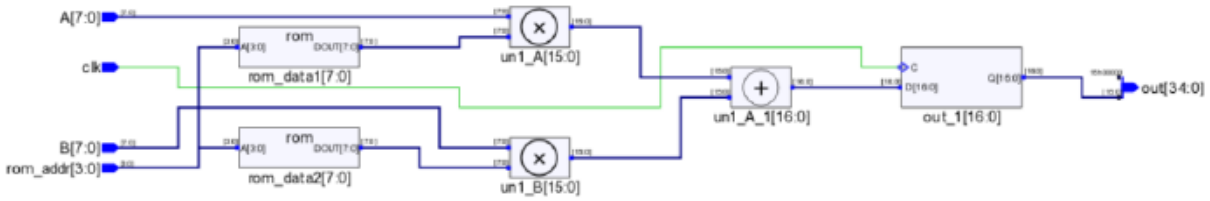
Example 62: Unsigned Mult-Add DOTP

The RTL is for DOTP computation of ab + cd. The synthesis tool infers a single Coefficient ROM Math block in DOTP mode with unsigned Mult-Add configuration. The ROM blocks are separate for each input.

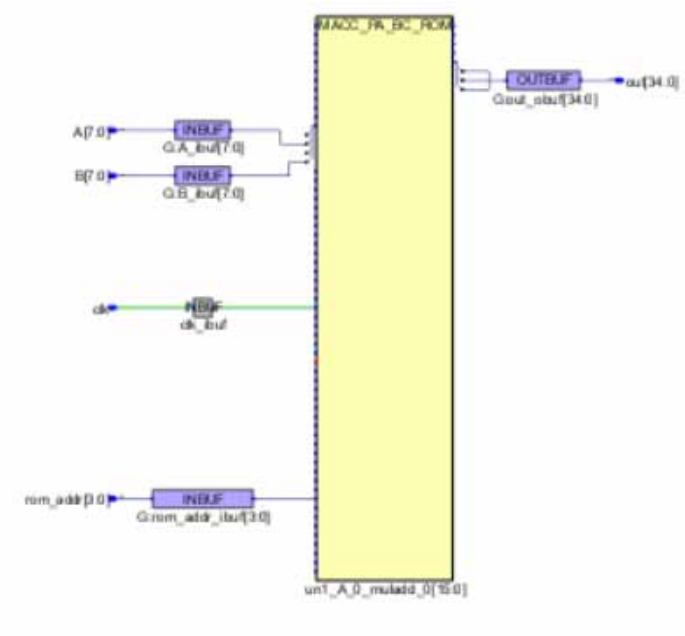
RTL

```
module test(clk,A,B,out,rom_addr);

input clk;
input  [7:0] A;
input  [7:0] B;
input [3:0] rom_addr;
output [16:0] out;

reg  [16:0] out;
reg [7:0] mem1 [0:2**4 -1];
reg [7:0] mem2 [0:2**4 -1];

initial
   begin
      $readmemb("mem.dat",mem1);
   end

initial
   begin
      $readmemb("mem.dat",mem2);
   end

wire  [7:0] rom_data1;
assign rom_data1 = mem1[rom_addr];

wire  [7:0] rom_data2;
assign rom_data2 = mem2[rom_addr];

always@(posedge clk)
   begin
      out <= (rom_data1 * A) + (rom_data2 * B) ;
   end
endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 MultAdd

Example 63: Signed Mult-Add DOTP Followed by Add

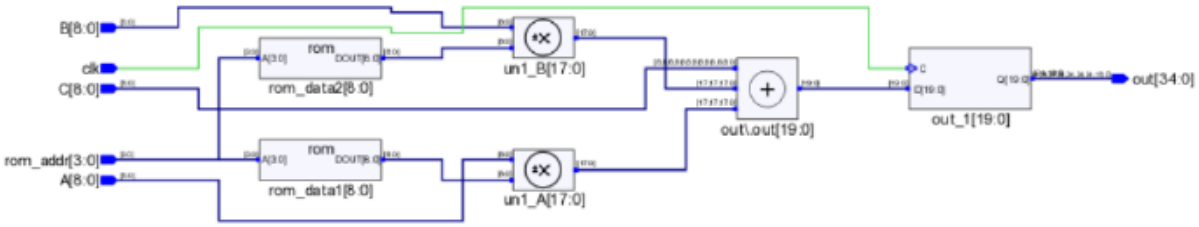
The RTL is for DOTP computation of ab + cd + e. The synthesis tool infers a single Coefficient ROM Math block in DOTP mode with Signed Mult-Add configuration. The ROM blocks are separate for each input.

RTL

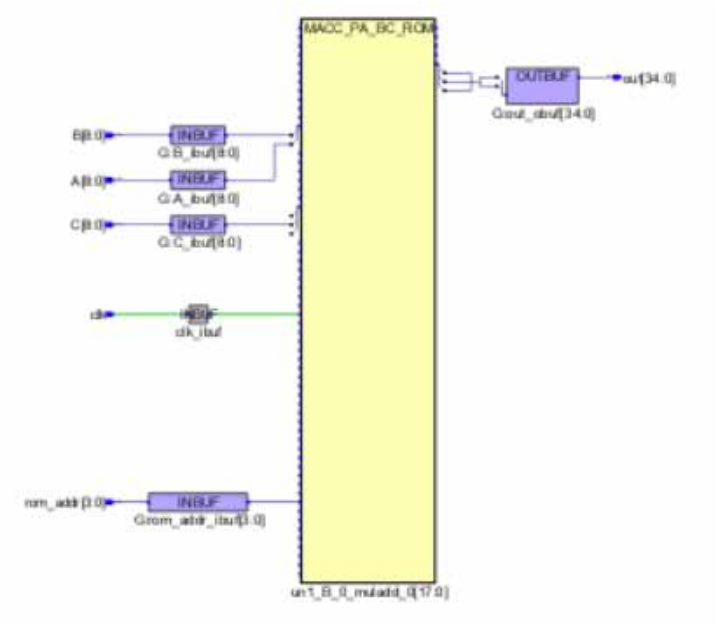
```
module test(clk,A,B,C,out,rom_addr);

input clk;
input signed [8:0] A;
input signed [8:0] B;
input signed [8:0] C;
input [3:0] rom_addr;
output [18:0] out;

reg signed [18:0] out;
reg [8:0] mem1 [0:2**4 -1];
reg [8:0] mem2 [0:2**4 -1];
```

```
initial
   begin
      $readmemb("mem.dat",mem1);
   end

initial
   begin
      $readmemb("mem.dat",mem2);
   end

wire signed [8:0] rom_data1;
assign rom_data1 = mem1[rom_addr];

wire signed [8:0] rom_data2;
assign rom_data2 = mem2[rom_addr];

always@(posedge clk)
   begin
      out <= (rom_data1 * A) + (rom_data2 * B) + C;
   end
endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 MultAdd

Example 64: Unsigned Mult-Add DOTP Followed by Add

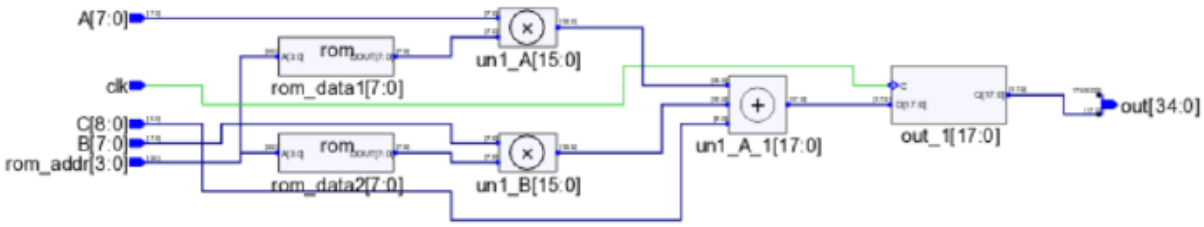
The RTL is for DOTP computation of ab + cd + e. The synthesis tool infers a single Coefficient ROM Math block in DOTP mode with Signed Mult-Add configuration. The ROM blocks are separate for each input.

RTL

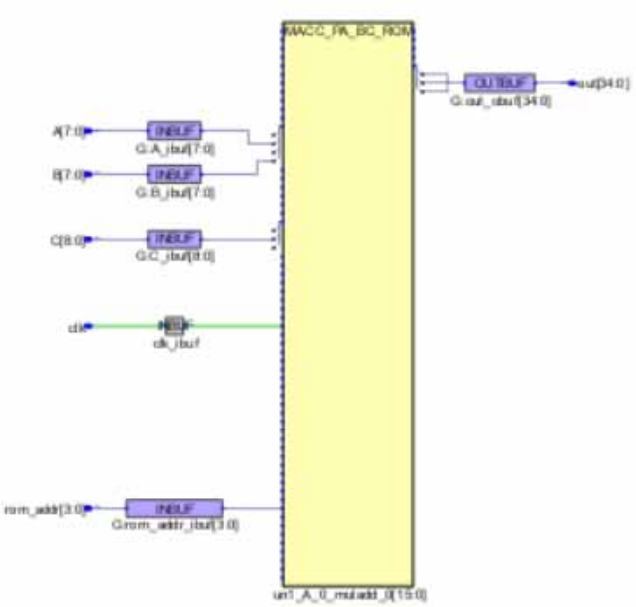
```
module test(clk,A,B,C,out,rom_addr);

input clk;
input  [7:0] A;
input  [7:0] B;
input  [8:0] C;
input [3:0] rom_addr;
output [17:0] out;

reg  [17:0] out;
reg [7:0] mem1 [0:2**4 -1];
reg [7:0] mem2 [0:2**4 -1];
```

```
initial
   begin
      $readmemb("mem.dat",mem1);
   end

initial
   begin
      $readmemb("mem.dat",mem2);
   end

wire  [7:0] rom_data1;
assign rom_data1 = mem1[rom_addr];

wire  [7:0] rom_data2;
assign rom_data2 = mem2[rom_addr];

always@(posedge clk)
   begin
      out <= (rom_data1 * A) + (rom_data2 * B) + C;
   end
endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 MultAdd

Example 65: Unsigned Mult-Add DOTP Followed by Acc

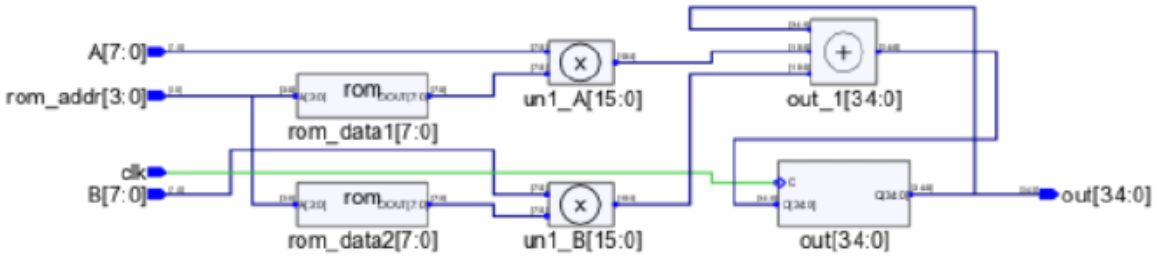
The RTL is for DOTP computation of Mult-Acc scenario. The synthesis tool infers a single Coefficient ROM Math block in DOTP mode with Unsigned Mult-Acc configuration. The ROM blocks are separate for each input.

RTL

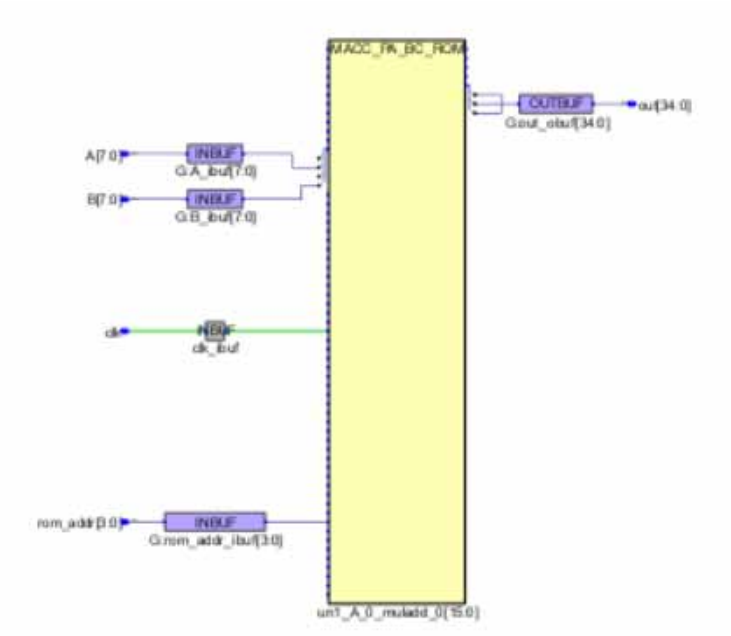
```
module test(clk,A,B,out,rom_addr);

input clk;
input  [7:0] A;
input  [7:0] B;
input [3:0] rom_addr;
output [34:0] out;
reg  [34:0] out;

wire  [34:0] prod;
reg [7:0] mem1 [0:2**4 -1];
reg [7:0] mem2 [0:2**4 -1];
```

```
initial
   begin
      $readmemb("mem.dat",mem1);
   end

initial
   begin
      $readmemb("mem.dat",mem2);
   end

wire  [7:0] rom_data1;
assign rom_data1 = mem1[rom_addr];

wire  [7:0] rom_data2;
assign rom_data2 = mem2[rom_addr];

always@(posedge clk)
   begin
      out <= out + prod;
   end
assign prod = (rom_data1 * A) + (rom_data2 * B);

endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 MultAcc

Example 66: Signed Mult-Add DOTP Followed by Acc

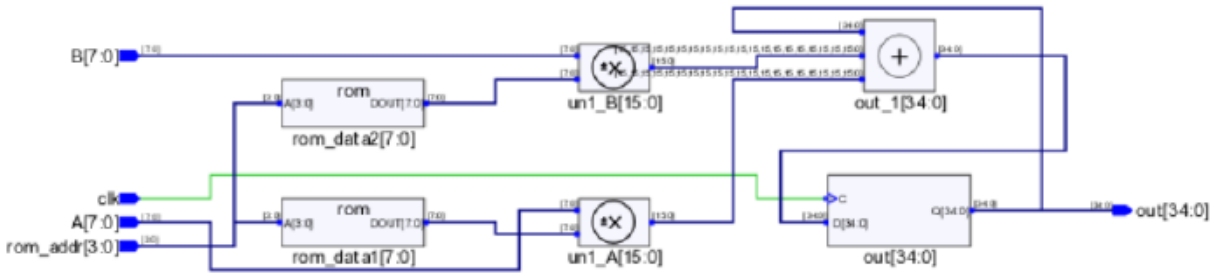
The RTL is for DOTP computation of Mult-Acc scenario. The synthesis tool infers a single Coefficient ROM Math block in DOTP mode with Signed Mult-Acc configuration. The ROM blocks are separate for each input.

RTL

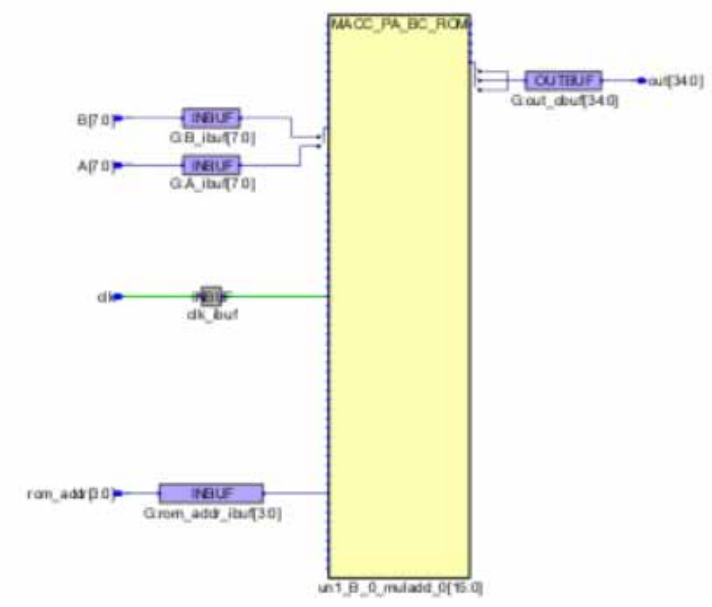
```
module test(clk,A,B,out,rom_addr);

input clk;
input signed [7:0] A;
input signed [7:0] B;
input [3:0] rom_addr;
output [34:0] out;

reg signed [34:0] out;
wire signed [34:0] prod;
reg [7:0] mem1 [0:2**4 -1];
reg [7:0] mem2 [0:2**4 -1];
```

```
initial
   begin
      $readmemb("mem.dat",mem1);
      end

initial
   begin
      $readmemb("mem.dat",mem2);
   end

wire signed [7:0] rom_data1;
assign rom_data1 = mem1[rom_addr];

wire signed [7:0] rom_data2;
assign rom_data2 = mem2[rom_addr];

always@(posedge clk)
   begin
      out <= out + prod;
   end
assign prod = (rom_data1 * A) + (rom_data2 * B);
endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA_BC_ROM: 1 MultAcc

Inferring MACC_PA_BC_ROM Block for Cascaded Chain Using BCOUT->B Connection

The MACC_PA_BC_ROM block supports the packing of cascaded chain delay tap registers at the input of multipliers as in a pre-cascaded Direct form FIR filter using BCOUT->B connection.

The design must conform to the following prerequisites:

- Cascaded chain packing will not happen if the multiplier output is registered due to DSP architecture limitation.
- Direct form FIR filter with pre-cascaded adder structure.
- Cascaded chain packing will not happen in case wide multipliers; that is, input data width > 18 for signed and >17 for unsigned.

The following examples show how to infer MACC_PA_BC_ROM blocks in cascade chain mode:

- Example 67: 4-tap Transpose FIR Filter, on page 158
- Example 68: 5-tap Direct Form FIR Filter, on page 162
- Example 69: 6-tap Systolic FIR Filter, on page 166
- Example 70: 4-tap Symmetric FIR filter, on page 171
- Example 71: 128-tap Symmetric FIR filter, on page 176

Example 67: 4-tap Transpose FIR Filter

The following code is an example of a 4-tap Transpose FIR filter that gets completely packed into the MACC blocks.

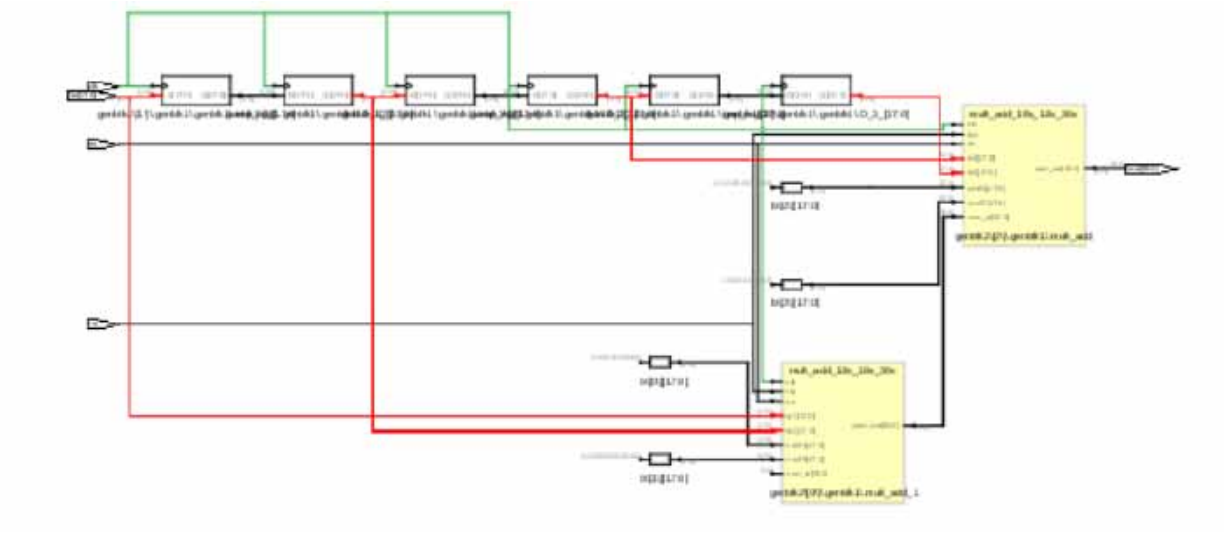
RTL

```
module mult_add(clk, rst, en, in1, in2, coef1, coef2, sum_in, sum_out);

parameter din_width = 18;
parameter coef_width = 18;
parameter dout_width = 2*din_width;

input clk, rst, en;
input signed [din_width - 1 : 0] in1;
input signed [din_width - 1 : 0] in2;
input signed [coef_width - 1 : 0] coef1;
input signed [coef_width - 1 : 0] coef2;
input signed[dout_width - 1 : 0] sum_in;
output signed[dout_width - 1 : 0] sum_out;

reg signed [dout_width - 1 : 0] sum_out;
wire signed [din_width + coef_width - 1 : 0] mult1;
wire signed [din_width + coef_width - 1 : 0] mult2;

assign mult1 = in1 * coef1;
assign mult2 = in2 * coef2;

always @ (posedge clk)
begin
      sum_out <= sum_in + mult1 + mult2;
end
endmodule

module fir(clk, rst, en, Din, Dout);

parameter din_width=18;
parameter coef_width = 18;
parameter dout_width=2*din_width;
parameter TapLen = 8;
```

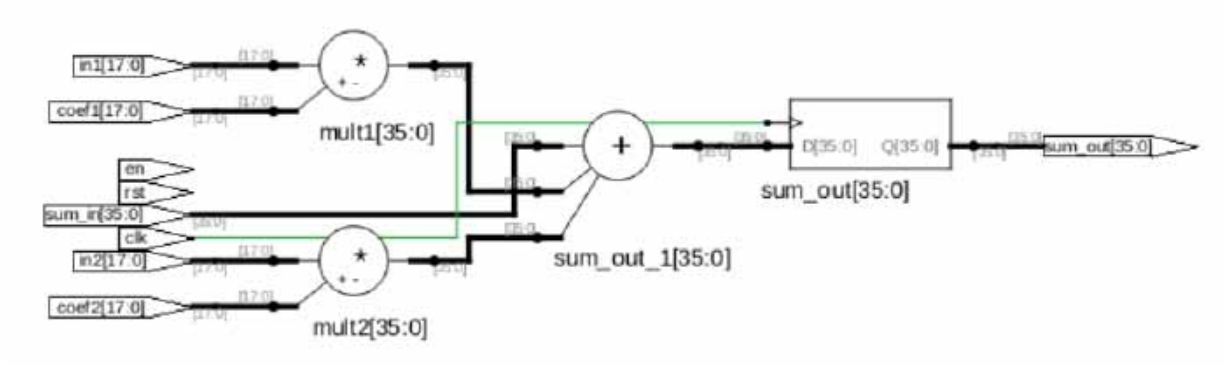
```
input clk, rst, en;
input signed [din_width - 1 : 0] Din;
output signed [dout_width - 1:0] Dout;
reg signed [din_width-1:0] D [TapLen-1:0];
wire signed [dout_width-1:0] sum_tmp [TapLen :0 ];
wire signed [coef_width-1:0] b [100:0] /* synthesis syn_keep=1 */;

assign sum_tmp[0] = 0;
generate
begin: CoeffGen
   assign b[0] = 18'b111111001111100000;
   assign b[1] = 18'b111001000001101000;
   assign b[2] = 18'b111001001000101000;
   assign b[3] = 18'b111111001111000100;
   assign b[4] = 18'b111111001111100000;
   assign b[5] = 18'b111101000001101001;
   assign b[6] = 18'b111101001000101010;
   assign b[7] = 18'b010101111111111010;
   assign b[8] = 18'b010101111111111010;
   assign b[9] = 18'b010101111111111010;
   assign b[10] = 18'b010101111111111010;
   assign b[11] = 18'b010111001111100000;
   assign b[12] = 18'b010001000001101000;
   assign b[13] = 18'b010001001000101000;
   assign b[14] = 18'b010111001111000100;
   assign b[15] = 18'b010111001111100000;
   assign b[16] = 18'b010111011111100000;
   assign b[17] = 18'b111111011111100000;
   assign b[18] = 18'b111001011001101000;
   assign b[19] = 18'b111001011000101000;
   assign b[20] = 18'b111111111111000100;
   assign b[21] = 18'b111111101111100000;
   assign b[22] = 18'b111101110001101001;
   assign b[23] = 18'b111101111000101010;
   assign b[24] = 18'b010101001111111010;
   assign b[25] = 18'b010101001111111010;
   assign b[26] = 18'b010101001111111010;
   assign b[27] = 18'b010101011111111010;
   assign b[28] = 18'b010111101111100000;
   assign b[29] = 18'b010001100001101000;
   assign b[30] = 18'b010001111000101000;

genvar j;
for(j = 31; j<= 100; j=j+1)
   begin
      assign b[j] = 18'b010111010111000100;
   end
end
endgenerate

//Generation of tap delay with additional delay element after every two taps

generate
genvar i;
for(i = 0; i<= TapLen-1; i=i+1)
   begin
      reg [din_width-1:0] tmp_reg;
```

```
   begin
      if(i==0)
      begin
         assignD[0] = Din;
      end
      else
      begin
         always @ (posedge clk)
         begin
            D[i] <= tmp_reg;
            tmp_reg <= D[i-1];

         end
      end
   end
end
endgenerate

generate
genvar n;
   for(n=0; n<=TapLen-1; n=n+2)
   begin
      if(n==TapLen-1) mult_add #(din_width, coef_width, dout_width)
mult_add_2(clk, rst, en, D[n], {din_width{1'b0}}, b[n], b[n+1], sum_tmp[n/2],
sum_tmp[n/2+1]);
      else if (n==0) mult_add #(din_width, coef_width, dout_width) mult_add_1(clk,
rst, en, D[n], D[n+1], b[n], b[n+1], 0, sum_tmp[n/2+1]);
      else  mult_add #(din_width, coef_width, dout_width) mult_add(clk, rst, en,
D[n], D[n+1], b[n], b[n+1], sum_tmp[n/2], sum_tmp[n/2+1]);
   end
endgenerate
assign Dout = (TapLen%2) ? sum_tmp[TapLen/2+1] : sum_tmp[TapLen/2];
endmodule

module fir_top(clk, rst, en, Din1, Dout1);

input clk, rst, en;
input signed [17 : 0] Din1;
output signed [35:0] Dout1;
reg signed [17 : 0] Din1_reg;

always @ (posedge clk)
begin
      Din1_reg <= Din1;
end
fir #(18, 18, 36,  4) fir1(clk, rst, en, Din1_reg, Dout1);
endmodule
```

SRS (RTL) View

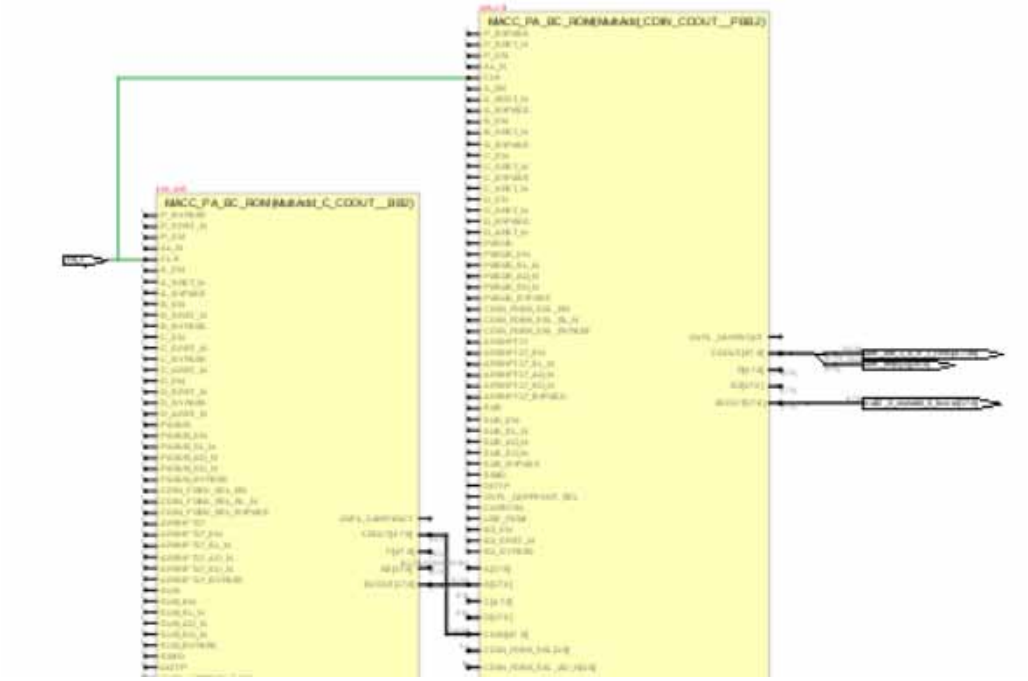


Each Mult-Add block in the above figure (yellow blocks) is implemented as below.



Snippet of the MACC_PA_BC_ROM connections with CDOUT-> CDIN and BCOUT->B connections.

SRM (Technology) View



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 4 of 924 (0%)
MACC_PA: 1 MultAdd
MACC_PA_BC_ROM: 3 MultAdds

Example 68: 5-tap Direct Form FIR Filter

In this example, a 5-tap Direct Form FIR filter gets completely packed into the MACC blocks.

RTL

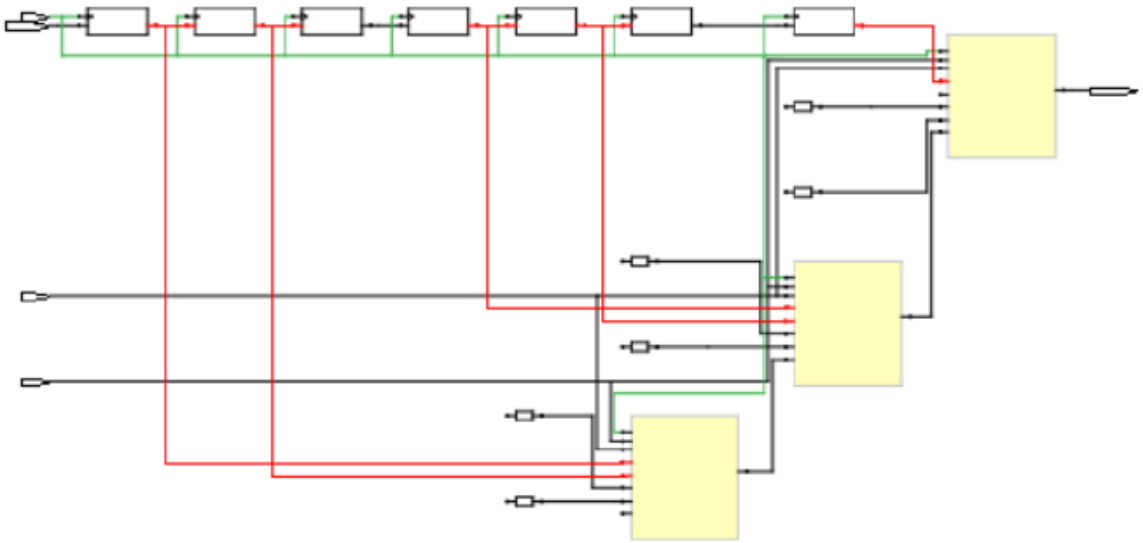
```
module mult_add(clk, rst, en, in1, in2, coef1, coef2, sum_in, sum_out);

parameter din_width = 18;
parameter coef_width = 18;
parameter dout_width = din_width+coef_width;

input clk, rst, en;
input  [din_width - 1 : 0] in1;
input  [din_width - 1 : 0] in2;
input  [coef_width - 1 : 0] coef1;
input  [coef_width - 1 : 0] coef2;
input [dout_width - 1 : 0] sum_in;
output [dout_width - 1 : 0] sum_out;
```

```
reg  [dout_width - 1 : 0] sum_out;
wire  [din_width + coef_width - 1 : 0] mult1;
wire  [din_width + coef_width - 1 : 0] mult2;

assign mult1 = in1 * coef1;
assign mult2 = in2 * coef2;

always @ (posedge clk)
   begin
      sum_out <= sum_in + mult1 + mult2;
   end

endmodule

module fir(clk, rst, en, Din, Dout);

parameter din_width=17;
parameter coef_width = 17;
parameter dout_width=2*din_width;
parameter TapLen = 8;

input clk, rst, en;
input  [din_width - 1 : 0] Din;
output  [dout_width - 1:0] Dout;

reg  [din_width-1:0] D [TapLen-1:0];
wire  [dout_width-1:0] sum_tmp [TapLen :0 ];
wire  [coef_width-1:0] b [31:0] /* synthesis syn_keep=1 */;
assign sum_tmp[0] = 0;

generate
   begin: CoeffGen
      assign b[0] = 17'b11111001111100000;
      assign b[1] = 17'b11001000001101000;
      assign b[2] = 17'b11001001000101000;
      assign b[3] = 17'b11111001111000100;
      assign b[4] = 17'b11111001111100000;
      assign b[5] = 17'b11101000001101001;
      assign b[6] = 17'b11101001000101010;
      assign b[7] = 17'b10101111111111010;
      assign b[8] = 17'b10101111111111010;
      assign b[9] = 17'b10101111111111010;
      assign b[10] = 17'b10101111111111010;
      assign b[11] = 17'b10111001111100000;
      assign b[12] = 17'b10001000001101000;
      assign b[13] = 17'b10001001000101000;
      assign b[14] = 17'b10111001111000100;
      assign b[15] = 17'b10111001111100000;
      assign b[16] = 17'b10111011111100000;
      assign b[17] = 17'b11111011111100000;
      assign b[18] = 17'b11001011001101000;
      assign b[19] = 17'b11001011000101000;
      assign b[20] = 17'b11111111111000100;
      assign b[21] = 17'b11111101111100000;
      assign b[22] = 17'b11101110001101001;
      assign b[23] = 17'b11101111000101010;
      assign b[24] = 17'b10101001111111010;
      assign b[25] = 17'b10101001111111010;
      assign b[26] = 17'b10101001111111010;
```

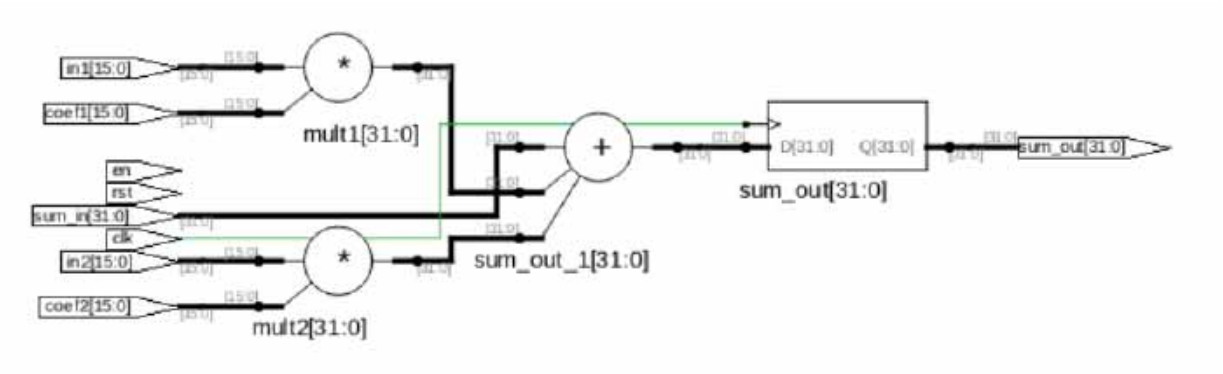
```
      assign b[27] = 17'b10101011111111010;
      assign b[28] = 17'b10111101111100000;
      assign b[29] = 17'b10001100001101000;
      assign b[30] = 17'b10001111000101000;
      assign b[31] = 17'b10111010111000100;
   end
endgenerate

//Generation of tap delay with aditional delay element after every two taps

generate
genvar i;
for(i = 0; i<= TapLen-1; i=i+1)
begin
   reg [din_width-1:0] tmp_reg;
   begin
      if(i==0)
      begin
         always @ (posedge clk)
         begin
               D[0] <= Din;
         end
      end
      else
      begin
         always @ (posedge clk)
         begin
               if(i%2==0)
               begin
               tmp_reg <= D[i-1];
               D[i] <= tmp_reg;
               end
               else
               D[i] <= D[i-1];
         end
      end
   end
end
endgenerate

generate
genvar n;
   for(n=0; n<=TapLen-1; n=n+2)
   begin
      if(n==TapLen-1) mult_add #(din_width, coef_width, dout_width)
mult_add_1(clk, rst, en, D[n], {din_width{1'b0}}, b[n], b[n+1], sum_tmp[n/2],
sum_tmp[n/2+1]);
      else  mult_add #(din_width, coef_width, dout_width) mult_add(clk, rst, en,
D[n], D[n+1], b[n], b[n+1], sum_tmp[n/2], sum_tmp[n/2+1]);
   end
endgenerate
assign Dout = (TapLen%2) ? sum_tmp[TapLen/2+1] : sum_tmp[TapLen/2];
endmodule

module fir_top(clk, rst, en, Din1, Dout1);
```

```
input clk, rst, en;
input  [15 : 0] Din1;
output  [31:0] Dout1;
fir #(16, 16, 32, 5) fir1(clk, rst, en, Din1, Dout1);
endmodule
```

SRS (RTL) View

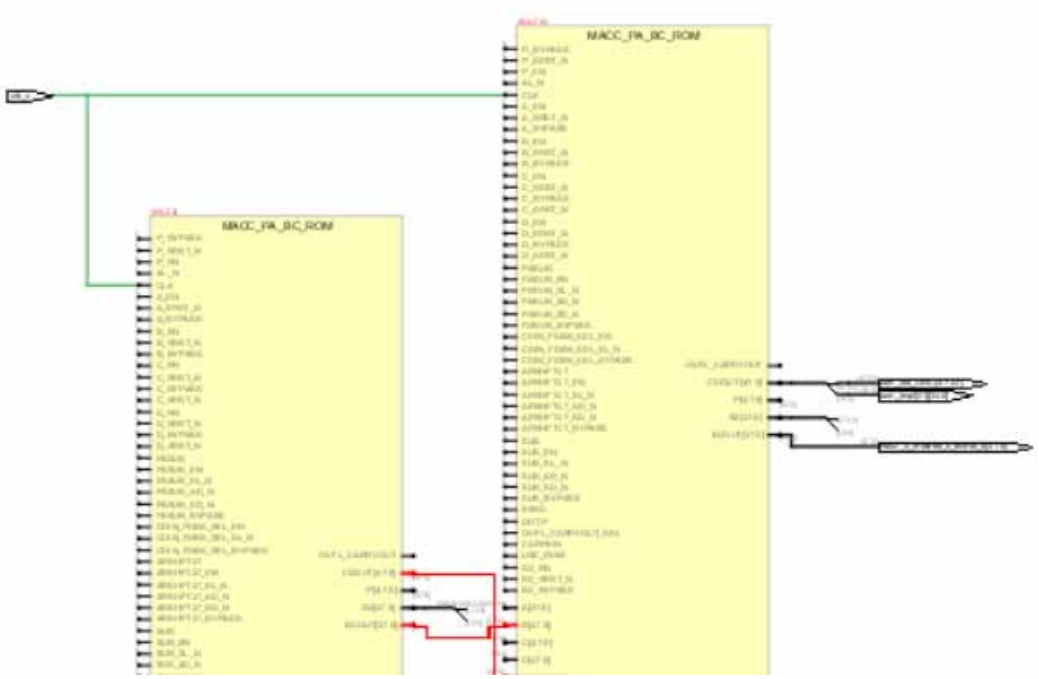


Each Mult-Add block (Yellow blocks) in the previous figure is implemented as shown in the following figure:



SRM (Technology) View

The snippet of the MACC_PA_BC_ROM connections with CDOUT-> CDIN and BCOUT->B connections.



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 5 of 924 (0%)
MACC_PA: 1 MultAdd
MACC_PA_BC_ROM: 4 MultAdds

Example 69: 6-tap Systolic FIR Filter

The following code is an example of a 6-tap Systolic FIR filter that gets completely packed into the MACC blocks.

RTL

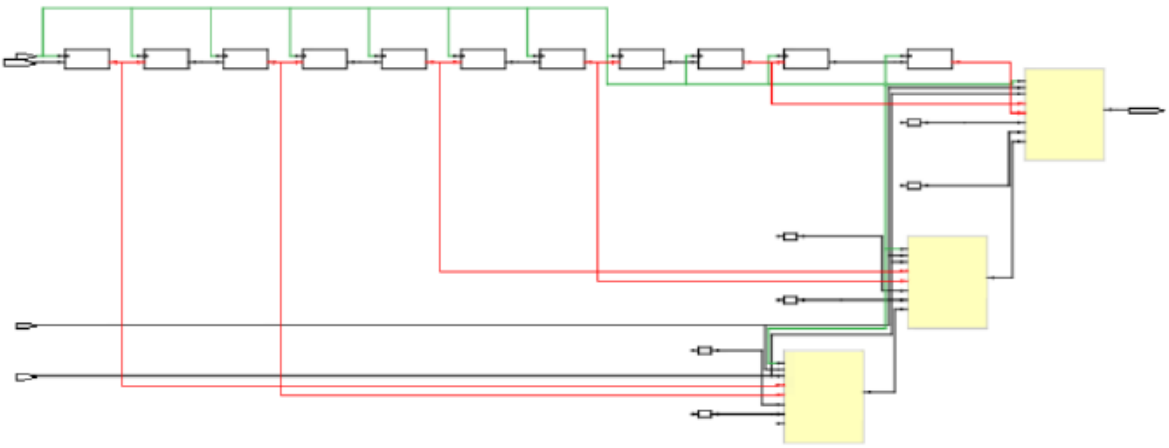
```
module mult_add(clk, rst, en, in1, in2, coef1, coef2, sum_in, sum_out);

parameter din_width = 18;
parameter coef_width = 18;
parameter dout_width = 2*din_width;

input clk, rst, en;
input signed [din_width - 1 : 0] in1;
input signed [din_width - 1 : 0] in2;
input signed [coef_width - 1 : 0] coef1;
input signed [coef_width - 1 : 0] coef2;
input signed[dout_width - 1 : 0] sum_in;
```

```
output signed[dout_width - 1 : 0] sum_out;
reg signed [dout_width - 1 : 0] sum_out;
reg signed [dout_width - 1 : 0] sum_out_reg;
wire signed [din_width + coef_width - 1 : 0] mult1;
wire signed [din_width + coef_width - 1 : 0] mult2;

assign mult1 = in1 * coef1;
assign mult2 = in2 * coef2;

always @ (posedge clk)
begin
      sum_out_reg <= sum_in + mult1;
      sum_out <= sum_out_reg + mult2;
end
endmodule

module mult_add1(clk, rst, en, in1, in2, coef1, coef2, sum_in, sum_out);

parameter din_width = 18;
parameter coef_width = 18;
parameter dout_width = 2*din_width;

input clk, rst, en;
input signed [din_width - 1 : 0] in1;
input signed [din_width - 1 : 0] in2;
input signed [coef_width - 1 : 0] coef1;
input signed [coef_width - 1 : 0] coef2;
input signed[dout_width - 1 : 0] sum_in;
output signed[dout_width - 1 : 0] sum_out;

reg signed [dout_width - 1 : 0] sum_out;
reg signed [din_width + coef_width - 1 : 0] mult1_reg;
wire signed [din_width + coef_width - 1 : 0] mult1;
wire signed [din_width + coef_width - 1 : 0] mult2;

assign mult1 = in1 * coef1;
assign mult2 = in2 * coef2;

always @ (posedge clk)
begin
      mult1_reg <= mult1;
      sum_out <= mult1_reg + mult2;
end
endmodule

module fir(clk, rst, en, Din, Dout);

parameter din_width=18;
parameter coef_width = 18;
parameter dout_width=2*din_width;
parameter TapLen = 8;

input clk, rst, en;
input signed [din_width - 1 : 0] Din;
output signed [dout_width - 1:0] Dout;

reg signed [din_width-1:0] D [TapLen-1:0];
wire signed [dout_width-1:0] sum_tmp [TapLen :0 ];
wire signed [coef_width-1:0] b [100:0] /* synthesis syn_keep=1 */;
```

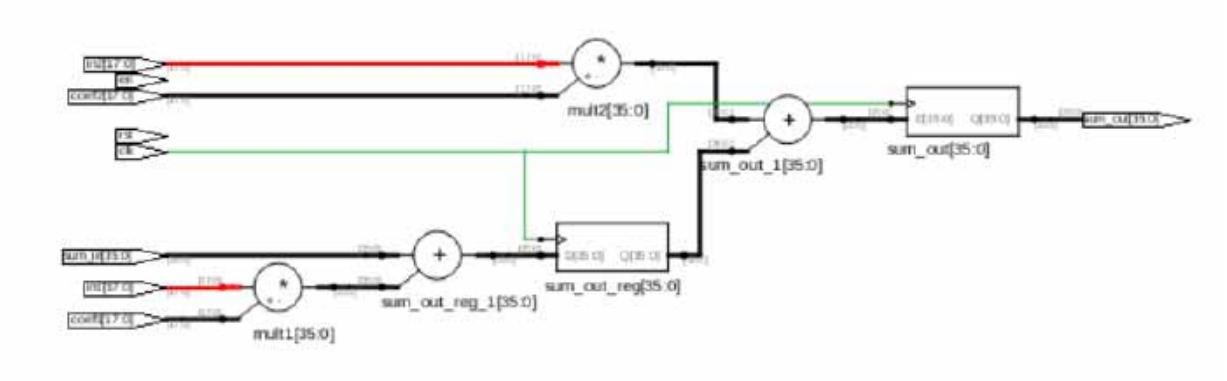
```
assign sum_tmp[0] = 0;
generate
begin: CoeffGen
   assign b[0] = 18'b111111001111100000;
   assign b[1] = 18'b111001000001101000;
   assign b[2] = 18'b111001001000101000;
   assign b[3] = 18'b111111001111000100;
   assign b[4] = 18'b111111001111100000;
   assign b[5] = 18'b111101000001101001;
   assign b[6] = 18'b111101001000101010;
   assign b[7] = 18'b010101111111111010;
   assign b[8] = 18'b010101111111111010;
   assign b[9] = 18'b010101111111111010;
   assign b[10] = 18'b010101111111111010;
   assign b[11] = 18'b010111001111100000;
   assign b[12] = 18'b010001000001101000;
   assign b[13] = 18'b010001001000101000;
   assign b[14] = 18'b010111001111000100;
   assign b[15] = 18'b010111001111100000;
   assign b[16] = 18'b010111011111100000;
   assign b[17] = 18'b111111011111100000;
   assign b[18] = 18'b111001011001101000;
   assign b[19] = 18'b111001011000101000;
   assign b[20] = 18'b111111111111000100;
   assign b[21] = 18'b111111101111100000;
   assign b[22] = 18'b111101110001101001;
   assign b[23] = 18'b111101111000101010;
   assign b[24] = 18'b010101001111111010;
   assign b[25] = 18'b010101001111111010;
   assign b[26] = 18'b010101001111111010;
   assign b[27] = 18'b010101011111111010;
   assign b[28] = 18'b010111101111100000;
   assign b[29] = 18'b010001100001101000;
   assign b[30] = 18'b010001111000101000;

genvar j;
   for(j = 31; j<= 100; j=j+1)
      begin
         assign b[j] = 18'b010111010111000100;
      end
   end
endgenerate

//Generation of tap delay with aditional delay element after every two taps

generate
genvar i;
for(i = 0; i<= TapLen-1; i=i+1)
begin
   reg [din_width-1:0] tmp_reg;
   begin
      if(i==0)
      begin
         always @ (posedge clk)
         begin
               D[0] <= Din;
```

```
         end
      end
      else
      begin
         always @ (posedge clk)
         begin
               D[i] <= tmp_reg;
               tmp_reg <= D[i-1];

         end
      end
   end
end
endgenerate

generate
genvar n;
   for(n=0; n<=TapLen-1; n=n+2)
   begin
      if(n==TapLen-1) mult_add #(din_width, coef_width, dout_width)
mult_add_2(clk, rst, en, D[n], {din_width{1'b0}}, b[n], b[n+1], sum_tmp[n/2],
sum_tmp[n/2+1]);
      else if (n==0) mult_add1 #(din_width, coef_width, dout_width) mult_add_1(clk,
rst, en, D[n], D[n+1], b[n], b[n+1], 0, sum_tmp[n/2+1]);
      else  mult_add #(din_width, coef_width, dout_width) mult_add(clk, rst, en,
D[n], D[n+1], b[n], b[n+1], sum_tmp[n/2], sum_tmp[n/2+1]);
   end
endgenerate
assign Dout = (TapLen%2) ? sum_tmp[TapLen/2+1] : sum_tmp[TapLen/2];
endmodule

module fir_top(clk, rst, en, Din1, Dout1);
input clk, rst, en;
input signed [17 : 0] Din1;
output signed [35:0] Dout1;
fir #(18, 18, 36,  6) fir1(clk, rst, en, Din1, Dout1);
endmodule
```

SRS (RTL) View

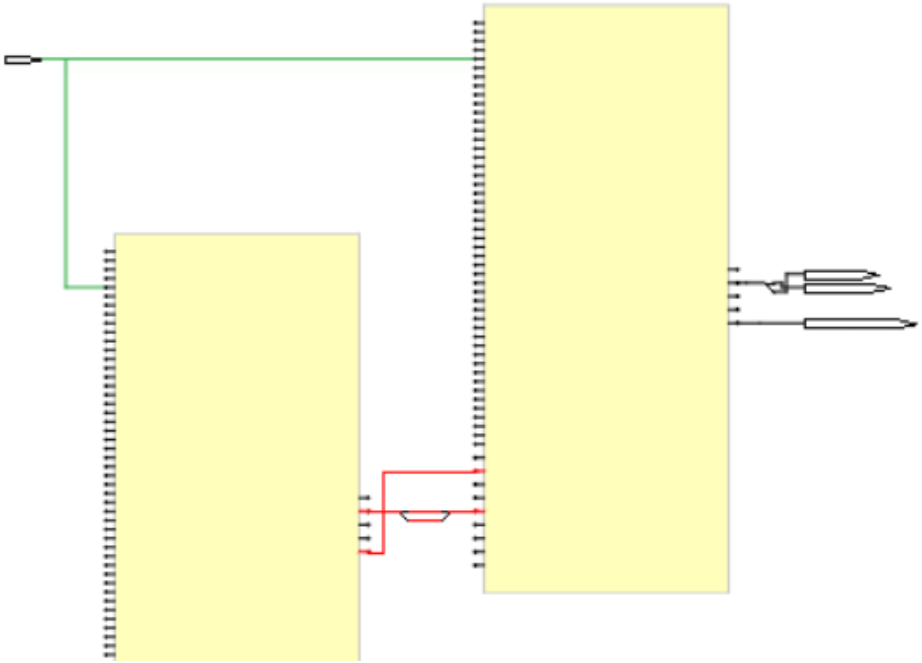


Each Mult-Add block (Yellow blocks) in the above figure is implemented as shown in the following figure:



SRM (Technology) View

Snippet of the MACC_PA_BC_ROM connections with CDOUT-> CDIN and BCOUT->B connections.



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 6 of 924 (0%)
MACC_PA: 1 MultAdd
MACC_PA_BC_ROM: 5 MultAdds

Example 70: 4-tap Symmetric FIR filter

Given below is an example of a 4-tap Symmetric FIR filter which gets completely packed into MACC blocks.

```
module mult_add(clk, rst, en, in1, in2, coef1, coef2, sum_in, sum_out);
parameter din_width = 17;
parameter coef_width = 17;
parameter dout_width = 2*din_width;
input clk, rst, en;
input  [din_width : 0] in1;
input  [din_width : 0] in2;
input  [coef_width - 1 : 0] coef1;
input  [coef_width - 1 : 0] coef2;
input [dout_width  : 0] sum_in;
output [dout_width : 0] sum_out;
reg  [dout_width : 0] sum_out;
reg  [dout_width : 0] sum_out_reg;
wire  [din_width + coef_width : 0] mult1;
```

```
wire  [din_width + coef_width : 0] mult2;
assign mult1 = in1 * coef1;
assign mult2 = in2 * coef2;
always @ (posedge clk)
begin
   sum_out_reg <= sum_in + mult1;
   sum_out <= sum_out_reg + mult2;
end
endmodule
```

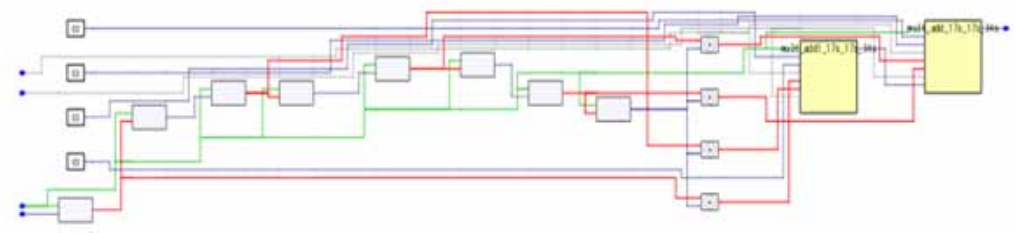
```
module mult_add1(clk, rst, en, in1, in2, coef1, coef2, sum_in, sum_out);
parameter din_width = 17;
parameter coef_width = 17;
parameter dout_width = 2*din_width;
input clk, rst, en;
input  [din_width : 0] in1;
input  [din_width : 0] in2;
input  [coef_width - 1 : 0] coef1;
input  [coef_width - 1 : 0] coef2;
input [dout_width : 0] sum_in;
output [dout_width : 0] sum_out;
reg  [dout_width : 0] sum_out;
reg  [din_width + coef_width : 0] mult1_reg;
wire  [din_width + coef_width : 0] mult1;
wire  [din_width + coef_width : 0] mult2;
assign mult1 = in1 * coef1;
assign mult2 = in2 * coef2;
always @ (posedge clk)
begin
   mult1_reg <= mult1;
   sum_out <= mult1_reg + mult2;
end
endmodule
```

```
module fir(clk, rst, en, Din, Dout);
parameter din_width=17;
parameter coef_width = 17;
parameter dout_width=2*din_width;
parameter TapLen = 4;
input clk, rst, en;
input  [din_width - 1 : 0] Din;
output  [dout_width:0] Dout;
reg  [din_width-1:0] D [TapLen-1:0];
reg  [din_width:0] D_add [TapLen-1:0];
wire  [dout_width:0] sum_tmp [TapLen :0 ];
wire  [coef_width-1:0] b [100:0] /* synthesis syn_keep=1 */;
assign sum_tmp[0] = 0;
generate
begin: CoeffGen
   assign b[0] = 17'b11111100111110000;
   assign b[1] = 17'b11100100000110100;
   assign b[2] = 17'b11100100100010100;
   assign b[3] = 17'b11111100111100010;
   assign b[4] = 17'b11111100111110000;
   assign b[5] = 17'b11110100000110100;
   assign b[6] = 17'b11110100100010101;
```

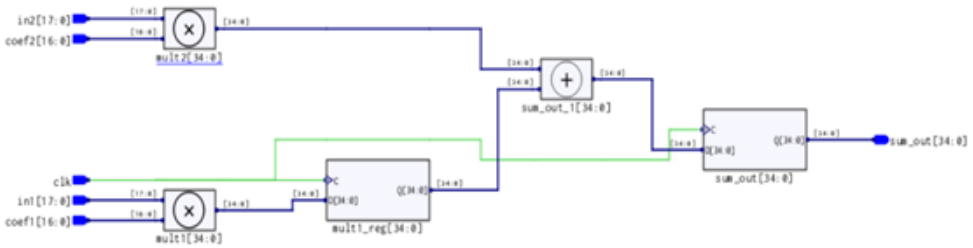
```
   assign b[7] = 17'b01010111111111101;
   assign b[8] = 17'b01010111111111101;
   assign b[9] = 17'b01010111111111101;
   assign b[10] = 17'b01010111111111101;
   assign b[11] = 17'b01011100111110000;
   assign b[12] = 17'b01000100000110100;
   assign b[13] = 17'b01000100100010100;
   assign b[14] = 17'b01011100111100010;
   assign b[15] = 17'b01011100111110000;
   assign b[16] = 17'b01011101111110000;
   assign b[17] = 17'b11111101111110000;
   assign b[18] = 17'b11100101100110100;
   assign b[19] = 17'b11100101100010100;
   assign b[20] = 17'b11111111111100010;
   assign b[21] = 17'b11111110111110000;
   assign b[22] = 17'b11110111000110100;
   assign b[23] = 17'b11110111100010101;
   assign b[24] = 17'b01010100111111101;
   assign b[25] = 17'b01010100111111101;
   assign b[26] = 17'b01010100111111101;
   assign b[27] = 17'b01010101111111101;
   assign b[28] = 17'b01011110111110000;
   assign b[29] = 17'b01000110000110100;
   assign b[30] = 17'b01000111100010100;
genvar j;
for(j = 31; j<= 100; j=j+1)
begin
   assign b[j] = 17'b01011101011100010;
end
end
endgenerate
//Generation of tap delay with additional delay element after every two taps
generate
genvar i;
for(i = 0; i<= TapLen-1; i=i+1)
begin
reg [din_width-1:0] tmp_reg;
begin
   if(i==0)
   begin
         always @ (posedge clk)
         begin
               D[0] <= Din;
         end
   end
   else
   begin
         always @ (posedge clk)
         begin
               D[i] <= tmp_reg;
               tmp_reg <= D[i-1];
            end
         end
      end
```

```
end
reg  [din_width-1:0] D_add_reg;
always @ (posedge clk)
begin
D_add_reg <= D[TapLen-1];
end
for(i = 0; i<= TapLen-1; i=i+1)
begin
assign D_add[i] = D[i] + D_add_reg;
end
endgenerate
generate
genvar n;
for(n=0; n<=TapLen-1; n=n+2)
begin
   if(n==TapLen-1) mult_add #(din_width, coef_width, dout_width) mult_add_2(clk,
rst, en, D_add[n], {din_width{1'b0}}, b[n], b[n+1], sum_tmp[n/2], sum_tmp[n/2+1]);
   else if (n==0) mult_add1 #(din_width, coef_width, dout_width) mult_add_1(clk,
rst, en, D_add[n], D_add[n+1], b[n], b[n+1], 0, sum_tmp[n/2+1]);
   else  mult_add #(din_width, coef_width, dout_width) mult_add(clk, rst, en,
D_add[n], D_add[n+1], b[n], b[n+1], sum_tmp[n/2], sum_tmp[n/2+1]);
end
endgenerate
assign Dout = (TapLen%2) ? sum_tmp[TapLen/2+1] : sum_tmp[TapLen/2];
endmodule
```

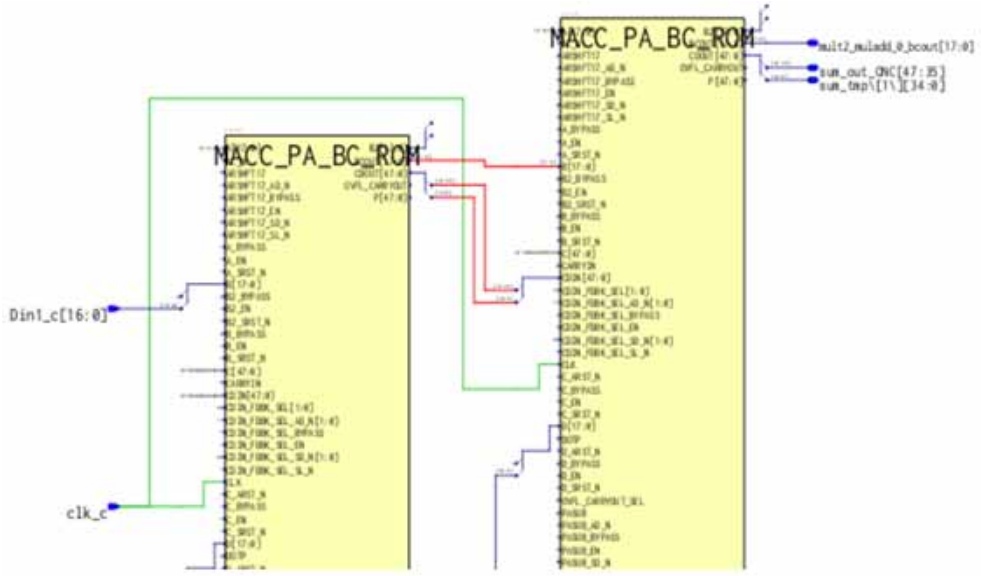
```
module fir_top(clk, rst, en, Din1, Dout1);
parameter din_width = 17;
parameter coef_width = 17;
parameter dout_width = din_width + coef_width;
parameter TapLen = 4;
input clk, rst, en;
input  [din_width - 1 : 0] Din1;
output  [dout_width:0] Dout1;
reg  [din_width - 1 : 0] Din1_reg;
fir #(din_width, coef_width, dout_width,  TapLen) fir1(clk, rst, en, Din1, Dout1);
endmodule
```



Each mult-add block in the above figure (yellow blocks) is implemented as below.



Below is a snippet of the MACC_PA_BC_ROM connections with CDOUT-> CDIN and BCOUT->B connections.

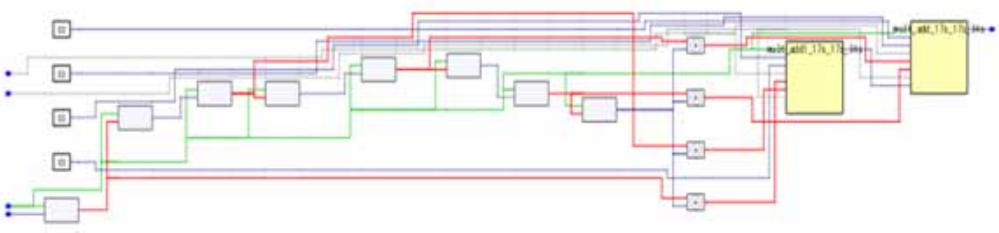


Example 71: 128-tap Symmetric FIR filter

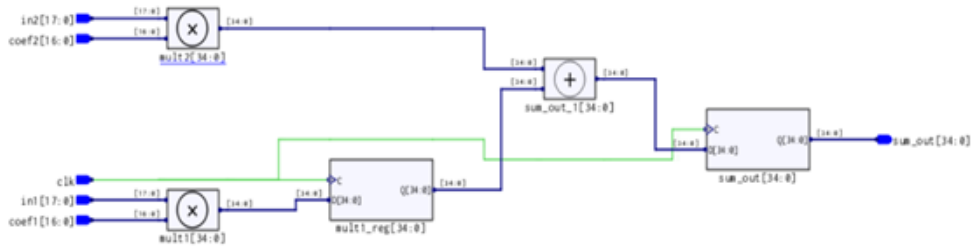
The following example is a 128-tap Symmetric FIR filter which gets completely packed into MACC blocks.

```
module SYM_FIR_RTL (CLK, RESETN, DATAI, COEFI, COEFI_VALID, DATA_VALID, FIRO);
parameter HTAPS      = 64;      // Half the number of filter taps
parameter TAPS       = 2*HTAPS; // Number of filter taps
parameter DATA_WIDTH = 16;      // Data width
parameter COEF_WIDTH = 16;      // Coefficient width
parameter OUT_WIDTH  = 48;      // Output width
// Input signals
// Clock and Sync-reset
input CLK;
input RESETN;
// Input Data and Coefficient
input signed [DATA_WIDTH-1:0] DATAI;
input signed [COEF_WIDTH-1:0] COEFI;
// Valid signals for Data and Coefficient
input COEFI_VALID;
input DATA_VALID;
// Filter output
output reg signed [OUT_WIDTH-1:0] FIRO;
// Data and Coefficient registers.
reg signed [COEF_WIDTH-1:0] coeffreg   [HTAPS:0];
reg signed [DATA_WIDTH-1:0] sample_data[TAPS-1:0];
// Filter output register
reg signed [OUT_WIDTH-1:0] FIR_DP [HTAPS-1:0];
integer i, k;
// Load Coefficients
always @( posedge CLK )
begin
   if ( RESETN )
   begin
      if ( COEFI_VALID )
      begin
         coeffreg[HTAPS] <= COEFI;
         for ( i = HTAPS-1; i >= 0; i = i-1 )
         begin
            coeffreg[i] <= coeffreg[i+1];
         end
      end
   end
   else
   begin
      for ( i = 0; i < HTAPS; i = i+1 )
      begin
         coeffreg[i] <= 0;
      end
   end
end
```

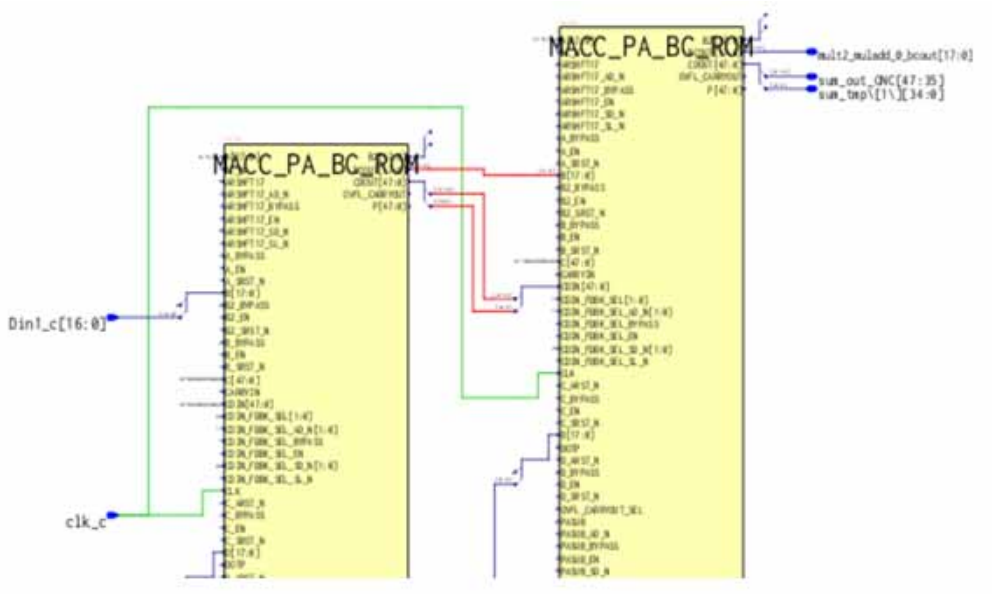
```
// Load Data
always @( posedge CLK )
begin
   if ( RESETN )
    begin
      if ( DATA_VALID )
   begin
      sample_data[0] <= DATAI;
      for ( k = 1; k < TAPS; k = k+1 )
      begin
         sample_data[k] <= sample_data[k-1];
      end
   end
end
else
begin
   for ( k = 0; k < TAPS; k = k+1 )
   begin
      sample_data[k] <= 0;
   end
end
end
// Symmetric Filter operation
always @( posedge CLK )
begin
   for ( i = 0; i < HTAPS; i=i+1 )
   begin
      if ( i == 0 )
      begin
      FIR_DP[i] <= (sample_data[i*2] + sample_data[TAPS-1]) * coeffreg[HTAPS-i];
      end
      else
      begin
      FIR_DP[i] <= FIR_DP[i-1] + ((sample_data[i*2] + sample_data[TAPS-1]) *
coeffreg[HTAPS-i]);
      end
   end
end
// Load to output register
always @( posedge CLK )
begin
   FIRO <= FIR_DP[HTAPS-1];
end
endmodule
```



Each mult-add block in the above figure (yellow blocks) is implemented as below.



The following snippet shows the MACC_PA_BC_ROM connections with CDOUT-> CDIN and BCOUT->B connections.



Inferring MACC_PA Block in SIMD mode

This section contains the following examples:

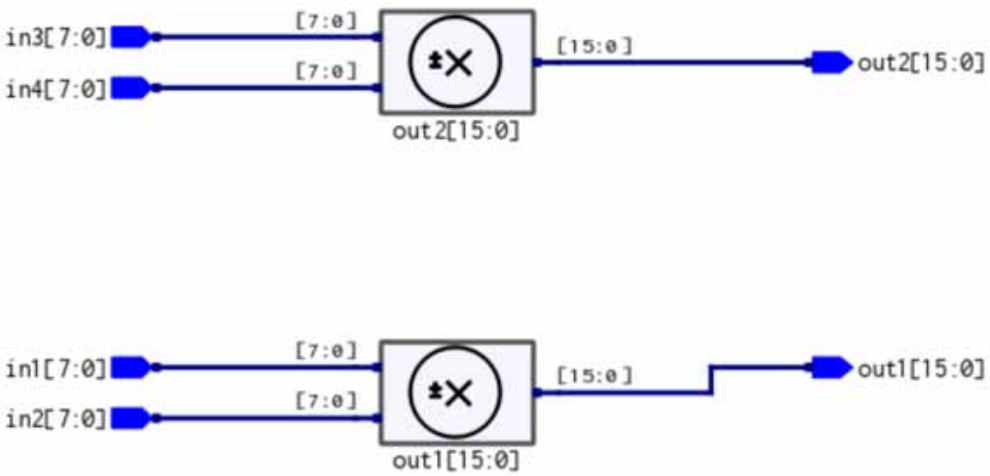
- Example 72: syn_multstyle set as "simd:1" on a pair of multipliers, on page 179
- Example 73: syn_multstyle set as "simd:n" on 3 pairs of multipliers, on page 181

Example 72: syn_multstyle set as "simd:1" on a pair of multipliers

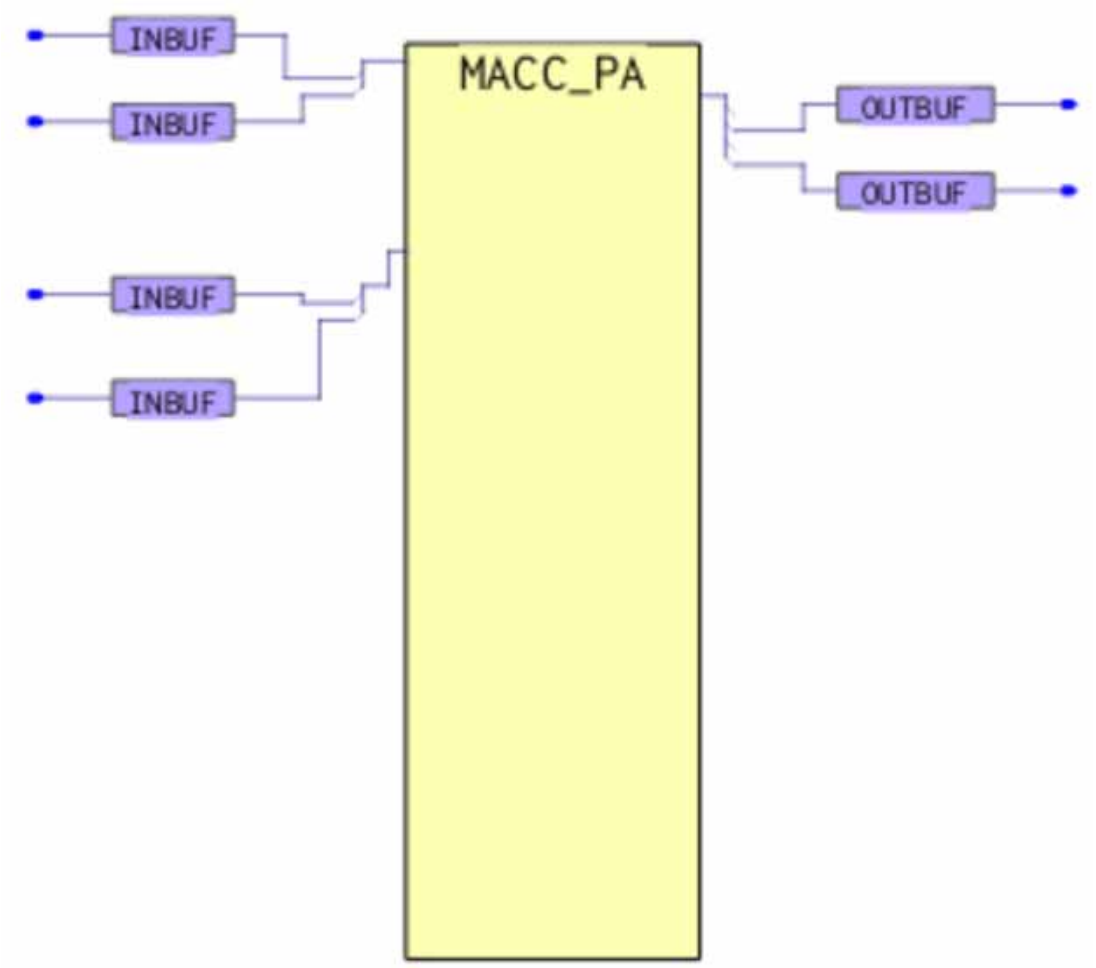
RTL

```
module top(in1, in2, in3, in4, out1, out2);

input signed [7:0] in1, in3;
input signed [7:0] in2, in4;

output signed [15:0] out1/* synthesis syn_multstyle = "simd:1" */;
output signed [15:0] out2/* synthesis syn_multstyle = "simd:1" */;

assign out1 = in1 * in2;
assign out2 = in3 * in4;

endmodule
```

SRS (RTL View)



SRM (Technology View)



Resource Usage

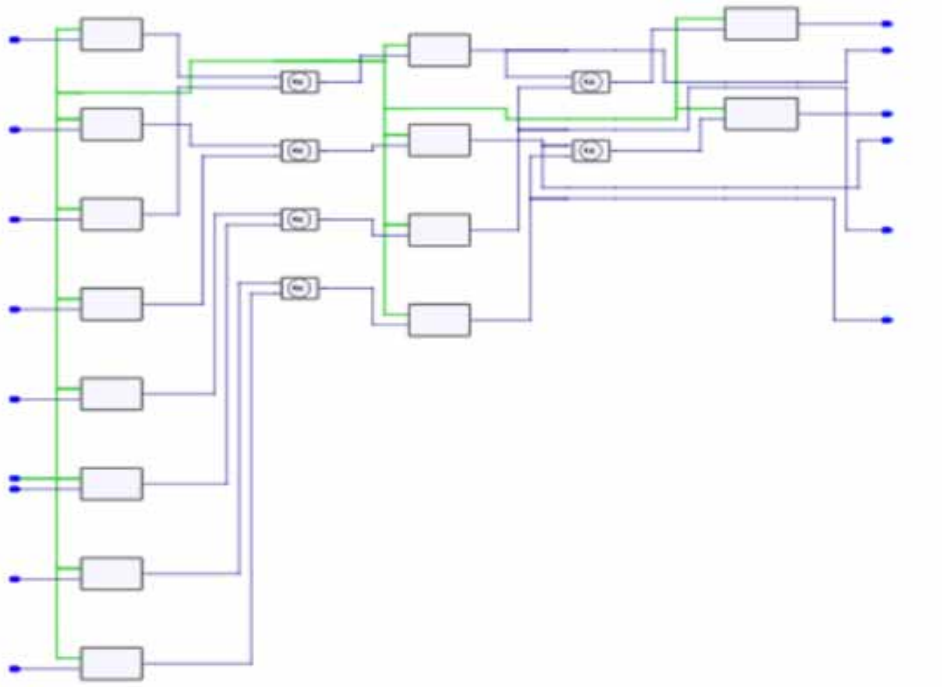
Sequential Cells:
SLE 0 uses
DSP Blocks: 1 of 924 (0%)
MACC_PA: 1 Mult
Total LUTs: 0

Example 73: syn_multstyle set as "simd:n" on 3 pairs of multipliers

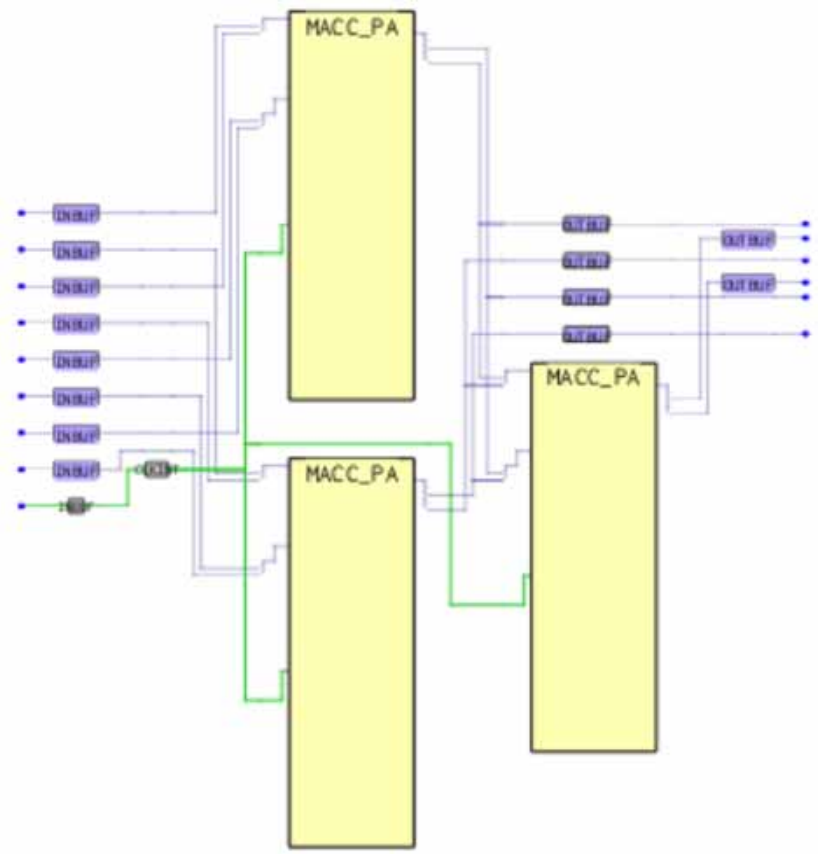
RTL

```
module top( a1,b1,c1,d1,a2,b2,c2,d2,x1,x2,x3,x4,p1,p2, clk);
input clk;
input signed [3:0] a1,b1,c1,d1,a2,b2,c2,d2;
output signed [7:0] x1 /* synthesis syn_multstyle = "simd:1" */;
output signed [7:0] x2 /* synthesis syn_multstyle = "simd:1" */;
output signed [7:0] x3 /* synthesis syn_multstyle = "simd:2" */;
output signed [7:0] x4 /* synthesis syn_multstyle = "simd:2" */;
output signed [15:0] p1 /* synthesis syn_multstyle = "simd:3" */;
output signed [15:0] p2 /* synthesis syn_multstyle = "simd:3" */;
reg signed [3:0] a1_reg ;
reg signed [3:0] b1_reg ;
reg signed [3:0] c1_reg ;
reg signed [3:0] d1_reg ;
reg signed [3:0] a2_reg ;
reg signed [3:0] b2_reg ;
reg signed [3:0] c2_reg ;
reg signed [3:0] d2_reg ;
reg signed [7:0] x1;
reg signed [7:0] x2;
reg signed [7:0] x3;
reg signed [7:0] x4;
reg signed [15:0] p1 ;
reg signed [15:0] p2 ;

always@(posedge clk)
begin
   a1_reg <= a1;
a2_reg <= a2;
   b1_reg <= b1;
b2_reg <= b2;
   c1_reg <= c1;
c2_reg <= c2;
   d1_reg <= d1;
d2_reg <= d2;
x1<= a1_reg * b1_reg;
x2<= c1_reg * d1_reg;
x3 <= a2_reg * b2_reg;
x4 <= c2_reg * d2_reg;
p1<= x1 * x2;
p2<= x3 * x4;
end
endmodule
```

SRS (RTL View)



SRM (Technology View)



Resource Usage

Sequential Cells:
SLE 0 uses
DSP Blocks: 3 of 68 (4%)
MACC_PA: 3 Mults
Total LUTs: 0

Limitations

For successful PolarFire Math block 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 6 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.
- DOTP mode RTL as below:

```
module test(clk,A,B,C,D,F,out,addr );
input clk;
input signed [8:0] A;
input signed [8:0] B;
input signed [8:0] C;
input signed [8:0] D;
input signed [34:0] F;
input [3:0] addr;
output [34:0] out;
reg signed [17:0] rom_out;
reg signed [34:0] out;
always@(posedge clk)
begin
case (addr)
4'd0 : rom_out <= 18'b011000001000110010;
4'd1 : rom_out <= 18'b000011011001110010;
4'd2 : rom_out <= 18'b010110111101100110;
4'd3 : rom_out <= 18'b101001100000000010;
4'd4 : rom_out <= 18'b011010011111110010;
4'd5 : rom_out <= 18'b010100001000101010;
4'd6 : rom_out <= 18'b101011010100011010;
4'd7 : rom_out <= 18'b010010010100101011;
4'd8 : rom_out <= 18'b011101110001111111;
4'd9 : rom_out <= 18'b100101011111101101;
4'd10 : rom_out <= 18'b010001100000100011;
4'd11 : rom_out <= 18'b111000001101100111;
4'd12 : rom_out <= 18'b101101011011110111;
4'd13 : rom_out <= 18'b011110100110000011;
4'd14 : rom_out <= 18'b001101101001111101;
4'd15 : rom_out <= 18'b111101010000100011;
default : rom_out <= 18'b000100101101011011;
endcase
out <= (rom_out[17:9] * A) + (rom_out[8:0] * B) ;
end
endmodule
```

- Multiplier Accumulator with Dynamic Add-Sub as below:

```
module test(
   input                CLK,
   input  signed [17:0] A,
   input  signed [17:0] B,
   input  signed [47:0] C,
   input                SUB,
   input                P_SRST_N,
   output signed [47:0] P
);
   reg signed [47:0] P_reg;
   always @( posedge CLK )
   begin
      if ( ~P_SRST_N )
         P_reg <= 48'b0;
      else if ( SUB )
         P_reg <= P_reg + C - (B * A);
      else
         P_reg <= P_reg + C + (B * A);
   end
   assign P = P_reg;
endmodule
```