Inferring Actel RTAX-DSP MATH Blocks

Actel® RTAX-DSP devices support 18x18-bit signed multiply-accumulate blocks. The architecture includes dedicated components called RTAX-DSP MATH blocks, which can perform DSP-related operations like multiplication followed by addition, multiplication followed by subtraction, and multiplication with accumulate. This application note provides a general description of the Actel RTAX-DSP MATH block component and shows you how to infer it with the Synplify® Pro synthesis tool.

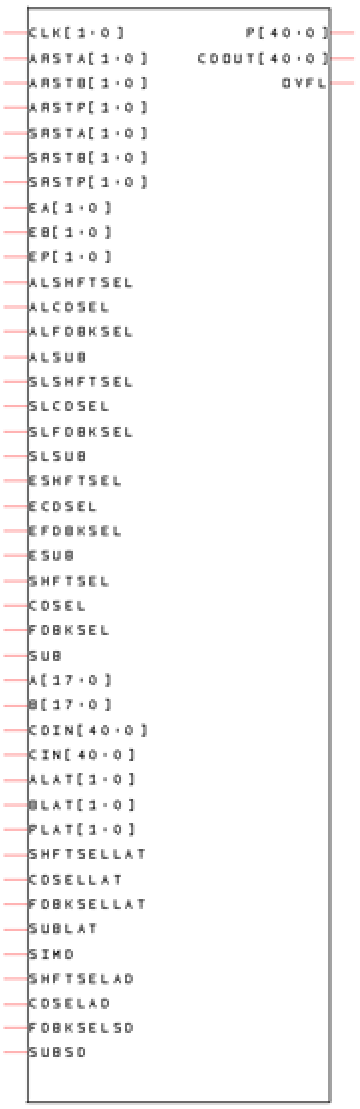
The application note describes the following:

The RTAX-DSP MATH Block

The Actel Axcelerator RTAX2000D and RTAX4000D devices support 18x18-bit signed multiply-accumulate RTAX-DSP MATH blocks. The multiplier takes two 18-bit signed signals and multiplies them for a 36-bit result. The result is then extended to 41 bits. In addition to multiplication followed by addition and multiplication followed by subtraction, the blocks can also accumulate the current multiplication product with a previous result, a constant, a dynamic value or a result from another RTAX-DSP MATH block.

All the signals of the RTAX-DSP MATH block (except CIN, CDIN and CDOUT) have optional registers. All the registers must use the same clock. Each of the registers has enables and resets, which can differ from each other.

The following figure shows the 18 x 18-bit Actel RTAX-DSP MATH block. For a complete list of all the block options and their configurations, refer to the Actel documentation.



Inferring RTAX-DSP MATH Blocks

Starting with the C2009.03A-2 version of the Synplify Pro synthesis tool, you can now infer RTAX-DSP MATH block components. This support is not available in the C-2009.06 multi-vendor version of the tool, but will be available in future multi-vendor releases.

You can write your RTL so that the synthesis tool recognizes the structures and maps them to RTAX-DSP MATH components. Currently the Synplify Pro tool extracts the following logic structures from the hardware description and maps them to RTAX-DSP MATH blocks.

- Multipliers
- Mult-adds (multiplier followed by an adder)
- Mult-subs (multiplier followed by a subtractor)

Note the following:

- The synthesis tool supports the inference of both signed and unsigned multipliers.
- The Actel RTAX-DSP 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 RTAX-DSP MATH blocks, as described in *Inferring RTAX-DSP MATH Blocks for Wide Multipliers*, on page 19.
- Starting with the C2009.06A version, the Synplify Pro synthesis tool supports the inference of RTAX-DSP MATH block components across different hierarchies. The multiplier, input registers, output registers, and subtractor/adders are packed into the same RTAX-DSP MATH block, even if they are in different hierarchies.
- The synthesis tool packs registers at the inputs and outputs of multiplier/multiplier-adder/multiplier-subtractor into RTAX-DSP MATH blocks.

  By default, the tool maps all multiplier inputs with a width of 3 or greater to RTAX-DSP 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 multipliers/multiplier-adders/multiplier-subtractors into RTAX-DSP 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 get priority, and are packed in the RTAX-DSP MATH block.
- 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 RTAX-DSP MATH blocks, as described in *Inferring RTAX-DSP MATH Blocks for Wide Multipliers*, on page 19.
- The synthesis tool packs different kinds of flip-flops at the inputs/outputs of the multiplier/multiplier- adder/multiplier-subtractor into RTAX-DSP 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

Controlling Inference with the syn_multstyle Attribute

Use the syn_multstyle attribute to control the inference of RTAX-DSP MATH blocks. By default, multipliers with input widths of 3 or greater are packed in the RTAX-DSP MATH block, while smaller input widths are mapped to logic.

There are two values for the syn_multstyle attribute: logic or dsp. You can apply the attribute globally or to individual modules, as the following sdc syntax examples illustrate:

```
define_global_attribute syn_multstyle {dsp|logic}
```

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

If the multipliers are inferred as RTAX-DSP 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 [9:0] 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 multiplier structures to RTAX-DSP MATH blocks.

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

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

For more information and examples, refer to the *FPGA Synthesis Reference Manual*.

Coding Style Examples

The following code examples demonstrate structures that the synthesis tool can implement in RTAX-DSP MATH blocks. There are many ways to code your DSP structures, but the synthesis tool does not map all of them to RTAX-DSP MATH blocks. The examples provided illustrate coding styles from which the synthesis tool can infer and implement RTAX-DSP MATH blocks. It is important that you use a supported coding structure so that the synthesis tool infers the RTAX-DSP 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 whether they are configured as Mult, MultAdd, or MultSub blocks. You should also check the final netlist to make sure that the structures you want were implemented.

See the following for examples of coding style:

- Example 1: 6x6-Bit Unsigned Multiplier, on page 5
- Example 2: 11x 9-Bit Signed Multiplier, on page 7
- Example 3: 18x18-Bit Signed Multiplier with Registered I/Os, on page 8
- Example 4: 17x17-Bit Unsigned Multiplier with Different Resets, on page 10
- Example 5: Unsigned Multiplier with Registered I/O and Different Clocks, on page 11
- Example 6: Multiplier-Adder, on page 13
- Example 7: Multiplier-Subtractors, on page 15

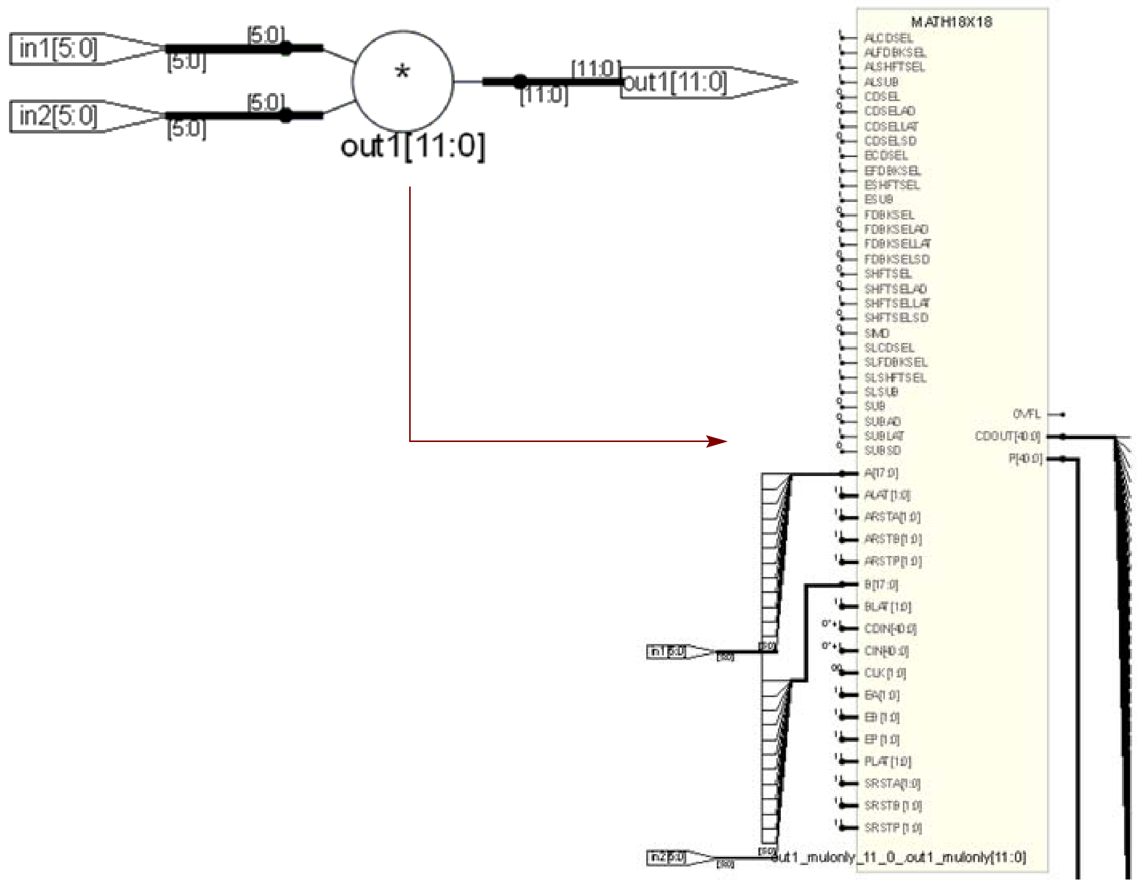
For other examples, see *Wide Multiplier Coding Examples, on page 23*.

Example 1: 6x6-Bit Unsigned Multiplier

The following design is a simple 6x6-bit unsigned multiplier:

```
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;
```

The FPGA synthesis tool maps this multiplier into one RTAX-DSP MATH block as shown below:



Resource Usage Report for Unsigned 6x6-Bit Multiplier

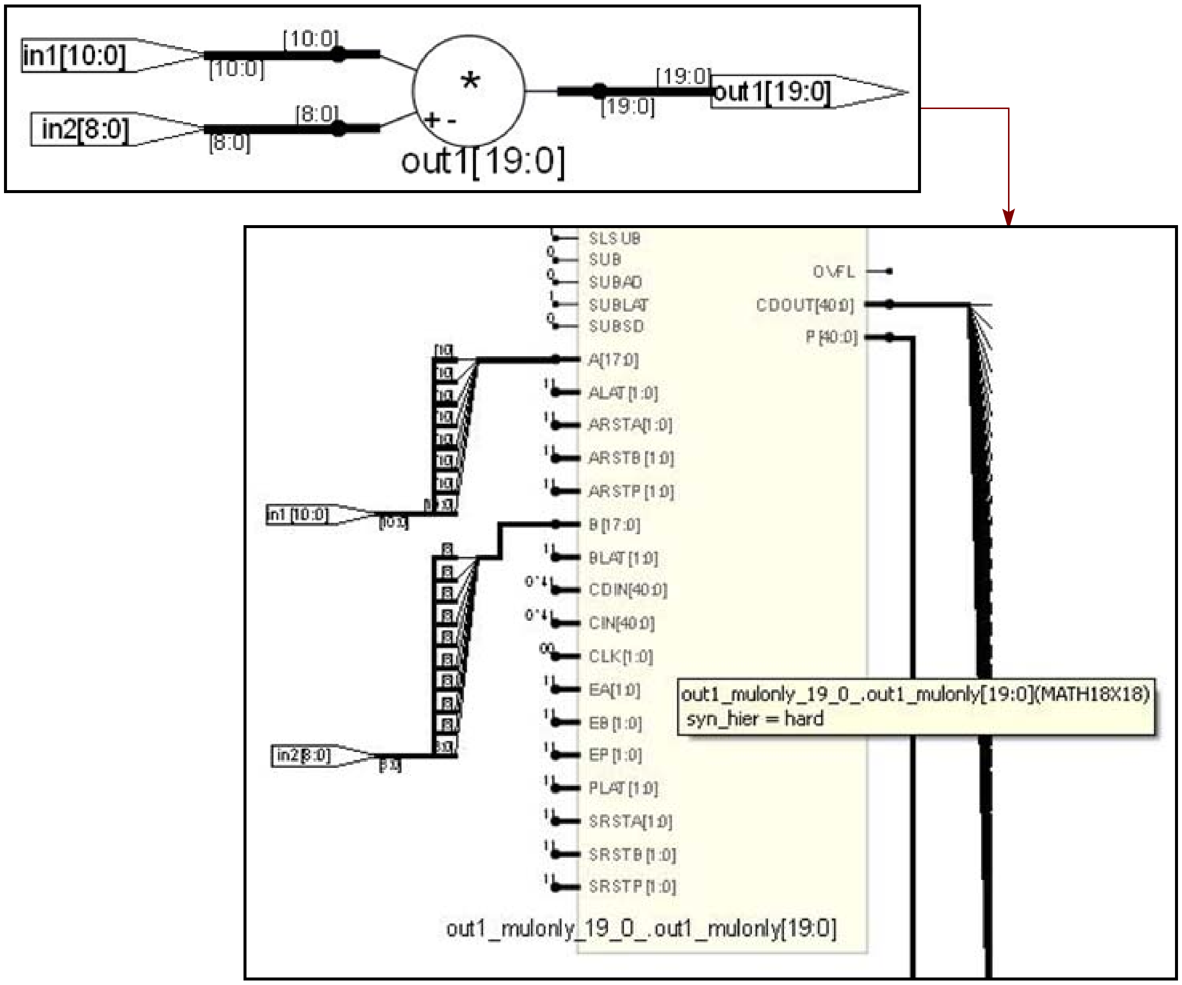
This is an extract from the log file (`.srr`) and shows resource usage details. It shows that the multiplier code was implemented in one RTAX-DSP MATH Mult block.

```
Target Part: rtax2000d_ccga/lga1272-s
Combinational Cells:    0 of 19712 (0%)
Sequential Cells:    0 of 9856 (0%)
Total Cells:         0 of 29568 (0%)
DSP Blocks:          1
Clock Buffers:       0
IO Cells:            0

Details:
   MATH18X18:      1    Mults
```

Example 2: 11x 9-Bit Signed Multiplier

This example is a 11x 9-bit signed multiplier. It gets mapped into one RTAX-DSP MATH block, as shown in the figure.

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

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

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

end behav;
```



Resource Usage Report for 11x9-Bit Signed Multiplier

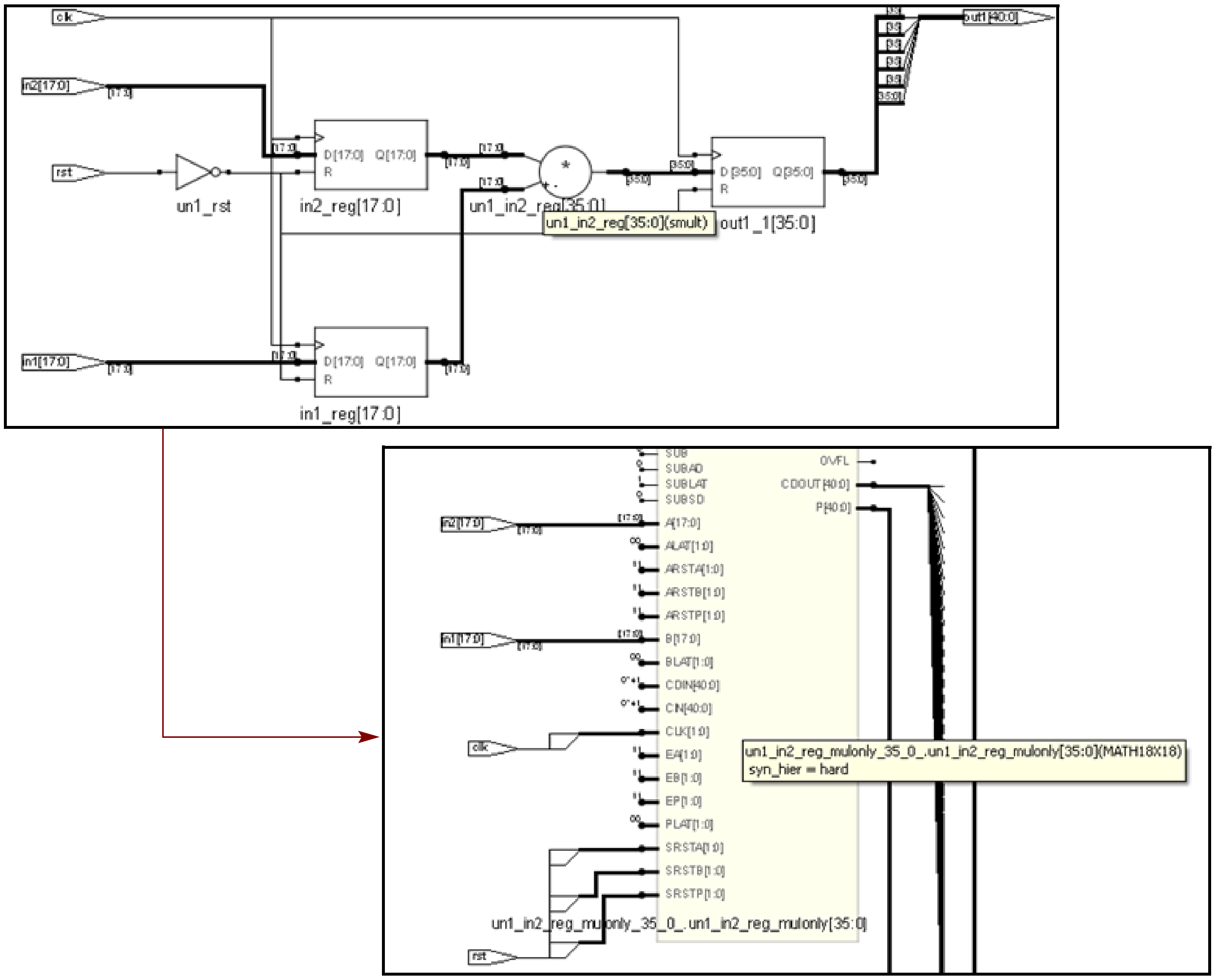
```
Target Part: rtax4000d_ccga/lga1272-s
Combinational Cells:    0 of 36960 (0%)
Sequential Cells:    0 of 18480 (0%)
Total Cells:         0 of 55440 (0%)
DSP Blocks:          1
Clock Buffers:       0
IO Cells:            0

Details:
   MATH18X18:      1    Mults
```

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.

```
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
```

The synthesis tool fits all this logic into one RTAX-DSP MATH block, as shown below.



Resource Usage Report for Signed 18x18-Bit Multiplier with Registered I/Os

```
Target Part: rtax4000d_ccga/lga1272-s
Combinational Cells:    0 of 36960 (0%)
Sequential Cells:    0 of 18480 (0%)
Total Cells:         0 of 55440 (0%)
DSP Blocks:          1
Clock Buffers:       0
IO Cells:            0

Details:
   MATH18X18:      1    Mults
```

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

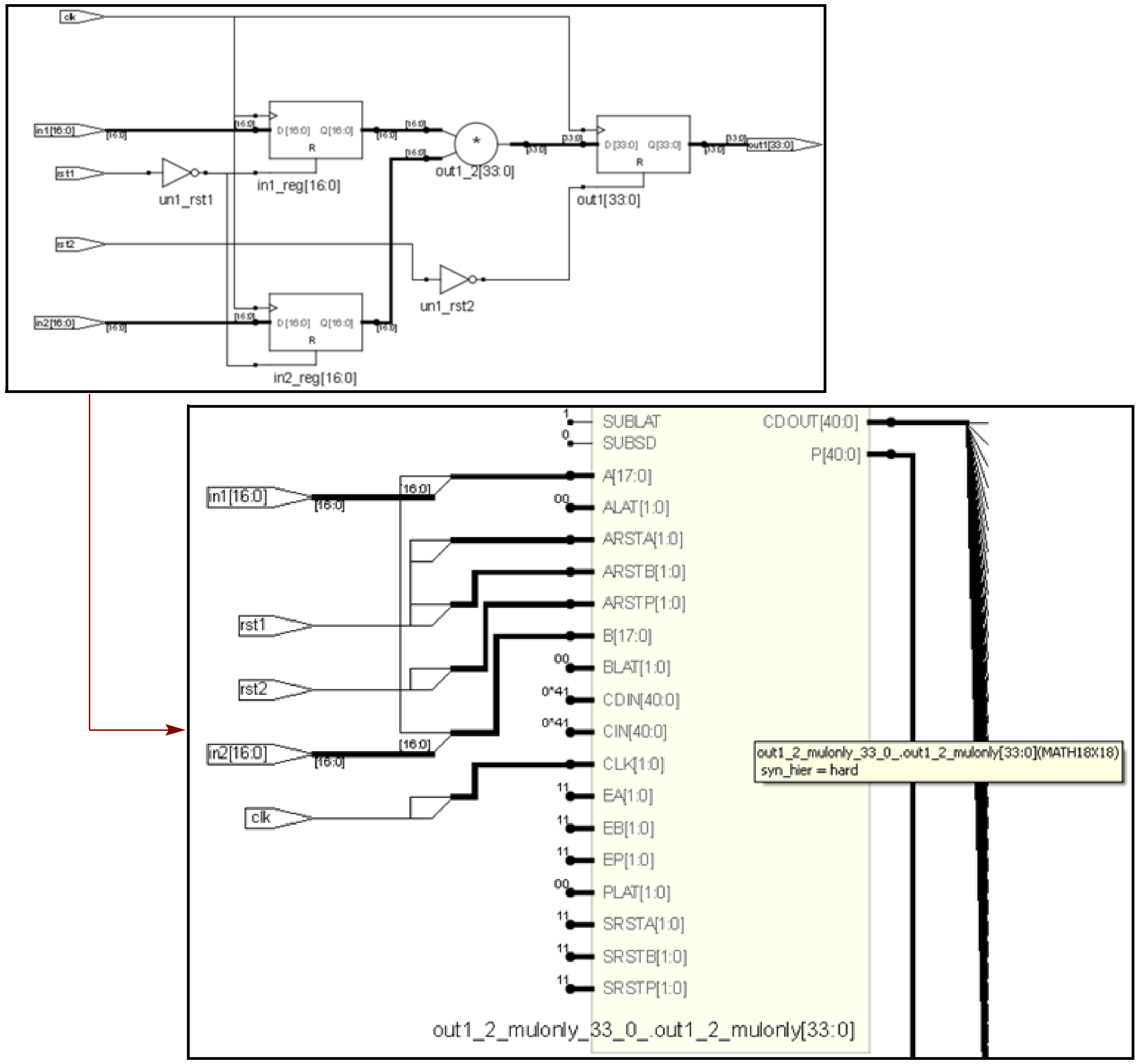
This is a VHDL example of a 17x17-bit unsigned multiplier, which has input and output registers with different asynchronous resets.

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

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

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

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

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

end behav;
```

Resource Usage Report for Unsigned 17x17-Bit Multiplier

```
Target Part: rtax4000d_ccga/lga1272-s
Combinational Cells:    0 of 36960 (0%)
Sequential Cells:    0 of 18480 (0%)
Total Cells:         0 of 55440 (0%)
DSP Blocks:          1
Clock Buffers:       0
IO Cells:            0

Details:
   MATH18X18:      1    Mults
```

The tool packs all the logic into one RTAX-DSP MATH block as shown below.



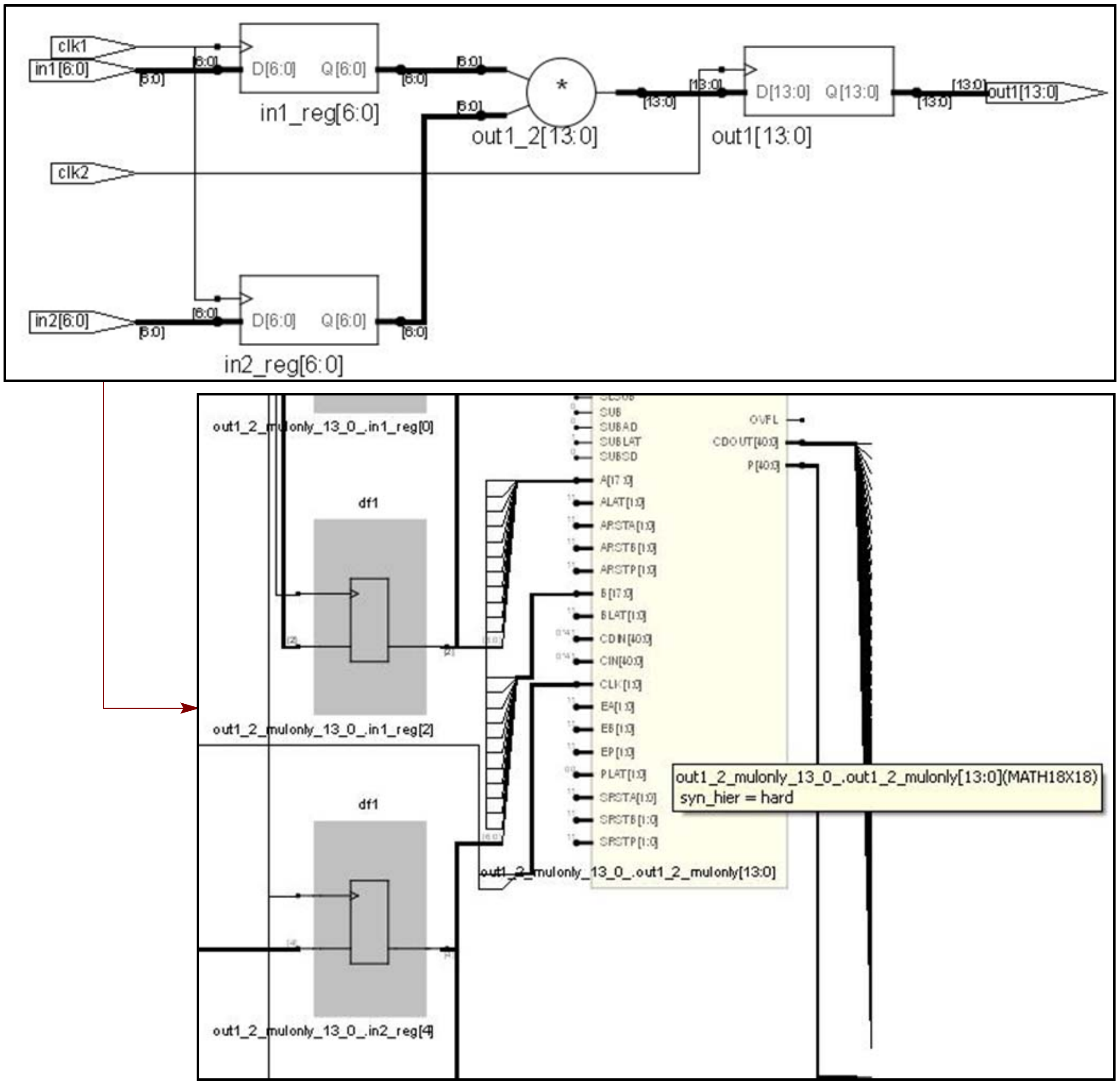
Example 5: Unsigned Multiplier with Registered I/O and Different Clocks

This example shows an unsigned multiplier whose input and outputs are registered with different clocks: clk1 and clk2, respectively.

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

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

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

endmodule
```

In a case like this one, the synthesis tool only packs the output registers and the multiplier into the RTAX-DSP MATH blocks. The input registers are implemented as logic outside the RTAX-DSP MATH block.



Resource Usage Report for Unsigned Multiplier with Different Clocks

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

```
Target Part: rtax4000d_ccga/lga1272-s
Combinational Cells:    1 of 36960 (0%)
Sequential Cells:    14 of 18480 (0%)
Total Cells:         15 of 55440 (1%)
DSP Blocks:          1
Clock Buffers:       0
IO Cells:            0

Details:
   buff:           1    comb:1
   df1:            14   seq:1
   MATH18X18:      1    Mults
```

Example 6: Multiplier-Adder

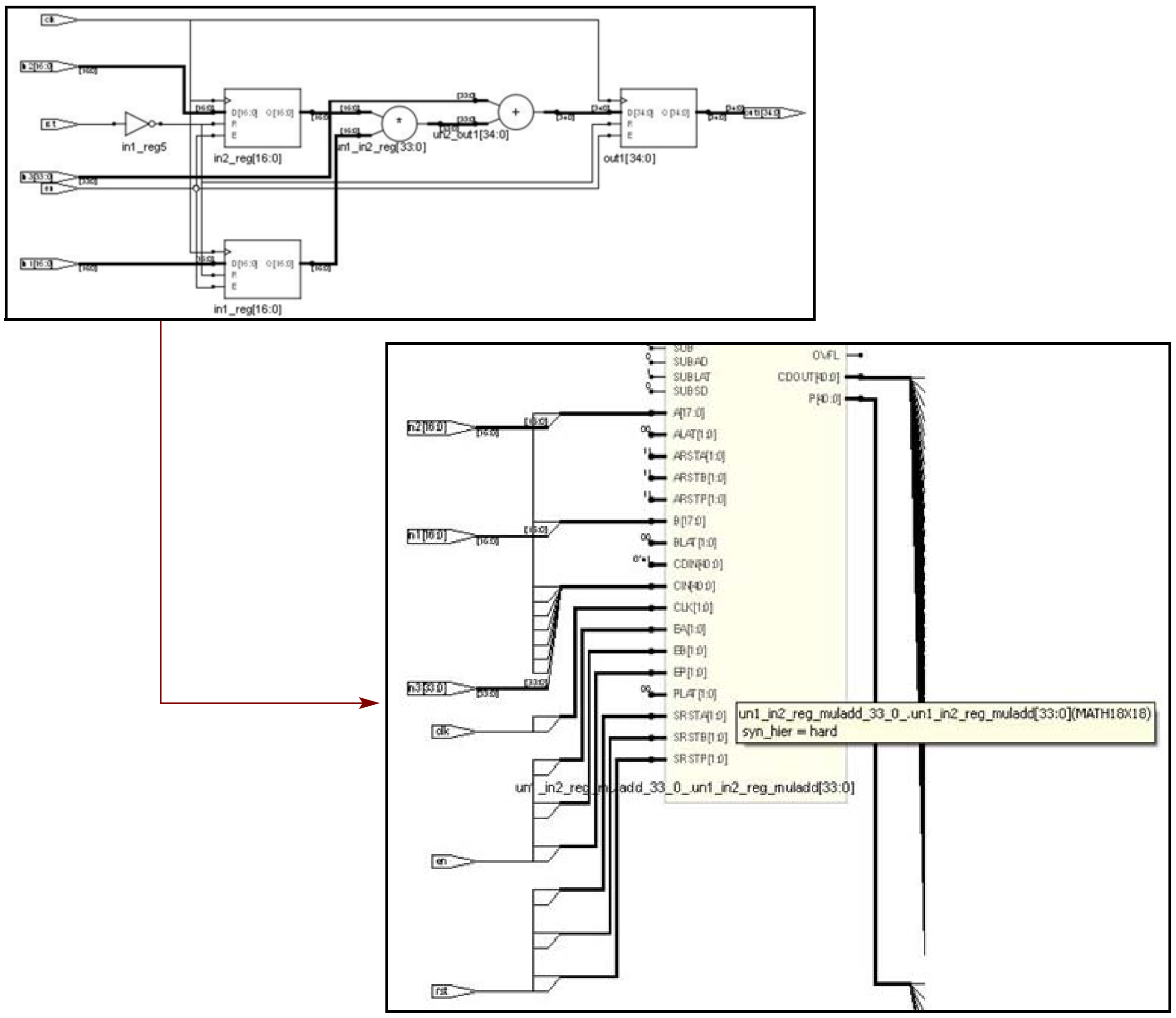
This VHDL example shows a multiplier whose output gets added with another input. The inputs and outputs are registered, and have enables and synchronous resets.

```
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;
```

Note that the RTAX-DSP MATH block does not support registered CIN inputs. In this example, if in3 is registered, then the synthesis tool infers flop primitives for in3 outside the RTAX-DSP MATH block and packs the remaining logic into the block.

The following figure shows how the design gets mapped into a RTAX-DSP MATH block.



Resource Usage Summary for Multiplier-Adder

```
Target Part: rtax2000d_ccga/lga1272-s
Combinational Cells:    0 of 19712 (0%)
Sequential Cells:    0 of 9856 (0%)
Total Cells:         0 of 29568 (0%)
DSP Blocks:          1
Clock Buffers:       0
IO Cells:            0

Details:
   MATH18X18:      1    MultAdds
```

Example 7: Multiplier-Subtractors

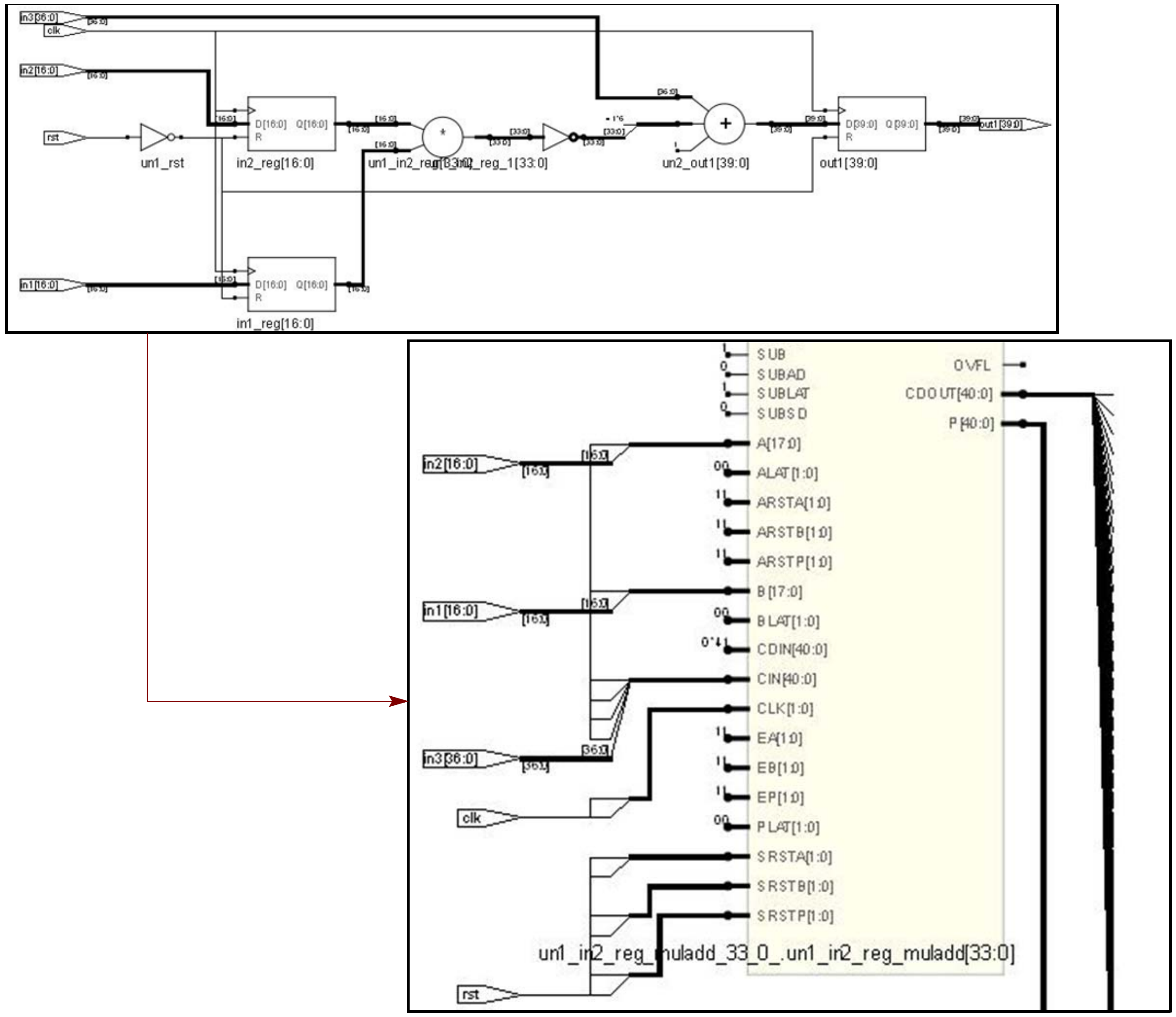
There are two ways to implement multiplier and subtract logic. The synthesis tool packs the logic differently, depending on which way it is implemented.

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

Unsigned MultSub Verilog Example (P = Cin - Mult)

```
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
```

The following figure shows how all the logic is mapped into the RTAX-DSP MATH block



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

The log file resource usage report shows that everything is packed into one RTAX-DSP MATH block, and one MultSub is inferred.

```
Target Part: rtax2000d_ccga/lga1272-s
Combinational Cells:    0 of 19712 (0%)
Sequential Cells:    0 of 9856 (0%)
Total Cells:         0 of 29568 (0%)
DSP Blocks:          1
Clock Buffers:       0
IO Cells:            0

Details:
   MATH18X18:      1    MultSubs
```

Signed MultSub VHDL Example (P = Cin - Mult)

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

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

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

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

The log file resource usage report shows that everything is packed into one RTAX-DSP MATH block, and one MultSub is inferred.

```
Target Part: rtax2000d_ccga/lga1272-s
Combinational Cells:    2 of 19712 (0%)
Sequential Cells:    0 of 9856 (0%)
Total Cells:         2 of 29568 (1%)
DSP Blocks:          1
Clock Buffers:       0
IO Cells:            0

Details:
   buff:            2    comb:1
   MATH18X18:       1    MultSubs
```

Signed MultSub Verilog Example (P = Mult - Cin)

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

always @ ( posedge clk )
begin
   if ( ~rst )
   begin
      in1_reg <= 17'b0;
      in2_reg <= 17'b0;
      out1 <= 40'b0;
   end
   else
   begin
```

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

      endmodule
```

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

In this case, the log file shows that only the multiplier and input registers are mapped to the RTAX-DSP MATH block. The subtractor and output registers are mapped to logic.

```
Target Part: rtax2000d_ccga/lga1272-s
Combinational Cells:    90 of 19712 (0%)
Sequential Cells:    40 of 9856 (0%)
Total Cells:         130 of 29568 (1%)
DSP Blocks:          1
Clock Buffers:       0
IO Cells:            0

Details:
   and2:           40   comb:1
   buff:           9    comb:1
   inv:            1    comb:1
   sub1:           40   comb:1

   df1:            40   seq:1

   MATH18X18:      1    Mults
```

Unsigned MultSub VHDL Example (P = Mult - Cin)

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

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

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

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

In this case, the log file shows that only the multiplier is mapped to the RTAX-DSP MATH block. The subtractor is mapped to logic.

```
Target Part: rtax2000d_ccga/lga1272-s
Combinational Cells:    35 of 19712 (0%)
Sequential Cells:    0 of 9856 (0%)
Total Cells:         35 of 29568 (1%)
DSP Blocks:          1
Clock Buffers:       0
IO Cells:            0

Details:
   add1:           18   comb:1
   inv:            17   comb:1

   MATH18X18:      1    Mults
```

Inferring RTAX-DSP MATH Blocks for Wide Multipliers

Starting with the C2009.06A version, the Synplify Pro synthesis tool fractures wide multipliers and packs them into multiple RTAX-DSP MATH block components, using the cascade and shift functions of the RTAX-DSP MATH block.

A wide multiplier is one where the width of any of its inputs is larger than 18 bits (signed) or 17 bits (unsigned). 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 RTAX-DSP MATH blocks, using the CDOUT and CDIN pins to propagate the cascade output of result P from one RTAX-DSP MATH block to the cascade input for operand C to the next RTAX-DSP MATH block. The tool also performs the appropriate shifting.

The maximum size for a cascaded RTAX-DSP MATH block structures is 52-bit x 52-bit signed multipliers or 51-bit x 51-bit unsigned multipliers. If the multiplier input widths exceed these limits, the tool first fractures the multiplier and determines the exact number of RTAX-DSP MATH block structures needed. (See *Example 12: 69x53-Bit Signed Multiplier (Inputs Wider than 52 Bits)*, on page 27 for an example.)

If a and b are the inputs of such a mult, they are fractured as follows:

```
B1 = 51-bit x 51-bit
B2 = (a - 51) -bit x 51-bit
B3 = 51-bit x (b - 51)-bit
B4 =  (a - 51) -bit x (b - 51)-bit
```

A 60x60 signed multiplier is first fractured into multiple blocks:

```
B1 = 51-bit  x 51-bit
B2 = (60 - 51)-bit  x 51-bit = 9-bit x 51-bit
B3 = 51-bit x (60 - 51)-bit  = 51-bit x 9-bit
B4 = (60 -51)-bit x (60 - 51)-bit = 9-bit x 9-bit
```

Each block is further fractured using the algorithm described in *Fracturing Algorithm*, on page 20.

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 and less than or equal to 51 bits
- For signed multipliers, either m or n or both must be greater than 18 bits and less than or equal to 52 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 multipliers 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 RTAX-DSP MATH block.

Mapping Fractured Multipliers

The following sections describe how the tool maps fractured multipliers with input widths less than or greater than 51 bits (unsigned) or 52 bits (signed).

Multipliers with Input Widths Less than 51/52 Bits (Unsigned/Signed)

When an unsigned multiplier with an input width between 18 and 51 bits or a signed multiplier with an input width between 19 and 52 bits is fractured into multiple multipliers, these multipliers are always packed in multiple RTAX-DSP 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 RTAX-DSP 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 RTAX-DSP MATH blocks using cascade and shift functions. See *Example 8: Unsigned 20x17-Bit Multiplier (One Wide Input)*, on page 23 and *Example 9: 21x18-Bit Signed Multiplier (One Wide Input)*, on page 24 for examples.

- Both inputs wide

  If both inputs are candidates for fracturing, they are fractured according to the fracturing algorithm.

A 51x26 wide multiplier is fractured as follows:

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

Mult2 & Mult4 are further fractured:

Mult2	**Mult4**
`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, which are packed into 6 RTAX-DSP MATH blocks using cascade and shift functions. See *Example 10: Unsigned 26x26-Bit Multiplier (Two Wide Inputs),* on page 25 and *Example 11: 35x35-Bit Signed Multiplier (Two Wide Inputs),* on page 26 for examples.

Multipliers with Input Widths Greater than 51/52 Bits (Unsigned/Signed)

When a multiplier with input width greater than 51 bits (unsigned) or 52 bits (signed) is fractured into many small multipliers, the width of these multipliers determines whether they get packed in a single RTAX-DSP MATH block or in multiple blocks using cascade and shift functions. Fractured multipliers are mapped based on the following rules:

- Fractured multipliers with input widths less than 17 bits (unsigned) or less than 18 bits (signed), are packed into a single RTAX-DSP MATH block.
- If the input width of a fractured multiplier is between 17 and 51 bits (unsigned) or 18 and 51 bits (signed), the tool packs them into multiple RTAX-DSP MATH blocks using cascade and shift functions.
- If the input width of a fractured multiplier is less than 3, it is mapped to logic by default. You can override this behavior with the syn_multstyle attribute, as described in *Controlling Inference with the syn_multstyle Attribute,* on page 4.

For wide multipliers, the tool always implements final-stage adders to implement the final output in logic. For example, this is how a 69x53 wide multiplier is fractured:

```
Mult1= 51-bit x 51-bit
Mult2= 51-bit x 2-bit
Mult3= 18-bit x 51-bit
Mult4= 18-bit x 2-bit
```

You then get the following:

`Mult1`	1 Mult and 8 MultAdds, using cascade and shift functions
`Mult3`	1 Mult and 2 MultAdds, using cascade and shift functions
`Mult2` `Mult4`	Mapped to logic
Connections	3 adders to connect the four fractured multipliers: `Mult 1, Mult2, Mult3`, and `Mult4`

Note: If a wide multiplier is followed by an adder or subtractor, only the wide multiplier is packed into the RTAX-DSP MATH blocks using cascade and shift functions. The adder or subtractor is mapped to logic.

Log File Message

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

```
@N: : test.v(43) | Multiplier un1_A[51:0] is implemented with multiple MATH18X18
Blocks using  cascade/shift feature.
```

Pipelined Registers with Wide Multipliers

The tool pipelines registers at the inputs and output of wide multipliers in different hierarchies into multiple RTAX-DSP 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 those are not sufficient, the tool considers input registers
- The maximum number of pipeline stages (including input and output registers) that can be accommodated in wide multiplier structure is <number of MATH blocks> + 1.

Note the following:

- 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 RTAX-DSP MATH blocks.
- If the output is unregistered and the inputs are registered with different clocks, the input registers are not pipelined in the RTAX- DSP 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 RTAX-DSP 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 RTAX-DSP MATH blocks, as long as all the registers use the same clock. The adder/subtractor and output register are mapped to logic.
- The tool does not pipeline registers at the inputs and outputs of wide multipliers whose widths exceed 51 bits (unsigned) or 52 bits (signed).

See *Example 13: 35x35-Bit Signed Multiplier with 2 Pipelined Register Stages,* on page 28 for an example.

Wide Multiplier Coding Examples

The following examples show how to code wide multipliers so that they are inferred and mapped to RTAX-DSP MATH blocks, according to the guidelines explained in *Inferring RTAX-DSP MATH Blocks for Wide Multipliers, on page 19*.

- Example 8: Unsigned 20x17-Bit Multiplier (One Wide Input), on page 23
- Example 9: 21x18-Bit Signed Multiplier (One Wide Input), on page 24
- Example 10: Unsigned 26x26-Bit Multiplier (Two Wide Inputs), on page 25
- Example 11: 35x35-Bit Signed Multiplier (Two Wide Inputs), on page 26
- Example 12: 69x53-Bit Signed Multiplier (Inputs Wider than 52 Bits), on page 27
- Example 13: 35x35-Bit Signed Multiplier with 2 Pipelined Register Stages, on page 28

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

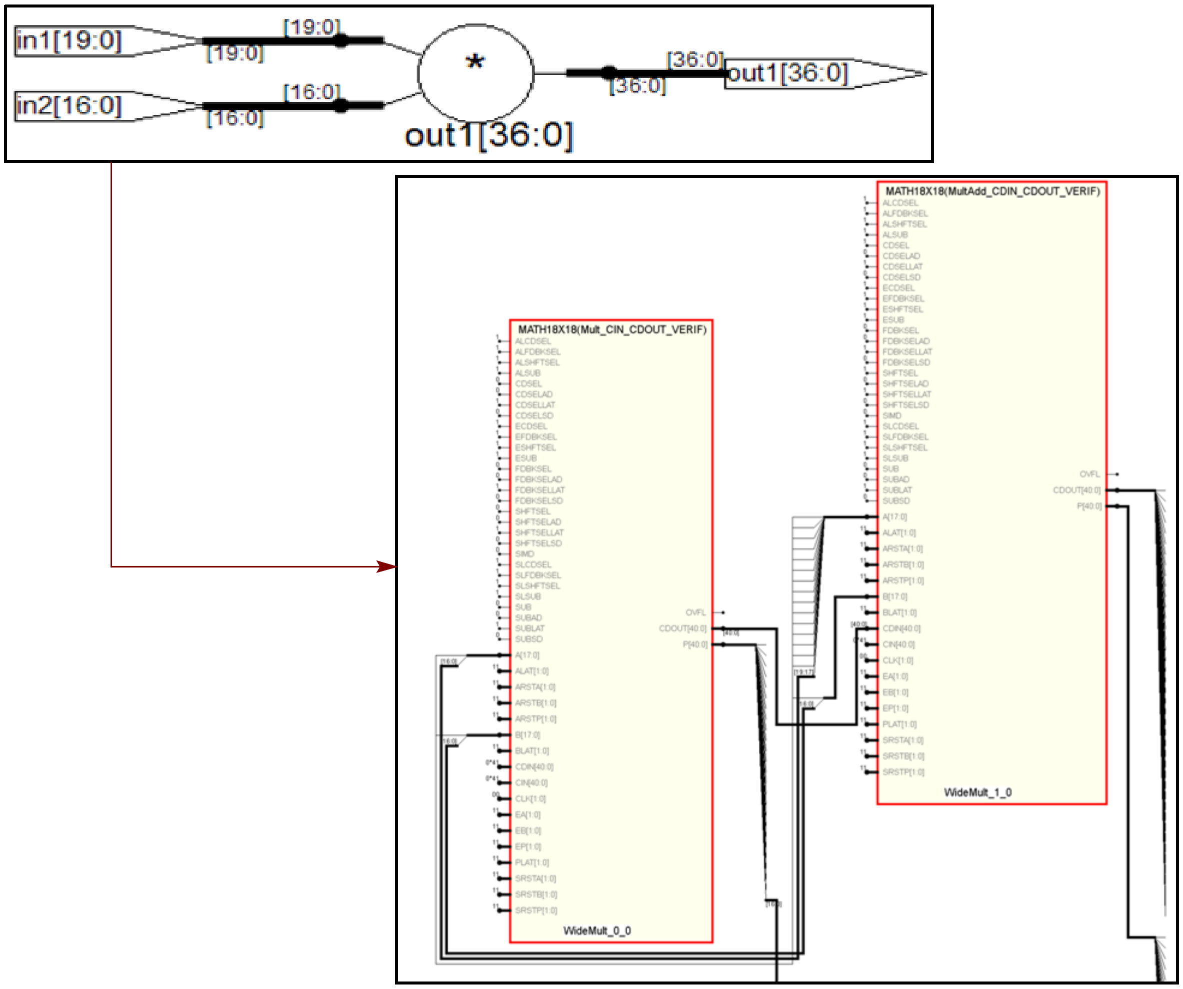
```
library iEEE;
use iEEE.std_logic_1164.all;
use iEEE.std_logic_unsigned.all;

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;
```

After synthesis, the log report shows that the synthesis tool split this multiplier and mapped it to two RTAX-DSP MATH blocks.

Resource Usage Report for Unsigned 20x17-Bit Multiplier

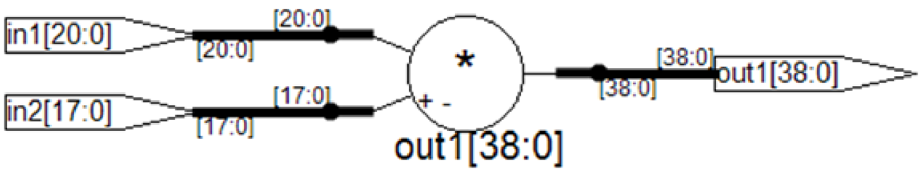
The report shows that the synthesis tool inferred 1 MultAdd and 1 Mult, as described in *Mapping Fractured Multipliers, on page 20*.

```
Target Part: rtax4000d_ccga/lga1272-s
Combinational Cells:        0 of 36960 (0%)
Sequential Cells:           0 of 18480 (0%)
Total Cells:                0 of 55440 (1%)
DSP Blocks:                 2
Clock Buffers:              0
IO Cells:                   0

Details:
   MATH18X18:     1         MultAdds
   MATH18X18:     1         Mults
```



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

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

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

endmodule
```



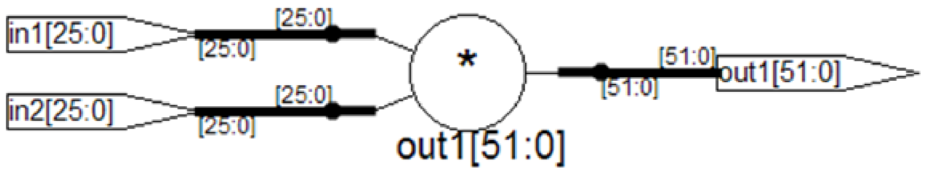
Resource Usage Report for Signed 21x18-Bit Multiplier

In accordance with the fracturing algorithm, synthesis tool reports the inference of 1 Mult and 1 MultAdd:

```
Target Part: rtax4000d_ccga/lga1272-s
Combinational Cells:        1 of 36960 (0%)
Sequential Cells:           0 of 18480 (0%)
Total Cells:                1 of 55440 (1%)
DSP Blocks:                 2
Clock Buffers:              0
IO Cells:                   0

Details:
   buff:           1        comb:1
   MATH18X18:      1        MultAdds
   MATH18X18:      1        Mults
```

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

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

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

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



Resource Usage Report for Unsigned 26x26-Bit Multiplier

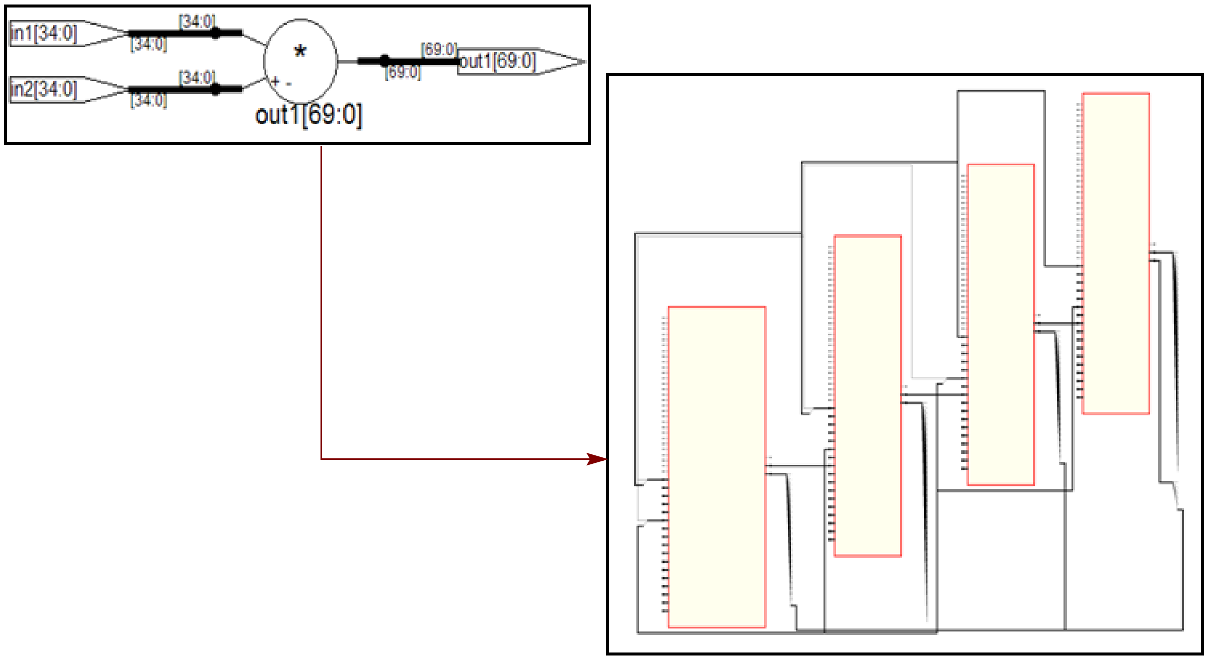
After synthesis, the log report shows that the synthesis tool split this multiplier and mapped it to four RTAX-DSP MATH blocks. It infers 1 Mult and 3 MultAdd blocks.

```
Target Part: rtax2000d_ccga/lga1272-s
Combinational Cells:     0 of 19712 (0%)
Sequential Cells:        0 of 9856 (0%)
Total Cells:             1 of 29568 (1%)
DSP Blocks:              4
```

```
Details:
MATH18X18:        3      MultAdds
MATH18X18:        1      Mults
```

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

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



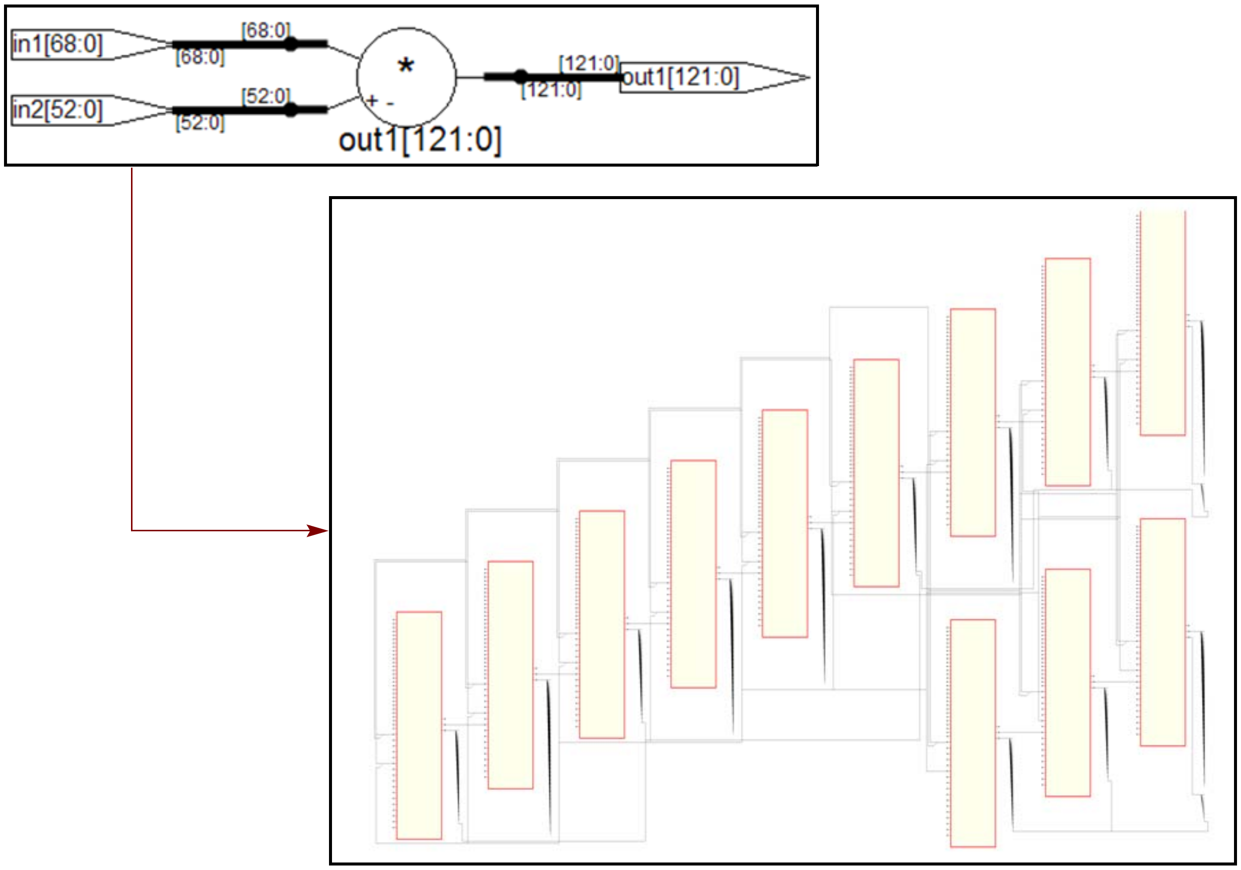
Resource Usage Report for Signed 35x35-Bit Multiplier

The synthesis tool infers 1 Mult and 3 MultAdd blocks.

```
Target Part: rtax2000d_ccga/lga1272-s
Combinational Cells:         0 of 19712 (0%)
Sequential Cells:            0 of 9856 (0%)
Total Cells:                 0 of 29568 (1%)
DSP Blocks:                  4

Details:
MATH18X18:        3         MultAdds
MATH18X18:        1         Mults
```

Example 12: 69x53-Bit Signed Multiplier (Inputs Wider than 52 Bits)

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



Resource Usage Report for Signed 69x53-Bit Multiplier

The synthesis tool fractures the 69x53 multiplier into many multipliers. It infers 1 Mult and 8 MultAdds for the 51x51 multiplier and 1 Mult and 2 MultAdds for the 18x51 multiplier. The 51x2 and 18x2 multipliers are mapped to logic.

```
Target Part: rtax2000d_ccga/lga1272-s
Combinational Cells:        388 of 19712 (0%)
Sequential Cells:           0 of 9856 (0%)
Total Cells:                388 of 29568 (1%)
DSP Blocks:                 12
```

```
Details:
add1:            250      comb:1
and2A            4        comb:1
buff             21       comb:1
cm8              92       comb:1
cm8inv           26
xor2             21       comb:1

MATH18X18:       10       MultAdds
MATH18X18:       2        Mults
```

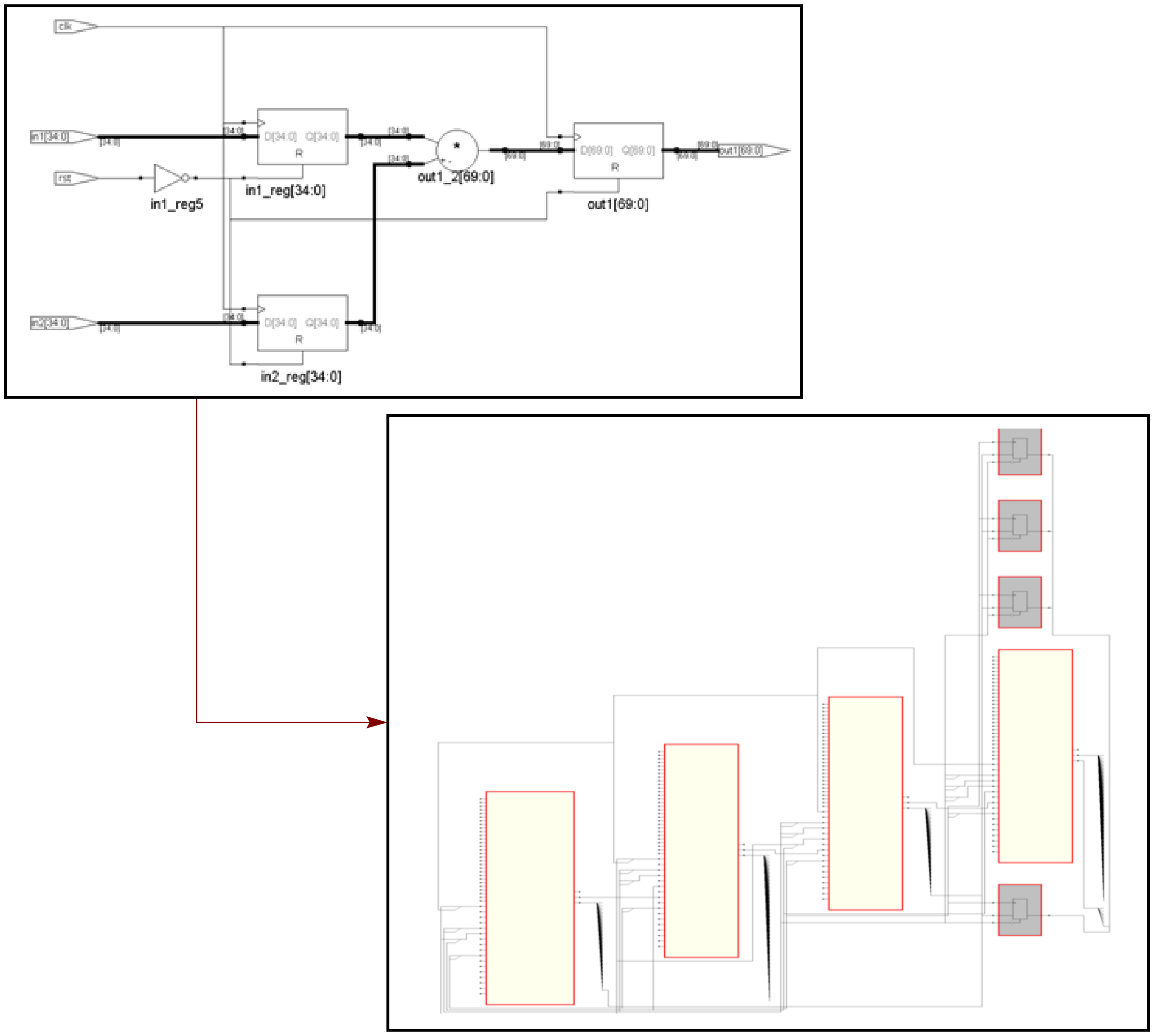
Example 13: 35x35-Bit Signed Multiplier with 2 Pipelined Register Stages

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

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

endmodule
```

The register pipelining algorithm first pipelines registers at the output of the RTAX-DSP 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 RTAX-DSP MATH block as required.

For example, 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 RTAX-DSP MATH blocks and infers sequential primitives at the output of the RTAX-DSP MATH blocks for register balancing.

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



Resource Usage Report for Signed 35x35-Bit Multiplier

The synthesis tool infers 1 Mult and 3 MultAdd blocks.

```
Target Part: rtax2000d_ccga/lga1272-s
Combinational Cells:         0 of 19712 (0%)
Sequential Cells:            34 of 9856 (0%)
Total Cells:                 34 of 29568 (1%)
DSP Blocks:                  4

Details:
dfc1b:             34        Seq:1
MATH18X18:         3         MultAdds
MATH18X18:         1         Mults
```

Current Limitations

For successful RTAX-DSP MATH inference with the Synplify Pro software, it is important that you use a supported coding structure, as there are some limitations to what the synthesis tool infers. See *Coding Style Examples*, on page 4 and *Wide Multiplier Coding Examples*, on page 23 for examples of supported structures. Currently, the tool does not support the following:

- Multiplier-Accumulators (MACs)
- Dynamic Mult-AddSubs
- Overflow extraction
- SIMD (Single Instruction Multiple Data) mode
  In this mode, the RTAX-DSP MATH block is fractured into two 9-bit x 9-bit multipliers.
- Arithmetic right shift for operand C
  When asserted, the tool performs a 17-bit arithmetic right shift on operand C going into the accumulator. In SIMD mode this is ignored.