Using ProASIC^PLUS RAM as Multipliers

Table of Contents

Introduction . . . 1
Basic Look-Up Table (LUT)-Based Multipliers . . . 2
Implementing a Basic LUT-Based Multiplier . . . 2
Partial Product Multipliers . . . 3
Implementing a Partial Product Multiplier . . . 4
Constant Coefficient Multiplier . . . 5
Implementing a Constant Coefficient Multiplier . . . 5
Performance and Utilization . . . 6
Constant Coefficient Multiplier Example . . . 7
Additional Considerations . . . 8
Conclusion . . . 8
Related Documents . . . 8
Application Notes . . . 8
Appendix: 8-bit Constant Coefficient Multiplier (Design Example) . . . 9
List of Changes . . . 14

Introduction

Multiplication is one of the more area-intensive functions in field programmable gate array (FPGAs). Traditional multiplication techniques use the digital equivalent of longhand multiplication. These techniques are basically shift and add procedures that 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.

Multiplication by hand is typically reduced to multiplying digits individually and referring back to memorized multiplication tables. A similar technique can be employed using the embedded memory on an FPGA. The performance of using the RAM as a look-up table multiplier is limited only by the delay of the memory access, and has the advantage of not consuming a large quantity of user gates on the FPGA.

This document describes three ways to use RAM blocks as multipliers:

- Basic single look-up table multiplier
- Partial product multiplier
- RAM-based constant coefficient multiplier.

For the ProASIC^PLUS® 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. In fact, the pure logic multiplier implementation for the ProASIC^PLUS, available in the Microsemi ACTgen macro generator, can produce a multiplier that runs at a frequency comparable to the partial product implementation. However, by pipelining the partial product multiplier, the performance can be greatly increased. 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.

© 2015 Microsemi Corporation

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 must 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 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.

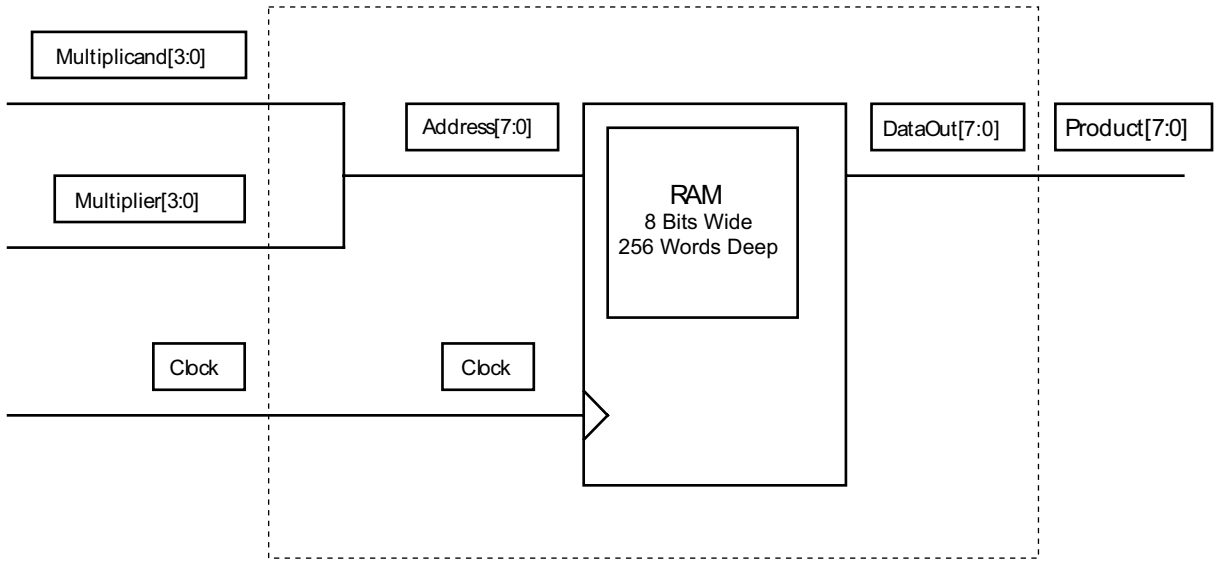

***Figure 1* • Basic Single LUT-Based Multiplier**

Since the memory block in ProASIC^{PLUS} is synchronous, this configuration will result in a synchronous multiplier. The multipliers 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-bit, 536-bit, and 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 long hand multiplication. For example, to multiply 24 × 43 = 1032 using longhand, as shown in Figure 2. Microsemi 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.

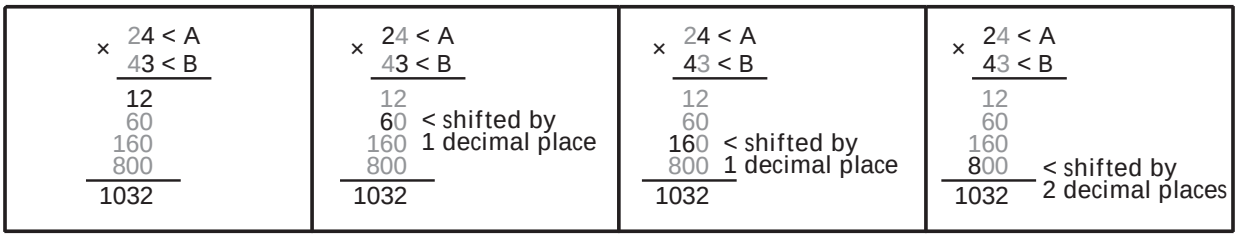

***Figure 2* • Partial Product Multiplier Techniques**

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 8×8 multiplication would require 128 KBytes of storage. Using the partial 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 four bits of A and multiply it by the lower four bits of B using the look-up table technique. Take the upper four bits of A and multiply it by the lower four bit of B and shift the partial product result to the left by four. Add the two results together for the first part of the product.

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

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.

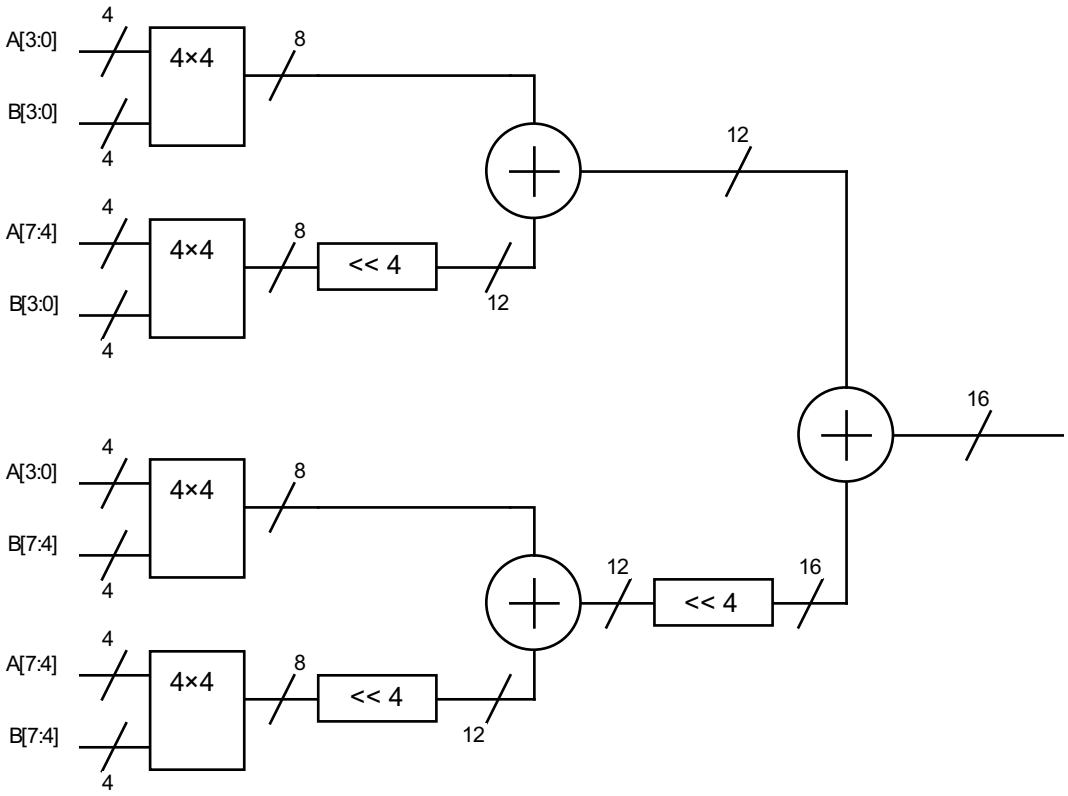

***Figure 3* • Partial Product Multiplier Techniques**

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

Because the ProASIC^{PLUS} memory block is 9 bits wide, the multiplier must be constructed using two RAM blocks, each configured as 256×8 memory blocks. The address lines will be shared between the two memory blocks, so the addresses in both the blocks are accessed simultaneously. The first memory block will hold the first 8 bits of the product and the second memory block will hold the second 8 bits of the product. The two outputs will be concatenated together.

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 2 C/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 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.

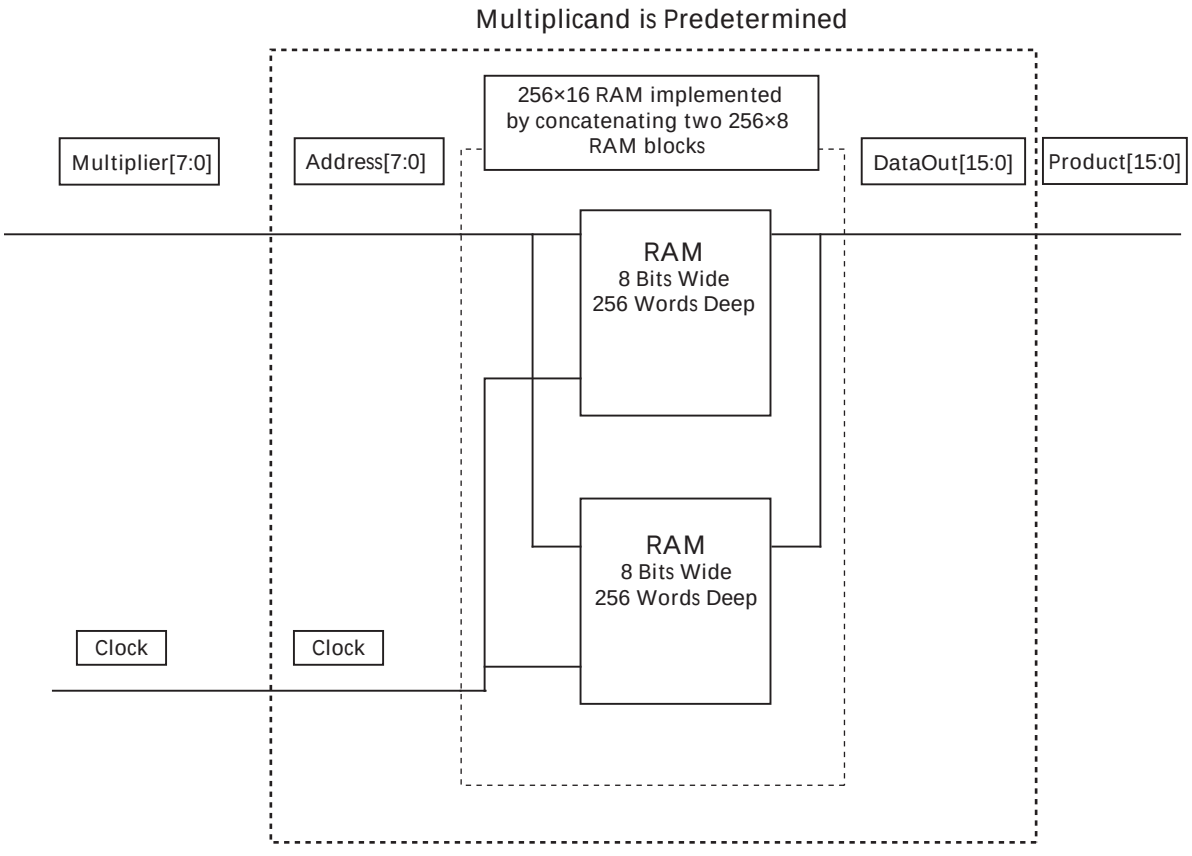

Figure 4 **• Constant Coefficient Multiplier Logic**

Performance and Utilization

Because of architectural variations, the effectiveness of each approach varies between device families. Table 1 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 Microsemi 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, and the Booth multiplier provided by ACTgen performs as well as implementing a partial product RAM multiplier without pipelining. The constant coefficient multiplier implemented in logic performs 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 has unused RAM cells, using the unused RAM as multipliers can save core tiles. Table 1 shows the number of core tiles required to implement each of the multipliers. Not counting the logic required to load the RAM cells, the 4×4 RAM multiplier requires only the RAM cell, and the 8-bit constant coefficient multiplier only requires two cells for cascading.

Table 1 • **Performance and Utilization of Multiplier Variations in an APA300**

Multiplier Used	Performance Mhz	Utilization	
		Core Tiles	RAM Blocks
4×4 synch RAM	160	0	1
4×4 Booth multiplier	70	91	0
8×8 Booth multiplier	47	343	0
8×8 partial product	40	320	4
8×8 partial product pipelined	90	376	4
Const8×8	95	60	0
Const8×8RAM	160	2	2

Constant Coefficient Multiplier Example

The constant coefficient 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 ProASIC^{PLUS} RAM makes preloading the memory block very simple. Since the memory in the ProASIC^{PLUS} 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 data can either be loaded from an external source, such as a microprocessor, using the logic within the device, or through the JTAG port using the UJTAG feature.

The UJTAG feature allows the user to interface with the internal array of the device through the JTAG ports. This allows you to send signals through the JTAG port to your design. One of the uses of this feature is to load data into RAM blocks. Refer to the *AC281: ProASICPLUS RAM/FIFO Blocks App Note* application note for details on how to load a RAM block using the UJTAG.

The example in Figure 5 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 0 E/h.
"Appendix: 8-bit Constant Coefficient Multiplier (Design Example)" on page 9 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 ProASIC^{PLUS} is synchronous, the multiplier acts as a synchronous multiplier.

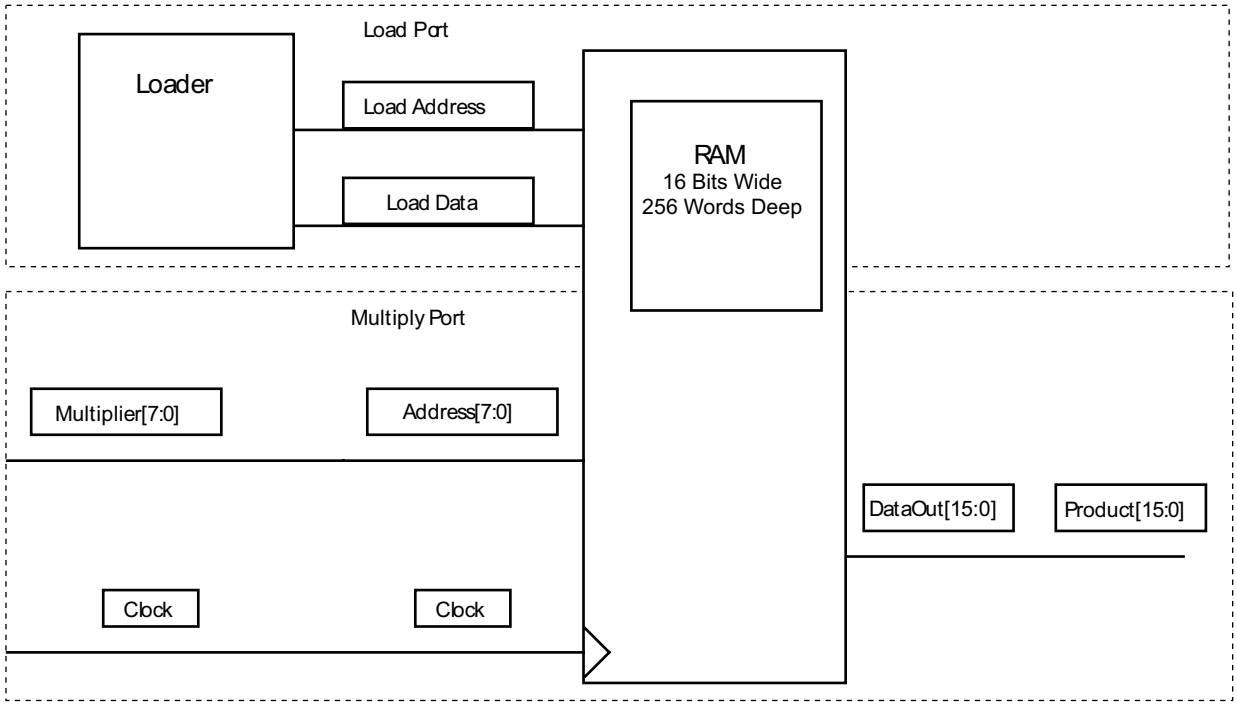

***Figure 5* • Example of a Constant Coefficient Multiplier**

Additional Considerations

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

A second approach is using an adder to generate values to be loaded in the RAM block without having to have the values pre-stored. However, using an adder to generate the values takes additional logic and requires 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 avoids 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 ProASIC^{PLUS} 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. The flexibility of the different techniques allows designers to utilize on-chip resources most effectively. Multiplication resources can be shifted between core tiles and RAMs as appropriate. 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.

Related Documents

Application Notes

AC281: ProASICPLUS RAM/FIFO Blocks App Note

Appendix: 8-bit Constant Coefficient Multiplier (Design Example)

The design implemented here is the example of the 8-bit constant coefficient multiplier described in the "Constant Coefficient Multiplier Example" section on page 7. 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
      Rammult8.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 RAMMULT8
      port(DI : in std_logic_vector(15 downto 0);
            DO : out std_logic_vector(15 downto 0);
            WADDR : in std_logic_vector(7 downto 0);
            RADDR : in std_logic_vector(7 downto 0);
            WRB : in std_logic;
            RDB : in std_logic;
            WCLOCK : in std_logic;
            RCLOCK : in std_logic;
            PO : out std_logic_vector (1 downto 0);
            PI : in std_logic_vector (1 downto 0);
            WPE : out std_logic;
            RPE : out std_logic);
   end component;
   component GND
         port(Y : out std_logic);
   end component;
   signal address : std_logic_vector (7 downto 0);
   signal dat : std_logic_vector (15 downto 0);
   signal mult_en, gndnet : std_logic;
   begin
   MULT_EN <= load;
   GND1 : GND
         port map(Y => gndnet);
   load1 : loader
         port map (enable => load, clr => clr, clk => clk, datal => dat,
addr => address);
   raminst : RAMMULT8
         port map (DI => dat,
                  DO => product,
                  WADDR => address,
                  RADDR => multiplier,
                  WRB => not(load),
```

```
                  RDB => mult_en,
                  WCLOCK => clk,
                  RCLOCK => mclk,
                  PO(0) => OPEN,
                  PO(1) => OPEN,
                  PI(0) => gndnet,
                  PI(1) => gndnet,
                  WPE => OPEN,
                  RPE => OPEN);
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, as shown in Figure 6.

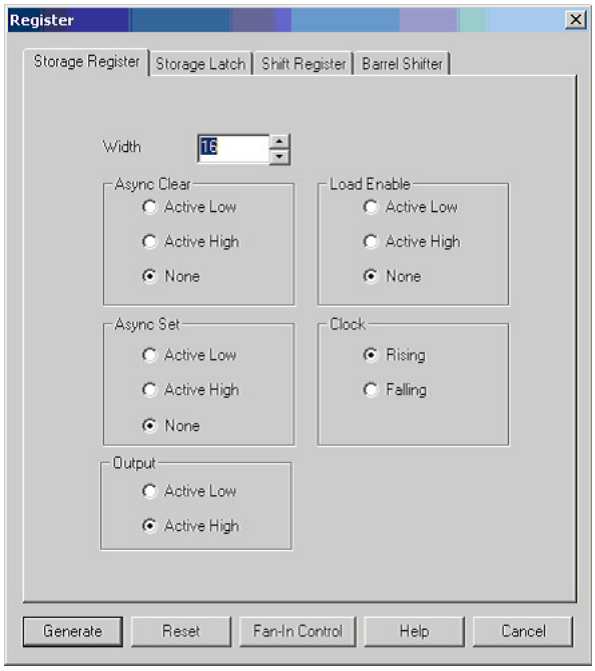

***Figure 6* • Register File Parameters**

Counter

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

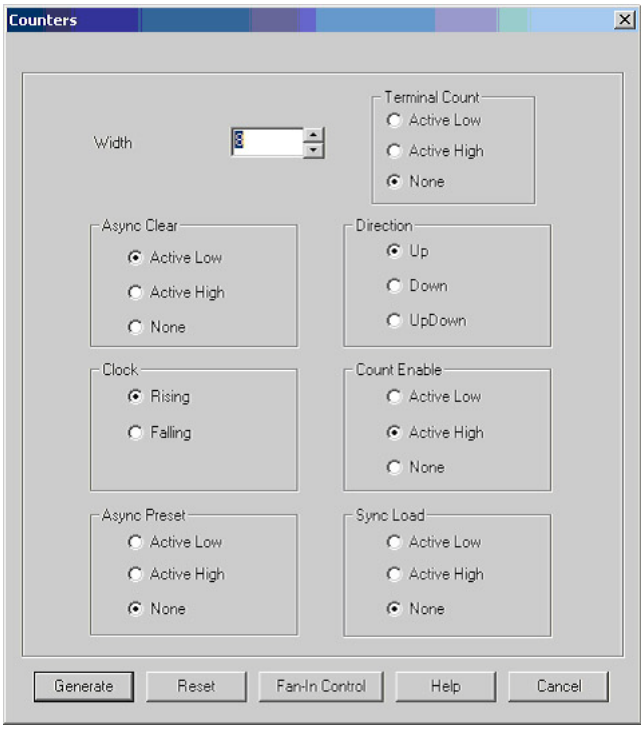

***Figure 7* • Counter Parameters**

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, as shown in Figure 8.

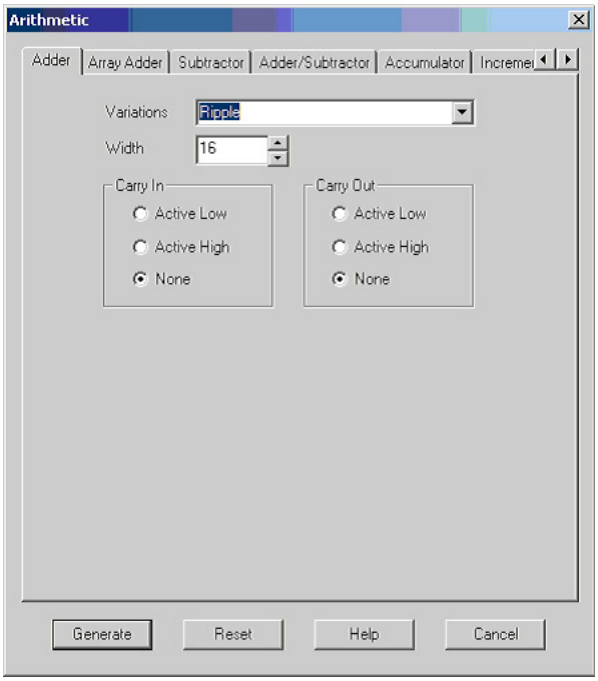

***Figure 8* • Adder Parameters**

Rammult8

The rammult8 is the memory block configuration used as the multiplier in this design. This component is implemented using two RAM cells cascaded together to form a 256 word deep by 16 bits wide RAM cell, as shown in Figure 9.

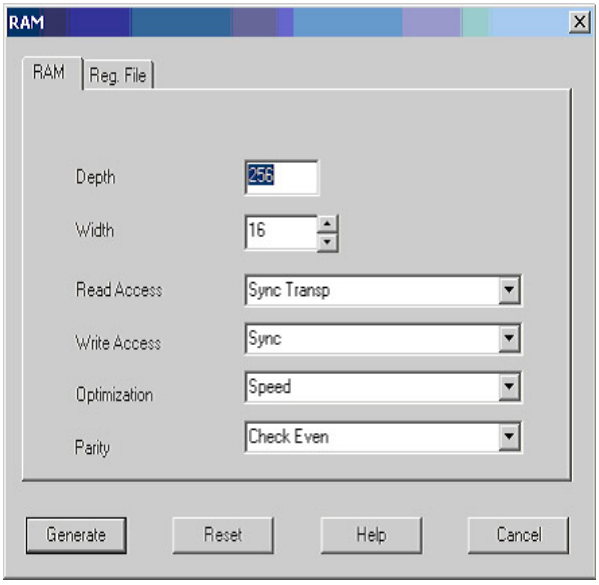

***Figure 9* • RAM Parameters**

List of Changes

The following table shows important changes made in this document for each revision.

Revision	Changes	Pages
Revision 1 (December 2015)	Non-technical updates.	NA
Revision 0 (January 2005)	Initial release.	NA

**The part number is located on the last page of the document.*

Microsemi Corporate Headquarters
One Enterprise, Aliso Viejo,
CA 92656 USA

Within the USA: +1 (800) 713-4113
Outside the USA: +1 (949) 380-6100
Sales: +1 (949) 380-6136
Fax: +1 (949) 215-4996

E-mail: sales.support@microsemi.com

© 2015 Microsemi Corporation. All rights reserved. Microsemi and the Microsemi logo are trademarks of Microsemi Corporation. All other trademarks and service marks are the property of their respective owners.

Microsemi Corporation (Nasdaq: MSCC) offers a comprehensive portfolio of semiconductor and system solutions for communications, defense & security, aerospace and industrial markets. Products include high-performance and radiation-hardened analog mixed-signal integrated circuits, FPGAs, SoCs and ASICs; power management products; timing and synchronization devices and precise time solutions, setting the world's standard for time; voice processing devices; RF solutions; discrete components; security technologies and scalable anti-tamper products; Ethernet Solutions; Power-over-Ethernet ICs and midspans; as well as custom design capabilities and services. Microsemi is headquartered in Aliso Viejo, Calif., and has approximately 3,600 employees globally. Learn more at **www.microsemi.com**.

Microsemi makes no warranty, representation, or guarantee regarding the information contained herein or the suitability of its products and services for any particular purpose, nor does Microsemi assume any liability whatsoever arising out of the application or use of any product or circuit. The products sold hereunder and any other products sold by Microsemi have been subject to limited testing and should not be used in conjunction with mission-critical equipment or applications. Any performance specifications are believed to be reliable but are not verified, and Buyer must conduct and complete all performance and other testing of the products, alone and together with, or installed in, any end-products. Buyer shall not rely on any data and performance specifications or parameters provided by Microsemi. It is the Buyer's responsibility to independently determine suitability of any products and to test and verify the same. The information provided by Microsemi hereunder is provided "as is, where is" and with all faults, and the entire risk associated with such information is entirely with the Buyer. Microsemi does not grant, explicitly or implicitly, to any party any patent rights, licenses, or any other IP rights, whether with regard to such information itself or anything described by such information. Information provided in this document is proprietary to Microsemi, and Microsemi reserves the right to make any changes to the information in this document or to any products and services at any time without notice.

51900088-1/12.15