Using Axcelerator RAM as Multipliers

Introduction

Multiplication is one of the more area-intensive functions in FPGAs. Traditional multiplication techniques use the digital equivalent of longhand multiplication, which we learned in elementary school. These techniques are basically shift and add procedures, which usually result in many levels of logic and limit performance. Pipelining can help to improve the clock performance of the multipliers; in this case, at the cost of more area.

Most humans multiply by individually multiplying digits and referring back to memorized multiplication tables. A similar technique can be employed using the embedded memory on an FPGA. The result of using the RAM as a look-up table multiplier incurs only the delay of the memory access, and has the advantage of not consuming a large quantity of user gates on the FPGA.

This document will address three ways of using RAM blocks as multipliers. The basic single look-up table multiplier, the partial product multiplier, and a RAM-based constant coefficient multiplier.

For the Axcelerator® devices, the single look-up table approach can create a very fast, but narrow, 4-bit multiplier. The partial product multiplier approach uses logic to reduce the amount of memory required, but is slower than a pure look-up table. A partial product multiplier can create a much faster multiplier at the cost of higher area utilization and memory usage. By pipelining the partial product multiplier, the performance can be increased further. The constant coefficient multiplier is the most efficient implementation since it uses a minimum of additional logic gates and still maintains the performance of the basic look-up table multiplier.

Basic Look-Up Table (LUT)-Based Multipliers

A basic LUT-based multiplier is simply a look-up table with the addresses arranged so that part of the address is the multiplicand and the other part is the multiplier. The data width should be set to the sum of the address width to accommodate the product.

Implementing a Basic LUT-Based Multiplier

In the case where a 4-bit value is multiplied by a 4-bit value, you will need a memory block that is 8 bits wide and 256 words deep. The first 4 bits of the address can be configured as the multiplicand and the second 4 bits can be configured as the multiplier. The memory will store the appropriate product values. To multiply the upper 4 bits by the lower four bits, feed both values into the address and clock the memory. The appropriate product value will appear on the RAM output. A diagram of this LUT-based multiplier implementation is shown in Figure 1 on page 2.

© 2005 Actel Corporation

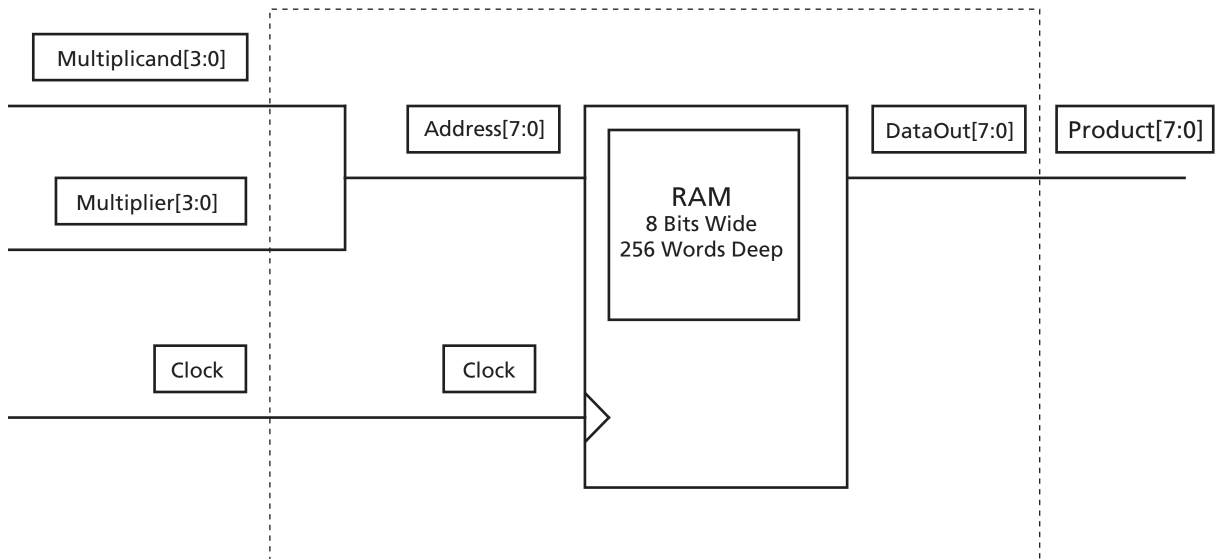

Figure 1 • **Basic Single LUT-Based Multiplier**

Since the memory block in the Axcelerator is synchronous, this configuration will result in a synchronous multiplier. The multiplier's clock frequency is only limited by the data access time of the memory.

This approach is more efficient than implementing multipliers in gates, but it can consume a large amount of memory. The amount of memory required increases with the square of the bit width. The example in Figure 1 requires 256 8-bit words of storage and demonstrates a 4×4 bit multiplier. For an 8×8 bit multiplier, 65,536 16-bit words must be stored using this technique.

Partial Product Multipliers

One way to mitigate the amount of memory required is to use partial product multiplication. This technique combines the look-up table approach with elements of longhand multiplication. For example, to multiply 24 × 43 = 1032 using longhand, we simplify the problem into the sum of 4 multiplication functions and three addition functions (4×3 + ((2×3) ×10)) + ((4×4) + ((2×4) ×10) ×10) = 1032 (Figure 2).

× 24 < A 43 < B 12 60 160 800 1032	× 24 < A 43 < B 12 60 < shifted by 1 decimal place 160 800 1032	× 24 < A 43 < B 12 60 160 < shifted by 1 decimal place 800 1032	× 24 < A 43 < B 12 60 160 800 < shifted by 2 decimal places 1032

Figure 2 • **Partial Product Multiplier Techniques**

Implementing a Partial Product Multiplier

In logic, this same technique can be used to reduce the amount of memory required to perform a multiplication. Using a basic look-up table technique, an 8-bit by 8-bit multiplication would require 128 kbytes of storage. Using partical product multipliers, as shown in Figure 3, the same procedure can be accomplished using 1 kbyte of storage.

In order to accomplish this in logic, using A as the multiplicand and B as the multiplier, take the lower 4 bits of A and multiply it by the lower four bits of B using the look-up table technique. Then take the upper four bits of A and multiply it by the lower four bits of B and shift the partial product result to the left by four. Then add the two results together for the first part of the product.

For the second part of the product, multiply the lower 4 bits of A by the upper four bits of B. Then do the same with the upper 4 bits of both A and B and shift this partial product value to the left by 4. Add the two values of the previous calculation and shift the whole result to the left by four.

Then add the first part of the product to the second part of the product for the final result.

Although this technique is not as fast as implementing the entire multiplication as a single memory element, it does greatly reduce the amount of memory required at the expense of using more core tiles.

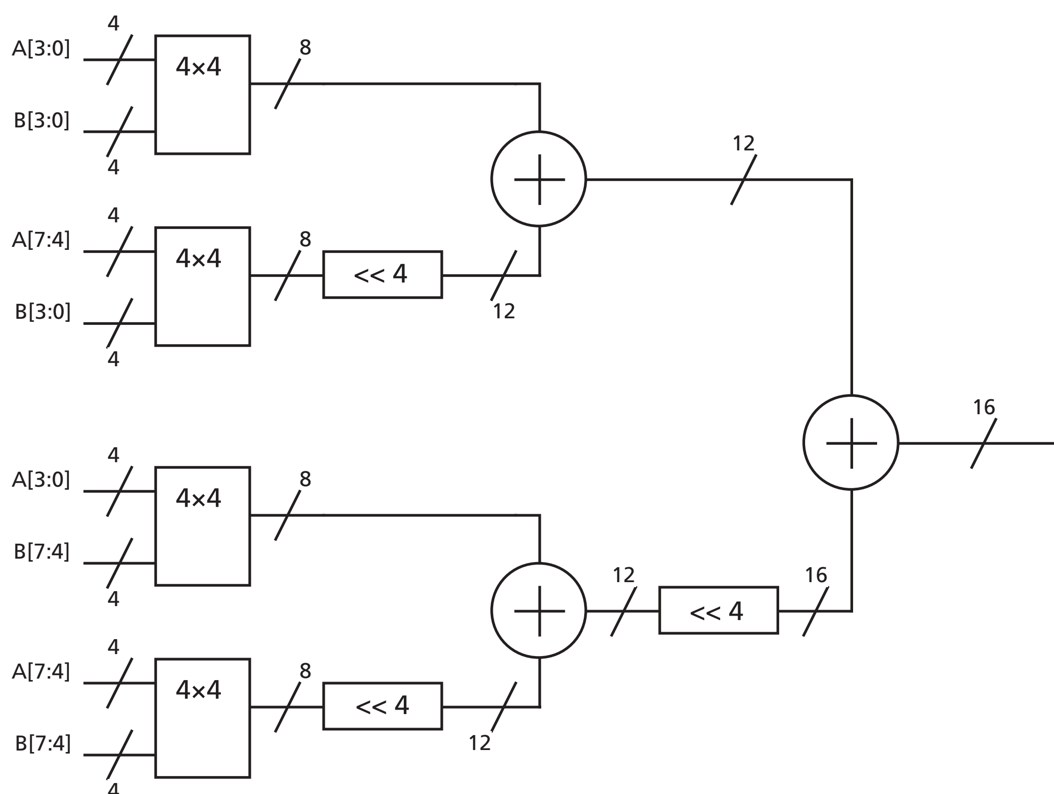

Figure 3 • **Partial Product Multiplier Logic Implementation**

Constant Coefficient Multiplier

A third approach to using memory blocks as multipliers is employing a constant coefficient multiplier. In many cases, especially in Digital Signal Processing (DSP) applications, the multiplicand remains constant and only the multiplier varies. Although ACTgen can create a constant coefficient multiplier using pure logic, implementing this function in RAM creates a much faster multiplier and uses very few logic gates.

Implementing a Constant Coefficient Multiplier

In this approach, only the multiplier must be assigned to the address lines of the memory block. The multiplicand is predetermined and the memory blocks are loaded with the appropriate product values. For example, given that the multiplicand is always 4/h, if the multiplier is B/h, when that value is sent to the address of the memory block, it will return the stored value 2C/h.

This type of multiplier scales linearly with the width of the values being multiplied. While a basic look-up table 8×8 multiplier uses one block of 65536×16 bit words, 128 kbytes of storage, and the partial product look-up table multiplier uses four blocks of 256×8 bit words, 1 kbyte, the constant coefficient multiplier requires only one block of 256×16 bit words, 0.5 kbyte, and does not incur the cost of the additional logic and delay incurred by using the partial product multiplier.

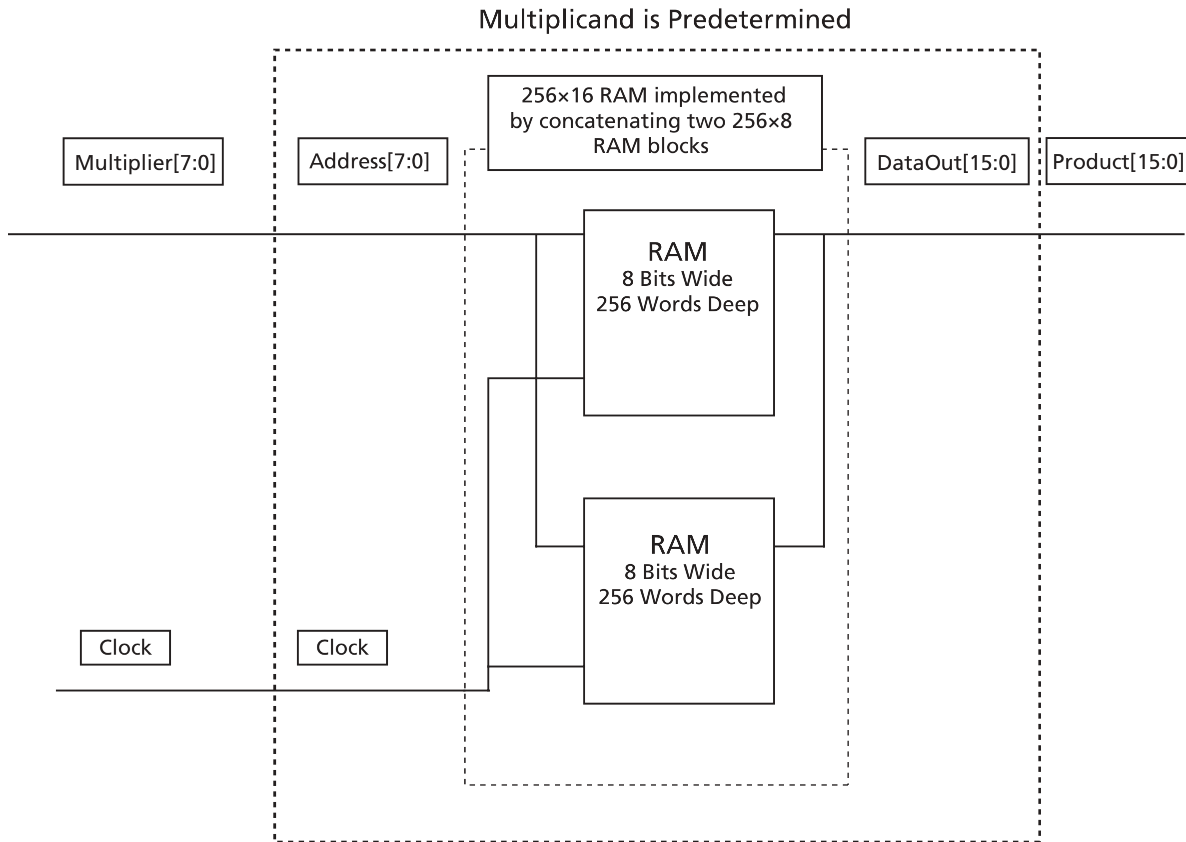

Figure 4 • **Constant Coefficient Multiplier Logic**

Performance and Utilization

Because of architectural variations, the effectiveness of each approach varies between device families. Table 1 on page 5 shows that for a 4×4 multiplier, the RAM-based multiplier is much faster than the equivalent Booth multiplier provided by the ACTgen macro generator. The Booth multiplier is an optimized multiplier that reduces the number of stages required to perform the multiplication function. However, as we expand to an 8×8 multiplier, the amount of memory required to implement the 8×8 multiplier in RAM is too large to be practical. Although the Booth multiplier created in ACTgen produces very good results, the partial product multiplier produces better results with fewer core tiles. However, it does consume four RAM blocks in the process. The performance of the partial product multiplier can further be improved by pipelining the add and shift stages. The constant coefficient multiplier implemented in logic performs very well, but a constant coefficient multiplier implemented in RAM is much faster and consumes less user logic.

Utilization is another consideration for choosing a multiplier. If your design leaves you with unused RAM cells, implementing multipliers with the unused RAM cells can save core tiles. Table 1 on page 5 shows the number of core tiles required to implement each type of multiplier. Not counting the logic required to load the RAM cells, both the 4×4 RAM multiplier and the 8-bit constant coefficient RAM multiplier require only a single RAM cell.

Table 1 • **Performance and Utilization of Multiplier Variations in an AX250**

Multiplier Used	Utilization			Performance (MHz)
	C Cells	R Cells	RAM Blocks	
4×4 bit RAM-based LUT	0	0	1	397
4×4 bit RAM-based LUT pipelined	0	0	1	488
4×4 Booth	32	0	0	179
4×4 Booth fast carry	46	0	0	168
4×4 Booth pipeline	33	12	0	284
8×8 Booth	148	0	0	96
8×8 Booth fast carry	168	0	0	119
8×8 Booth pipelined	151	64	0	207
8-bit partial product	59	27	4	168
8-bit partial product pipelined	43	80	4	320
8-bit logic-based constant coefficient	21	0	0	316
8-bit RAM-based constant coefficient	0	0	1	400

Constant Coefficient RAM Multiplier Example

The constant coefficient RAM multiplier is the most efficient implementation and will be the multiplier used in this example. The RAM block must first be loaded with data in order to produce the correct product values. The Axcelerator RAM makes preloading the memory block very simple. Since the memory in the Axcelerator has two ports, the read port can be dedicated to reading the data for multiplication and the write port can be dedicated to loading data.

The example in Figure 5 on page 6 uses logic within the device as a simple memory loader to preload the RAM for use as an 8-bit constant coefficient multiplier with a 8-bit multiplicand value of 0E/h."Appendix 1" on page 7 includes the design files and the ACTgen generation screens for this example. The memory loader is simply a counter that cycles through the addresses available, with an adder that increments the product values and feeds them into a register file that passes the correct data for each address. Once the loader is finished, the load signal is de-asserted and the RAM block is ready to be used as a multiplier. Since the memory in the Axcelerator is synchronous, the multiplier acts as a synchronous multiplier.

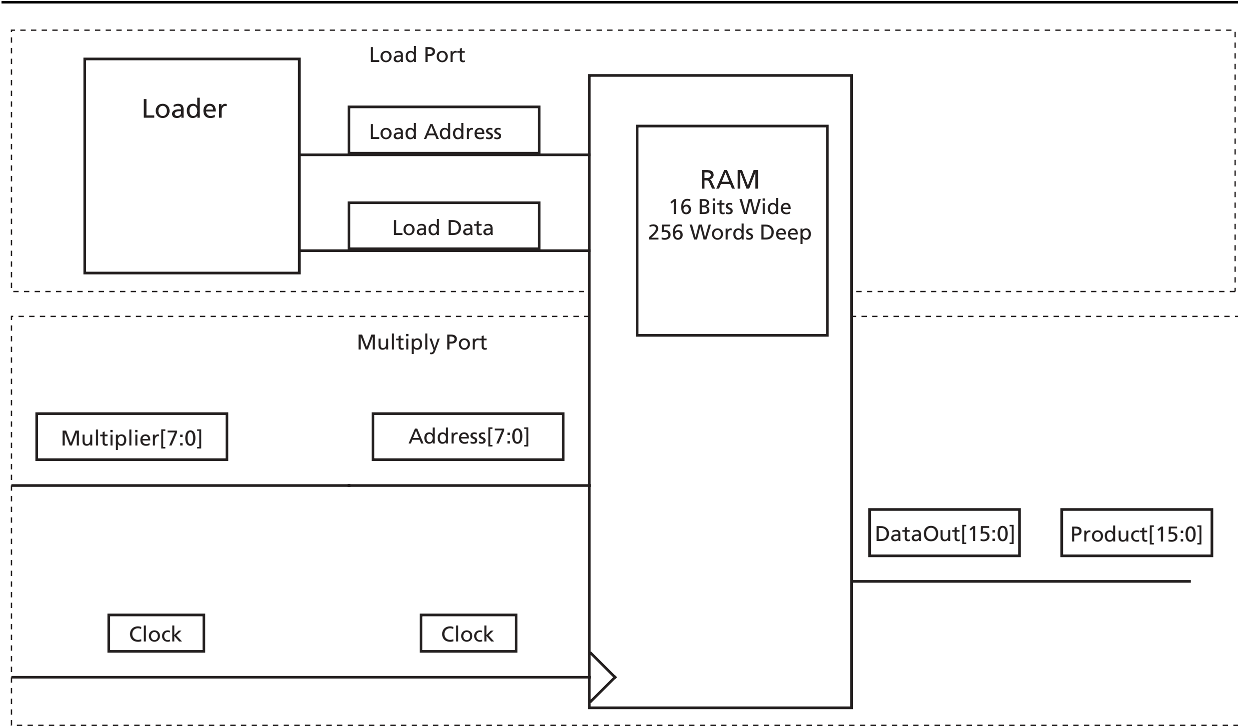

Figure 5 • **Example of a Constant Coefficient Multiplier**

Additional Considerations

Although using RAM blocks as multipliers can save area in many cases, there is overhead required in using this approach. The RAM block must be loaded with the correct values before they can be used as multipliers. An interface for loading and incrementing the RAM block can then load the data on power-up.

A second approach is using an adder to generate values for the RAM block which can be loaded without having the values pre-stored. However, using an adder to generate the values requires additional logic and time to create and store the proper values.

If a microprocessor is available in the system, it can be used to generate the proper values and load them into the RAM blocks. This approach circumvents the additional storage required by the first approach and the logic overhead of the additional multiplier or adder in the second approach.

Conclusion

Using the Axcelerator memory as look-up tables can greatly increase the speed of functions that require multiplication. Several techniques can be used, depending upon the widths and types of the values to be multiplied. For applications where one of the values being multiplied remains constant, often found in DSP functions, the constant coefficient multiplier is the fastest and most efficient look-up table multiplier.

Appendix 1

Design Example: 8-Bit Constant Coefficient Multiplier

The design implemented here is a detailed example of the 8-bit constant coefficient multiplier described in the "Constant Coefficient RAM Multiplier Example" section on page 5. This design includes a loading module that loads the proper product values into the RAM and prepares it for use as a multiplier.

After briefly asserting the active low clear signal, bring clear and load signals high. Allow the clk to cycle for 256 cycles in order to load the memory. When the memory is loaded, bring the load signal low in order to allow the RAM to start functioning as a multiplier.

The mclk, used for multiplying, is independent of the clk signal, the loading clock. This allows the multiplying clock to run at a different rate than the clock used to load the data.

Design Hierarchy

```
Multiply.vhd
     Loader.vhd
          Counter.vhd
          Adder.vhd
          Register16.vhd
     RAM16x8.vhd
```

Multiply

The multiply module combines the loader module, which loads the proper values for multiplying by E/h, with the RAM module, which will act as the actual multiplier.

```
-- multiply.vhd
library IEEE;
use IEEE.std_logic_1164.all;

entity multiply is

   port(load, clr, clk, mclk : in std_logic;
           multiplier: in std_logic_vector (7 downto 0);
           product : out std_logic_vector (15 downto 0));
   end multiply;

   architecture structure of multiply is

   component loader
      port(enable, clr, clk : in std_logic;
              datal : out std_logic_vector (15 downto 0);
              addr : out std_logic_vector (7 downto 0));
   end component;

   component ram16x8
      port( DATA : in std_logic_vector(15 downto 0); PROD : out
              std_logic_vector(15 downto 0); LOAD_ADDR : in
              std_logic_vector(7 downto 0); MULT : in std_logic_vector(
              7 downto 0);LOAD_EN, MULT_EN, LOAD_CLK, MULT_CLK :
      in std_logic) ;
   end component;

   signal address : std_logic_vector (7 downto 0);
   signal dat : std_logic_vector (15 downto 0);
   signal mult_en : std_logic;

   begin

         MULT_EN <= load;
```

```
      load1 : loader
            port map (enable => load, clr => clr, clk => clk, datal => dat,
addr => address);

   ram : ram16x8
         port map (DATA => dat, PROD => product, LOAD_ADDR => address,
MULT => multiplier,

              LOAD_EN => load, MULT_EN => mult_en, LOAD_CLK =>
clk, MULT_CLK => mclk);

end structure;
```

Loader

The loader module accepts a clock, a clear, and an enable signal. It ties together the register, counter, and adder, which performs the actual data loading for the RAM.

```
-- loader.vhd
library IEEE;
use IEEE.std_logic_1164.all;

entity loader is
     port(enable, clr, clk : in std_logic;
             datal : out std_logic_vector (15 downto 0);
             addr : out std_logic_vector (7 downto 0));
end loader;

architecture struct of loader is

component counter
     port(Enable, Aclr, Clock : in std_logic; Q : out
             std_logic_vector(7 downto 0)) ;
end component;

component register16
     port( Data : in std_logic_vector(15 downto 0);Enable, Aclr,
             Clock : in std_logic; Q : out std_logic_vector(15 downto 0
             )) ;
end component;

component adder
     port( DataA : in std_logic_vector(15 downto 0); DataB : in
             std_logic_vector(15 downto 0); Sum : out std_logic_vector(
             15 downto 0)) ;
end component;

constant multiplicand : std_logic_vector := "0000000000001110";
signal data, data2 : std_logic_vector (15 downto 0);

begin

count : counter
port map (Enable => enable, Aclr => clr, Clock => clk, Q => addr);
values : adder

port map (DataA => data2, DataB => multiplicand, sum => data);
reg : register16

port map (Data => data, Enable => enable, Aclr => clr, Clock => clk,
Q => data2);

datal <= data2;

end struct;
```

Register16

The register16 file is generated using ACTgen. The register file is a 16-bit parallel storage register and is used to gate the values from the counter and allows the values to be initially cleared. The register file is generated using the parameters shown in Figure 6:

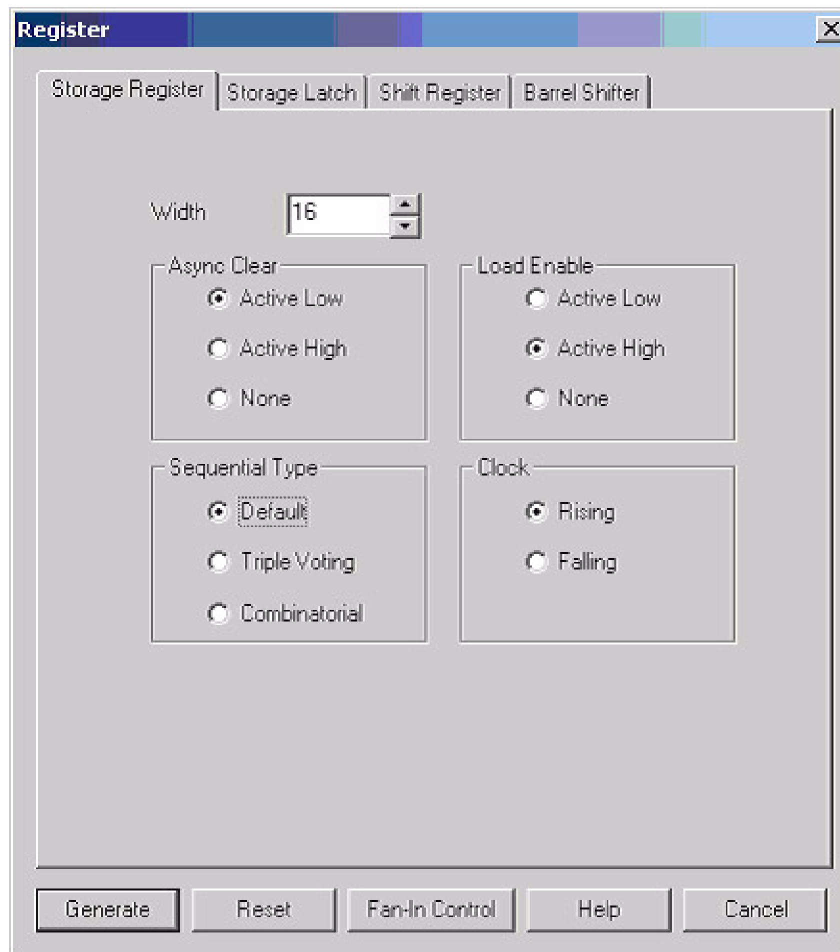

Figure 6 • **Register File Parameters**

Counter

The counter is a 8-bit counter that cycles through all the address values for the RAM. This counter is also generated using ACTgen with the parameters shown in Figure 7.

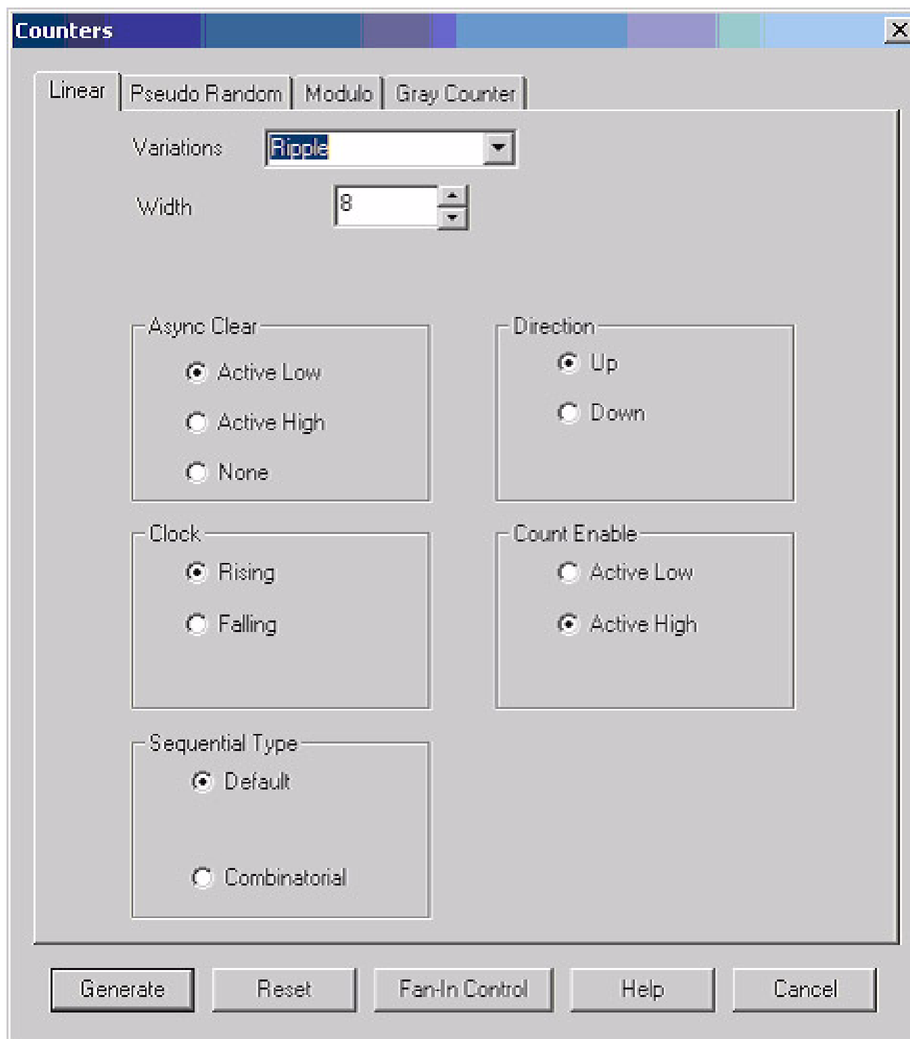

Figure 7 • **Counter Parameters**

Adder

The Adder component is a 16-bit adder used to create the content of the RAM. Since speed is not a major concern for this component, a ripple adder was chosen to minimize utilization (Figure 8).

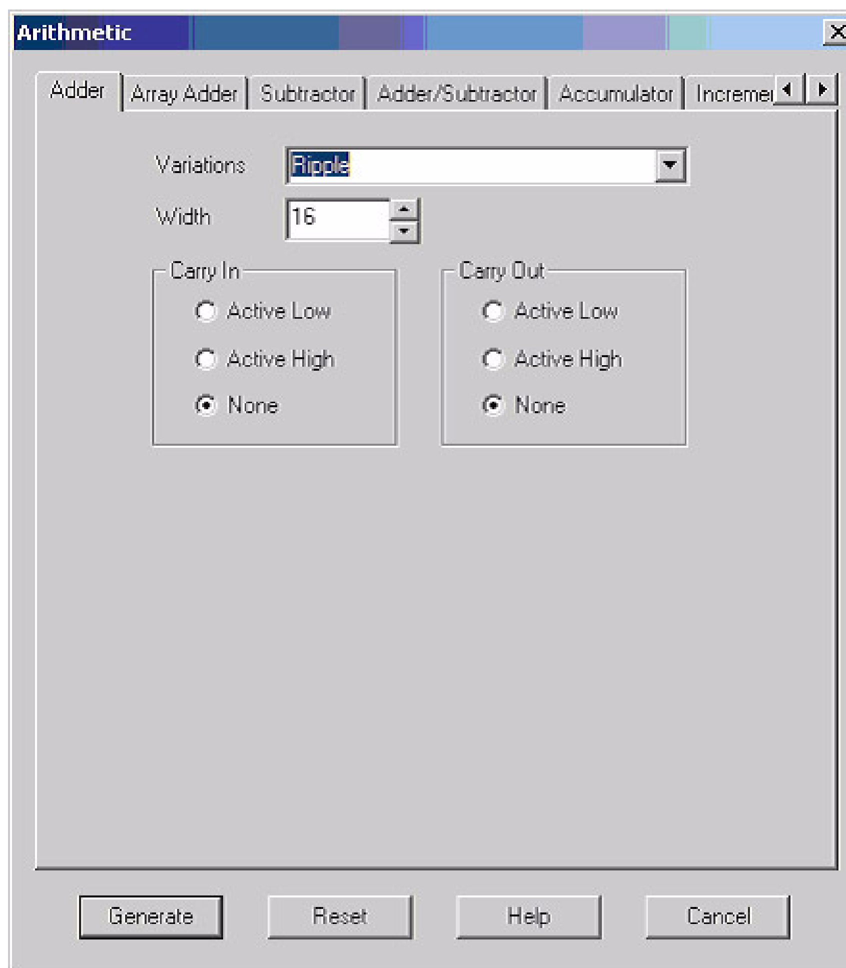

Figure 8 • **Adder Parameters**

RAM16×8

The RAM16×8 is the memory block configuration used as the multiplier in this design. The 16-bit width is required to store the product information, and the 256-bit depth will provide the 8-bit address line used for the multiplier (Figure 9).

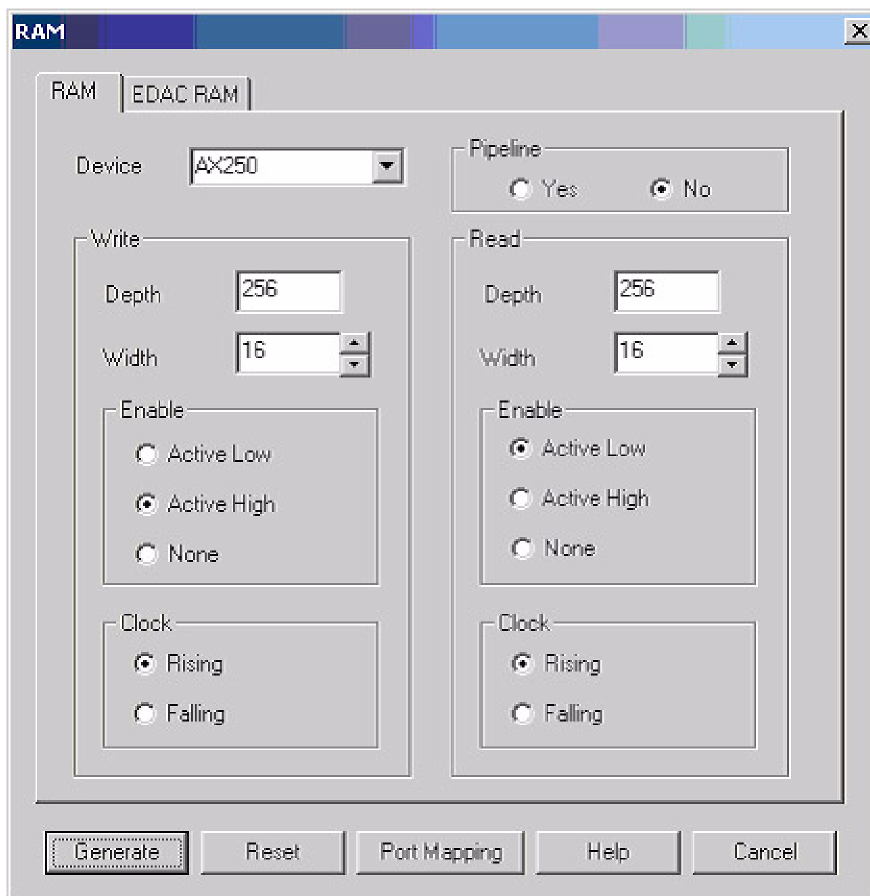

Figure 9 • **RAM Parameters**

Since ACTgen has the ability to rename ports for the Axcelerator RAM, the port mapping shown in Figure 10 is used to make the signal names of the RAM more meaningful as a multiplier.

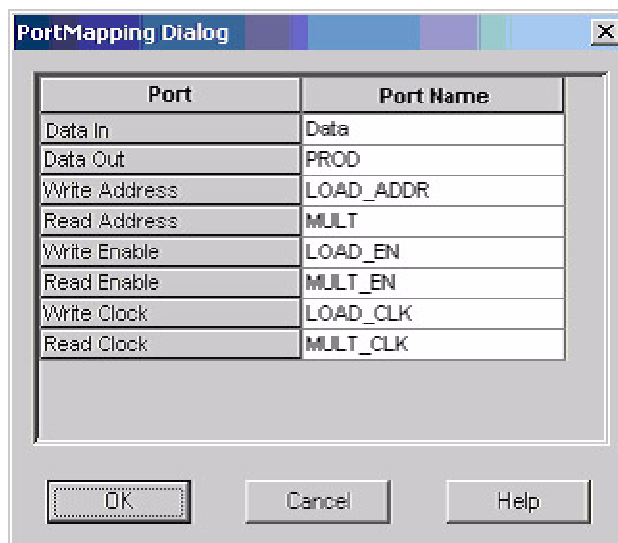

Figure 10 • **PortMapping Dialog**

Actel and the Actel logo are registered trademarks of Actel Corporation.
All other trademarks are the property of their owners.

www.actel.com

Actel Corporation

2061 Stierlin Court
Mountain View, CA
94043-4655 USA

Phone 650.318.4200
Fax 650.318.4600

Actel Europe Ltd.

Dunlop House, Riverside Way
Camberley, Surrey GU15 3YL
United Kingdom

Phone +44 (0) 1276 401 450
Fax +44 (0) 1276 401 490

Actel Japan
www.jp.actel.com

EXOS Ebisu Bldg. 4F
1-24-14 Ebisu Shibuya-ku
Tokyo 150 Japan

Phone +81.03.3445.7671
Fax +81.03.3445.7668

Actel Hong Kong
www.actel.com.cn

Suite 2114, Two Pacific Place
88 Queensway, Admiralty
Hong Kong

Phone +852 2185 6460
Fax +852 2185 6488

51900089-0/1.05