Using Fusion, IGLOO, and ProASIC3 RAM as Multipliers

Table of Contents

Introduction 1
Basic Lookup Table (LUT) Based Multipliers 1
Partial Product Multipliers 2
Constant Coefficient Multiplier 3
Performance and Utilization 4
Constant Coefficient Multiplier Example 5
Conclusion 6
Related Documents 6

Introduction

Multiplication is one of the more area intensive functions in FPGAs. Traditional multiplication techniques use the digital equivalent of longhand multiplication. 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 people 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 lookup table multiplier incurs only the delay of the memory access and has the advantage of not consuming a large number of user gates on the FPGA.

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

- Basic single lookup table multiplier
- Partial product multiplier
- RAM-based constant coefficient multiplier

For the Fusion, IGLOO®, and ProASIC®3 families of devices, the single lookup table approach can create a very fast but narrow, four-bit multiplier. The partial product multiplier approach uses logic to reduce the amount of memory required, but is slower than a pure lookup table. In fact, the pure logic multiplier implementation for these devices available in the Microsemi® SmartGen core generator can produce a multiplier that runs at a frequency comparable to the partial product implementation, though the pure logic approach uses more core tiles. 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 lookup table multiplier.

Basic Lookup Table (LUT) Based Multipliers

A basic LUT-based multiplier is simply a lookup 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.

© 2015 Microsemi Corporation

Implementing a Basic LUT-Based Multiplier

In the case where a four-bit value is multiplied by a four-bit value, you will need a memory block that is eight bits wide and 256 words deep. The first four bits of the address can be configured as the multiplicand and the second four bits can be configured as the multiplier. The memory will store the appropriate product values. To multiply the upper four 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.

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

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

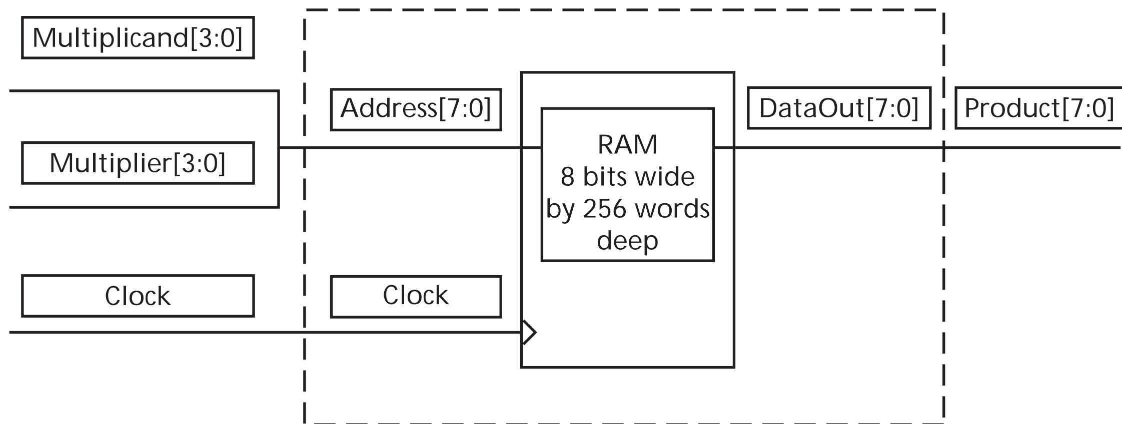

Figure 1 **• Basic Single LUT-Based Multiplier**

Partial Product Multipliers

One way to mitigate the amount of memory required is to use partial product multiplication. This technique combines the lookup table approach with elements of longhand multiplication. For example, to multiply 24 x 43 = 1,032 using longhand, simplify the problem into the sum of four multiply functions and three add functions (Figure 2).

$$(4 \times 3 + ((2 \times 3) \times 10)) + (((4 \times 4) + ((2 \times 4) \times 10)) \times 10) = 1{,}032$$

EQ 1

24 < A	24 < A	24 < A	24 < A
X43 < B	X43 < B	X43 < B	X43 < B
12	12	12	12
60	60 < Shifted by 1	60 < Shifted by 1	60 < Shifted by 2
160	160 decimal place	160 decimal place	160 decimal places
800	800	800	800
1,032	1,032	1,032	1,032

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 multiply function. Using a basic lookup table technique, an eight-bit by eight-bit multiply would require 128 kb of storage. As shown in Figure 3, using partial product multipliers, the same procedure can be accomplished using 1 kb 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 lookup 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 four bits of A by the upper four bits of B. Then do the same with the upper four bits of both A and B and shift this partial product value to the left by four. 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.

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

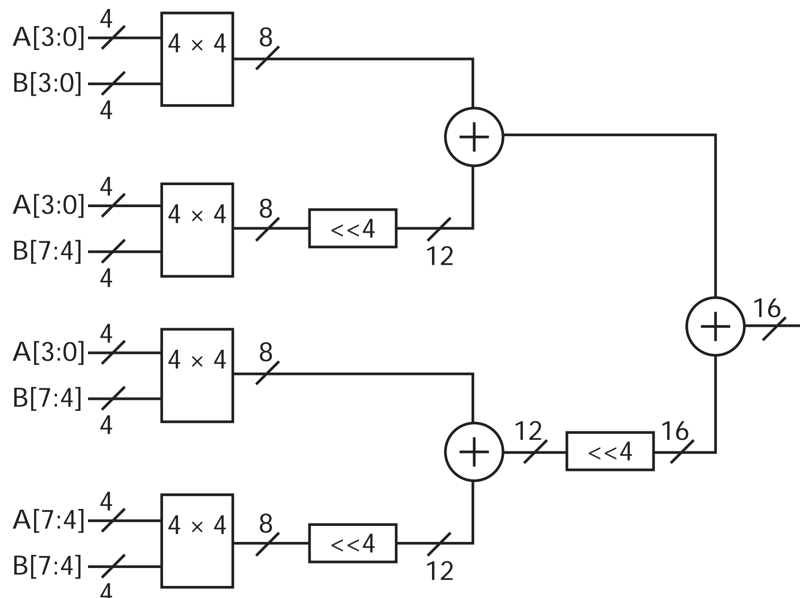

***Figure 3* • Partial Product Multiplier Logic Implementation**

Constant Coefficient Multiplier

A third approach to using memory blocks as multipliers is employing a constant coefficient multiplier. In many cases, especially in DSP applications, the multiplicand remains constant and only the multiplier varies.

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 block is loaded with the appropriate product values (Figure 4 on page 4). 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 lookup table 8 x 8 multiplier uses one block of 65,536 x 16 bit words (128 kb) of storage, and the partial product lookup table multiplier uses four blocks of 256 x 8 bit words (1 kb) the constant coefficient multiplier

requires one block of 256 x 16 bit words (0.5 kb) and does not incur the cost of the additional logic and delay incurred by using the partial product multiplier.

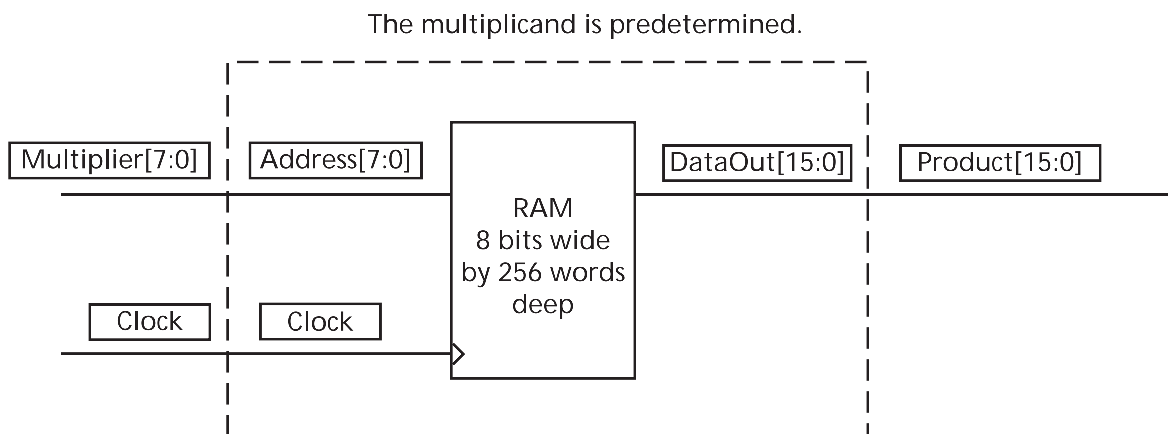

***Figure 4* • Constant Coefficient Multiplier Logic**

Performance and Utilization

Because of architectural variations, the effectiveness of each approach varies between device families. Table 1 shows, for a 4 x 4 multiplier, the RAM-based multiplier is much faster than the equivalent Booth multiplier provided by the SmartGen core 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 x 8 multiplier, the amount of memory required to implement the 8 x 8 multiplier in RAM is too large to be practical, and the Booth multiplier provided by SmartGen performs as well as implementing a partial product RAM multiplier. Also, as shown in Table 1, pipelining either the Booth multiplier or the partial product multiplier increases the performance of both, and both implementations run at similar speeds. However, a constant coefficient multiplier is clearly much faster than either implementation.

Utilization is another consideration for choosing a multiplier. If your design leaves you with unused RAM cells, employing 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 x 4 RAM multiplier requires only the RAM cell, and the eight-bit constant coefficient multiplier only requires two cells. The partial product multiplier uses a third fewer tiles to implement than does the Booth multiplier.

***Table 1* • Performance and Utilization of Multiplier Variations**

Multiplier Used	Performance MHz	Utilization	
		Core Tiles	RAM Blocks
4 x 4 RAM multiplier	293	0	1
4 x 4 Booth multiplier	98	79	0
4 x 4 pipelined Booth multiplier	158	92	0
8 x 8 Booth multiplier	68	305	0
8 x 8 Booth multiplier with 1 pipeline stage	102	344	0
8 x 8 Booth multiplier with 2 pipeline stage	123	386	0
8 x 8 Booth multiplier with 3 pipeline stage	120	431	0
8 x 8 partial product multiplier	63	196	4

Table 1 • **Performance and Utilization of Multiplier Variations (continued)**

Multiplier Used	Performance MHz	Utilization	
		Core Tiles	RAM Blocks
8 x 8 partial product multiplier with pipelining	129	311	4
8 x 8 constant coefficient multiplier	281	2	1

Note: Timing numbers are based on worst-case, commercial numbers for an AFS600 in a–2 speed grade.

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 ProASIC3 RAM makes preloading the memory block very simple. Refer to the *ProASIC3 Flash Family FPGAs*, *ProASIC3E Flash Family FPGAs* and *ProASIC3L Flash Family FPGAs* datasheets for more information. Since the memory in the IGLOO, Fusion, and ProASIC3 device has two ports, one port can be dedicated to reading the data for multiply and the other can be dedicated to loading data. The data can either be loaded from embedded Flash memory or 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 you 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 "SRAM and FIFO Memories in Microsemi's Low Power Flash Devices" chapter of the *IGLOO* and *ProASIC3 FPGA fabric user's guides* 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 a four-bit constant coefficient multiplier with a four-bit multiplicand value of E/h. "Appendix 1" on page 8 includes the design files and the SmartGen 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 deasserted, and the RAM block is ready to be used as a multiplier. Since the memory in the IGLOO, Fusion and ProASIC3 devices is synchronous, the multiplier acts as a synchronous multiplier.

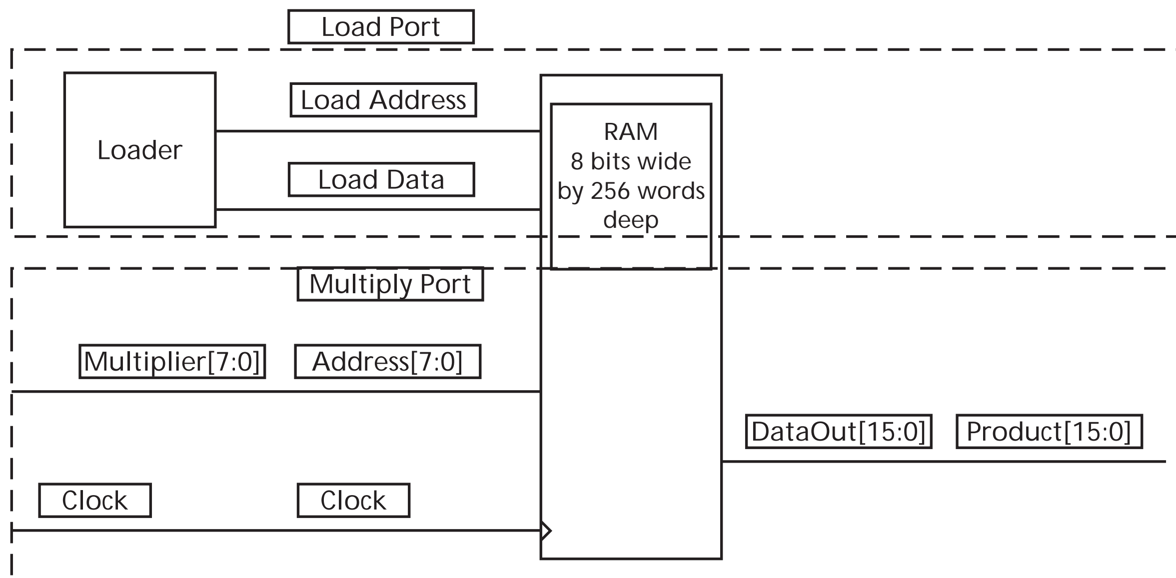

Figure 5 • **Constant Coefficient Multiplier Logic**

Additional Considerations

While in many cases using RAM blocks as multipliers can save area, 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 to load and increment the RAM block can then load the data on power-up.

A second approach is using a multiplier or adder to generate values in the RAM block to be loaded without having to have the values prestored. However, using either a multiplier or an adder to generate the values takes additional logic and does require time to create and store the proper values.

If a microprocessor is available in the system, it can also 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 Fusion, IGLOO, and ProASIC3 device memory as lookup 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 the most efficient lookup table multiplier.

Related Documents

Datasheets

IGLOO Low Power Flash FPGAs

http://www.microsemi.com/documents/IGLOO_DS.pdf

IGLOOe Low Power Flash FPGAs

http://www.microsemi.com/documents/IGLOOe_DS.pdf

IGLOO PLUS Low Power Flash FPGAs

http://www.microsemi.com/documents/IGLOOPLUS_DS.pdf

Fusion Family of Mixed-Signal FPGAs

http://www.microsemi.com/documents/Fusion_DS.pdf

ProASIC3 Flash Family of FPGAs

http://www.microsemi.com/documents/PA3_DS.pdf

ProASIC3E Flash Family of FPGAs

http://www.microsemi.com/documents/PA3E_DS.pdf

ProASIC3L LowPower Flash FPGAs

http://www.microsemi.com/documents/PA3L_DS.pdf

User Guides

IGLOO FPGA Fabric User’s Guide

http://www.microsemi.com/documents/IGLOO_UG.pdf

ProASIC3 FPGA Fabric User’s Guide

http://www.microsemi.com/documents/PA3_UG.pdf

Fusion FPGA Fabric User’s Guide

http://www.microsemi.com/documents/Fusion_UG.pdf

Appendix 1

Design Example: 8-Bit Constant Coefficient Multiplier

The design implemented here is the example for the eight-bit constant coefficient multiplier described above. 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
      Reg16.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, RESET :
            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, RESET
=> clr);

end structure;
```

Loader

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

```
-- loader
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 reg16
        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 : reg16
    port map (Data => data, Enable => enable, Aclr => clr, Clock => clk,
                Q => data2);

    datal <= data2;

    end struct;
```

Reg16

The reg16 register file is generated using SmartGen. The register file is a 16-bit parallel storage register and is used to gate the values from the counter. It allows the values to be initially cleared. The register file is generated using the parameters shown in Table 2.

Table 2 • **Reg16 Parameters**

Parameter	IGLOO	Fusion	ProASIC3
Width		16	16
Async Clear		Active low	Active low
Async Set		Active high	None
Output		Active high	Active high
Load Enable		Active high	Active high
Clock		Rising	Rising

Adder

The adder component is a 16-bit adder that continually increments the values loaded into the RAM by a value of E/h, see Table 3.

Table 3 • **Adder Parameters**

Parameter	IGLOO	Fusion	ProASIC3
Variations		Ripple	Ripple
Width		16	16
Carry In		None	None
Carry Out		None	None

Counter

The counter is an eight-bit counter that cycles through all the address values for the RAM. This counter is also generated using SmartGen with the parameters shown in Table 4.

Table 4 • **Counter Parameters**

Parameter	IGLOO	Fusion	ProASIC3
Width		8	8
Async Clear		Active low	Active low
Clock		Rising	Rising
Async Preset		None	None
Terminal Count		None	None
Direction		Up	Up
Count Enable		Active high	Active high
Sync Load		None	None

RAM16x8

The RAM16x8 is the memory block configuration used as the multiplier in this design. The memory block is 256 words deep with a pair of eight-bit addresses and 16-bit data buses (-Table 5).

Table 5 • **Counter Parameters**

Parameter	IGLOO	Fusion	ProASIC3
Write Depth		256	256
Write Width		16	16
Read Depth		256	256
Read Width		16	16
Clocks		Independent read and write clocks	Independent read and write clocks
Write Clock		Rising	Rising
Read Clock		Rising	Rising
RAM Type		Two port	Two port
Reset		Active low	Active low
Write Enable		Active high	Active high
Read Enable		Active low	Active low
Write Mode A		N/A	Retain output data
Read Pipeline A		–	No
Write Mode B		N/A	–
Read Pipeline B		–	–
Initialize RAM		–	N/A
Customize RAM Content		–	N/A

Figure 6 shows the port map is used in order to make the signals more meaningful as a multiplier.

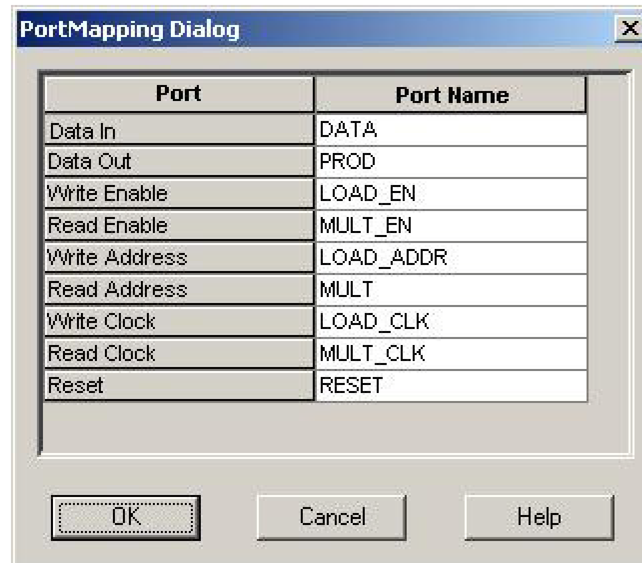

***Figure 6* • Port Mapping Dialog**

List of Changes

Date	Changes	Page
Revision 2 (July 2015)	Non-technical Updates.	NA
Revision 1 (November 2008)	Initial Release.	NA

*Note: *The revision number is located in the part number after the hyphen. The part number is displayed at the bottom of the last page of the document. The digits following the slash indicate the month and year of publication.*

Microsemi Corporation (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 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 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.

51900074-2/07.15