Using ProASIC3/E RAM as Multipliers

Introduction

Multiplication is one of the more area intensive functions in FPGAs. Traditional multiplication techniques use the digital equivalent of long-hand 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 look-up 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 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 ProASIC3/E devices, the single look-up 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 look-up table. In fact, the pure logic multiplier implementation for the ProASIC3/E available in the Actel ACTgen 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 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 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 on page 2.

Since the memory block in the ProASIC3/E 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 4x4 bit multiplier with 256 eight-bit words of storage required. For an 8x8 bit multiplier, 65,536 16-bit words must be stored using this technique.

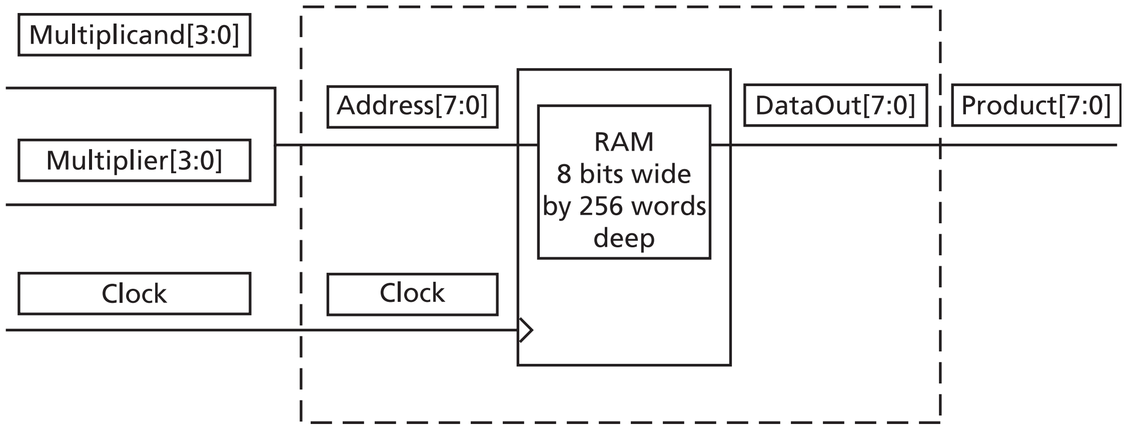


Figure 1 • **Basic Single LUT-Based Multiplier**

Partial Product Multipliers

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

(4X3 + ((2X3) X 10)) + (((4X4) + ((2X4) X 10)) X 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 look-up table technique, an eight-bit by eight-bit multiply would require 128 kb of storage. As shown in Figure 3 on page 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 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 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.

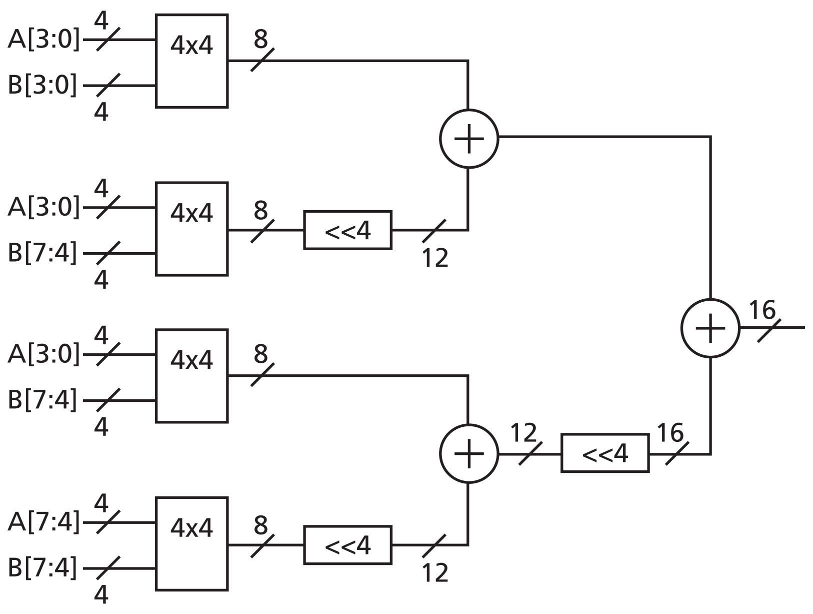


Figure 3 • **Partial Product Multiplier Logic Implementation**

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 look-up table 8x8 multiplier uses one block of 65536x16 bit words, 128 kb, of storage, and the partial product look-up table multiplier uses four blocks of 256x8 bit words, 1 kb, the constant coefficient multiplier requires one block of 256x16 bit words, 0.5 kb, and does not incur the cost of the additional logic and delay incurred by using the partial product multiplier.

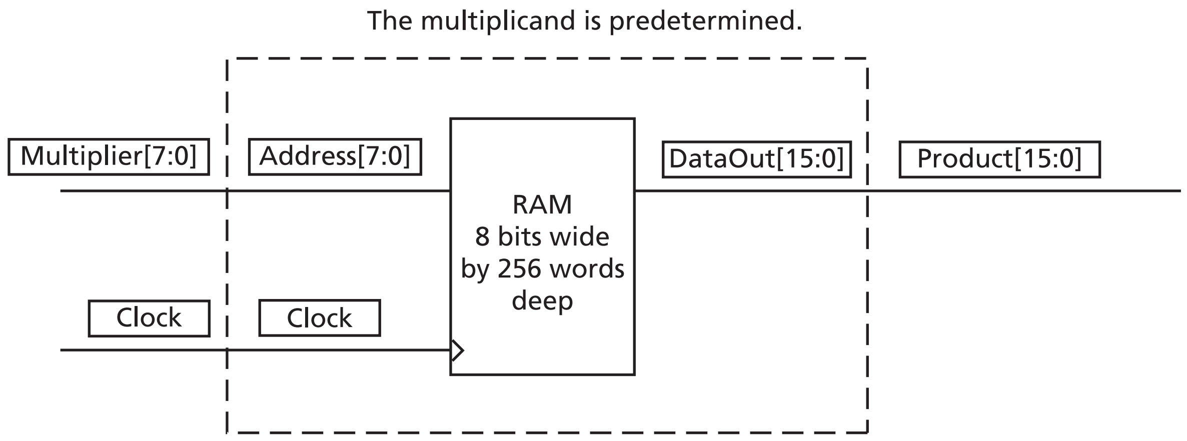

The multiplicand is predetermined.
Multiplier[7:0]
Address[7:0]
RAM 8 bits wide by 256 words deep
DataOut[15:0]
Product[15:0]
Clock
Clock

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 4x4 multiplier, the RAM-based multiplier is much faster than the equivalent Booth multiplier provided by the ACTgen 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 8x8 multiplier, the amount of memory required to implement the 8x8 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. 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 4x4 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 as the Booth multiplier.

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

		Utilization	
Multiplier Used	**Performance MHz**	**Core Tiles**	**RAM Blocks**
4x4 RAM multiplier	293	0	1
4x4 Booth multiplier	98	79	0
4x4 pipelined Booth multiplier	158	92	0
8x8 Booth multiplier	68	305	0
8x8 Booth multiplier with 1 pipeline stage	102	344	0
8x8 Booth multiplier with 2 pipeline stage	123	386	0
8x8 Booth multiplier with 3 pipeline stage	120	431	0
8x8 partial product multiplier	63	196	4
8x8 partial product multiplier with pipelining	129	311	4
8x8 constant coefficient multiplier	281	2	1

Note: *Timing numbers are based on worst-case, commercial numbers for an A3P250 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/E RAM makes preloading the memory block very simple. Since the memory in the ProASIC3/E 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 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 *ProASIC3/E RAM/FIFO Blocks* 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 a four-bit constant coefficient multiplier with a four-bit multiplicand value of E/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 deasserted, and the RAM block is ready to be used as a multiplier. Since the memory in the ProASIC3/E is synchronous, the multiplier acts as a synchronous multiplier.

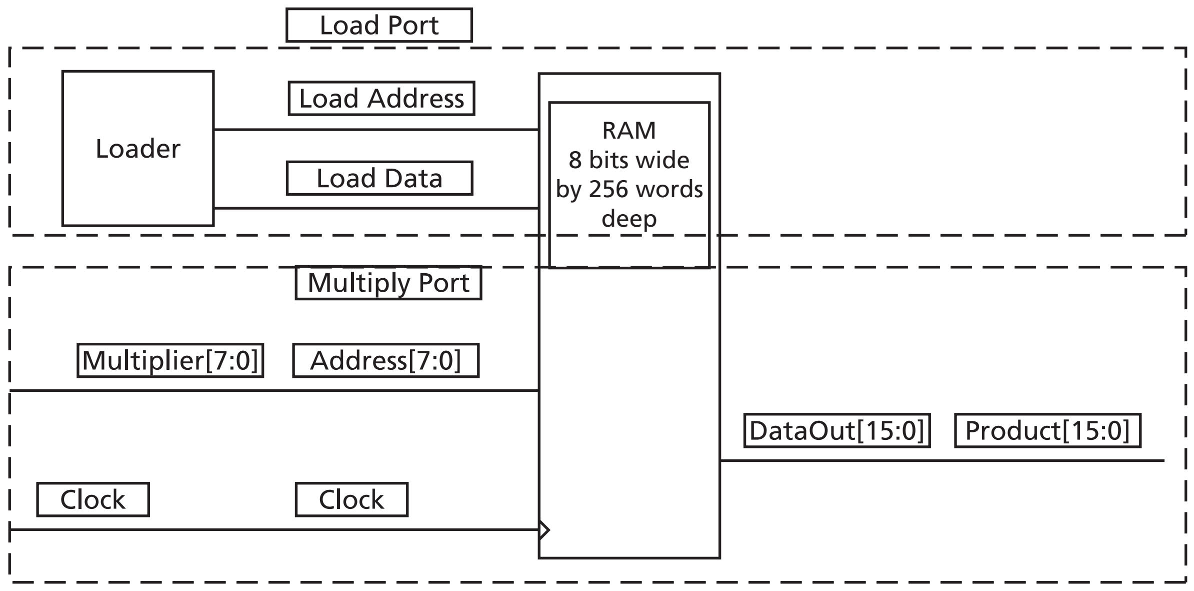


Figure 5 • **Constant Coefficient Multiplier Logic**

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 ProASIC3/E 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 the most efficient look-up table multiplier.

Related Documents

Application Notes

ProASIC3/E SRAM/FIFO Blocks

http://www.actel.com/documents/PA3FROM_AN.pdf

List of Changes

Previous Version	Changes in the current version 51900074-1/3.05*	Page
51900074-0/1.05*	Table 1 was updated.	4

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

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 prepared 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

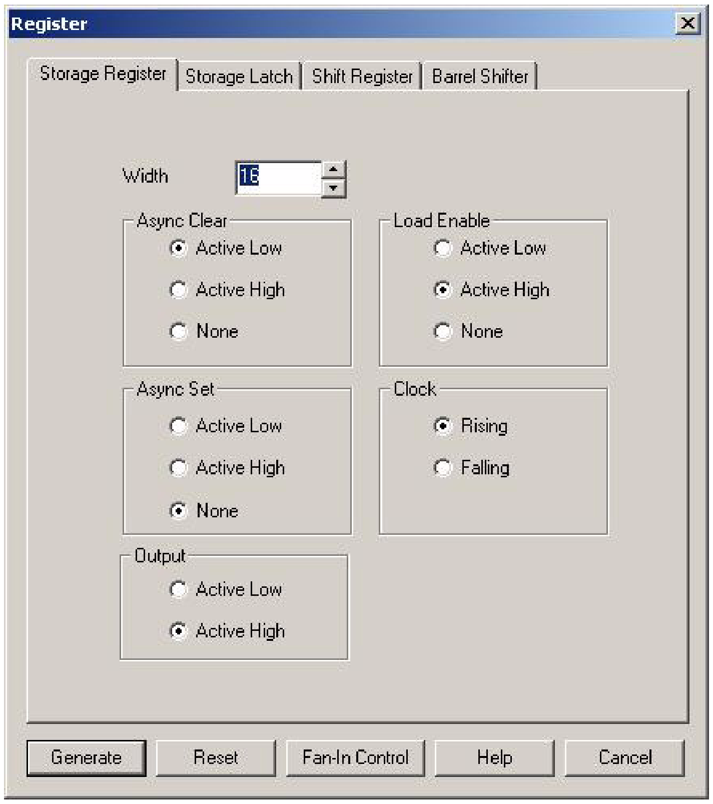
The loader module accepts a clock, a clear, and an enable signal. It ties together the register, counter, and adder which performs the actual data loading for the RAM.

```
-- 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);

data1 <= data2;

end struct;
```

Reg16

The reg16 register file is generated using ACTgen. The register file is an 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 following parameters (Figure 6).



Width 16

Async Clear: [✓ Active Low] Active High None

Load Enable: Active Low [✓ Active High] None

Async Set: Active Low Active High [✓ None]

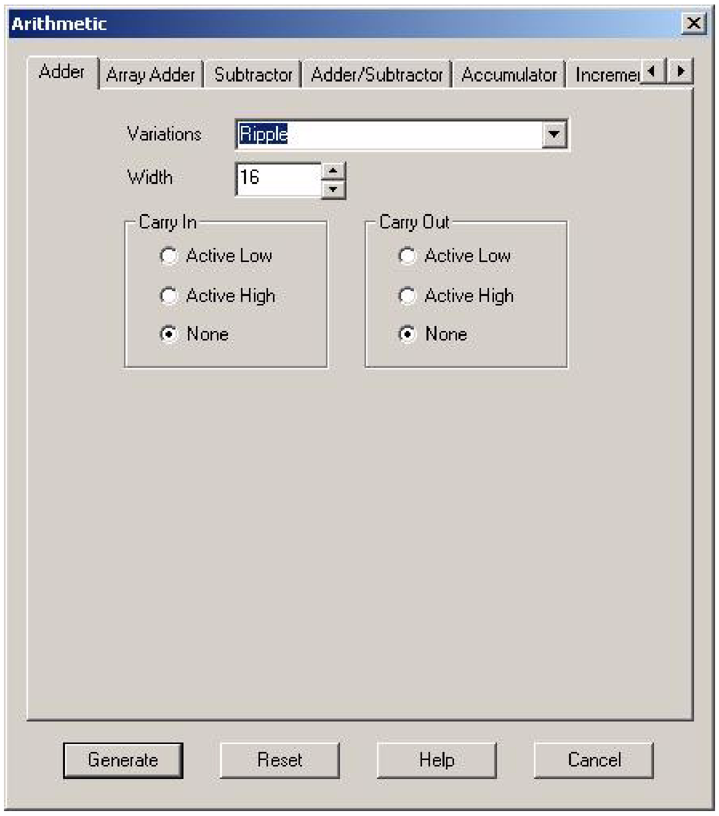
Clock: [✓ Rising] Falling

Output: Active Low [✓ Active High]

Figure 6 • **Reg16**

Adder

The adder component is a 16-bit adder with continually increments the values loaded into the RAM by a value of E/h (Figure 7).



Variations: Ripple
Width: 16

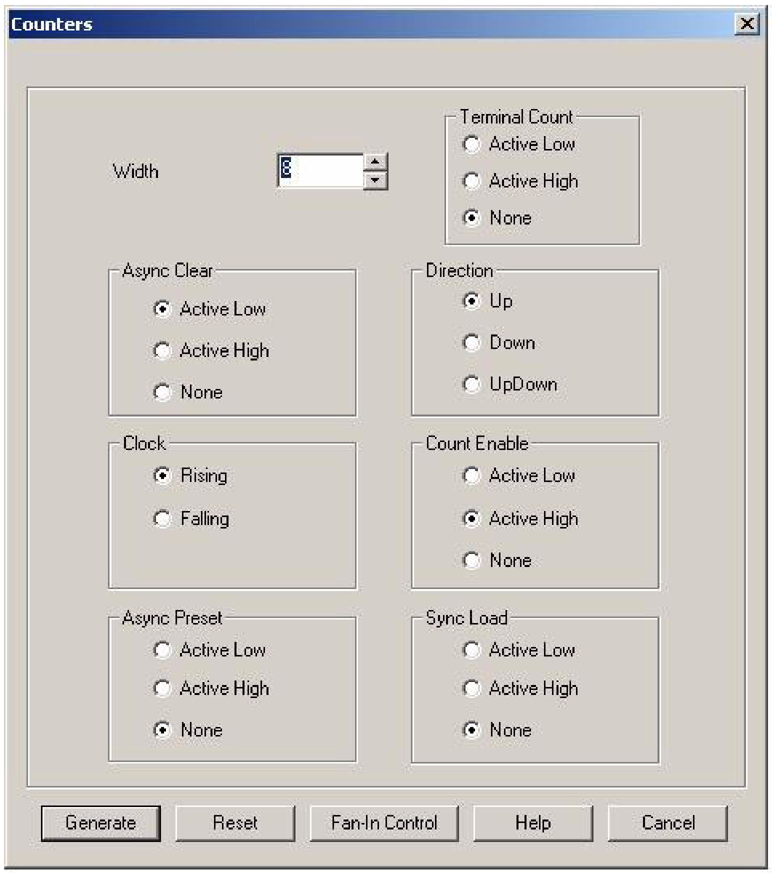
Carry In: ( ) Active Low ( ) Active High [✓ None]

Carry Out: ( ) Active Low ( ) Active High [✓ None]

Figure 7 • **Adder**

Counter

The counter is a eight-bit counter which cycles through all the address values for the RAM. This counter is also generated using ACTgen with the following parameters (Figure 8).



Counters

Width: 8

Terminal Count: Active Low / Active High / [✓ None]

Async Clear: [✓ Active Low] / Active High / None

Direction: [✓ Up] / Down / UpDown

Clock: [✓ Rising] / Falling

Count Enable: Active Low / [✓ Active High] / None

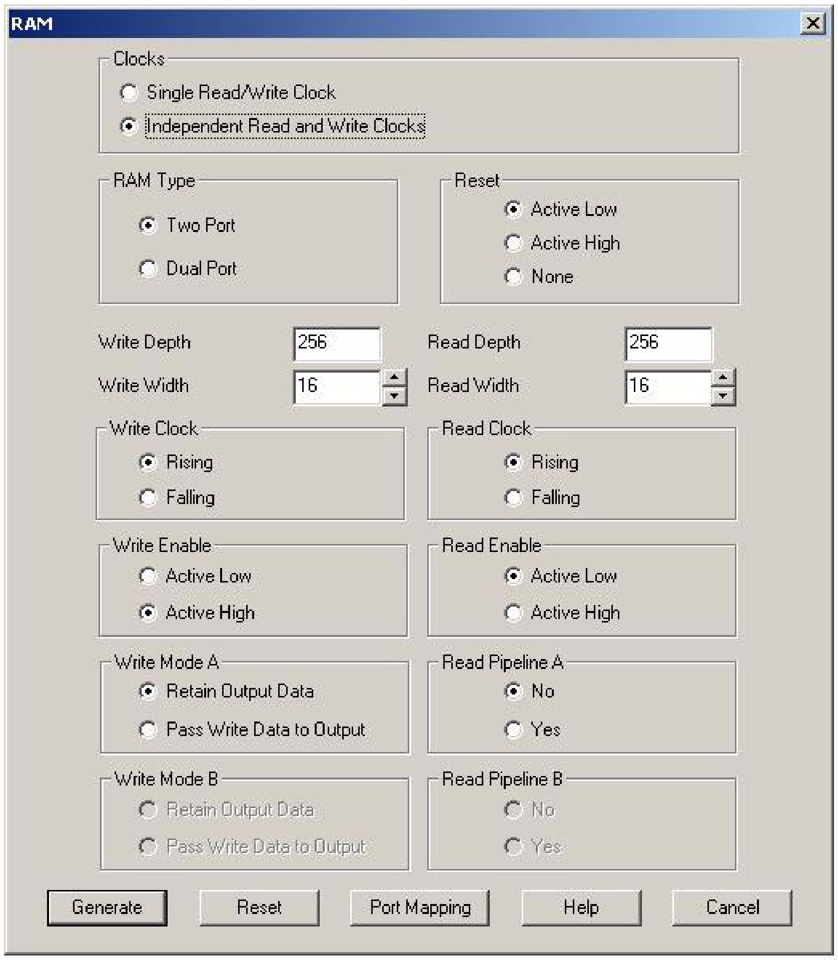
Async Preset: Active Low / Active High / [✓ None]

Sync Load: Active Low / Active High / [✓ None]

Figure 8 • **Counter**

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 (Figure 9).



Clocks
Single Read/Write Clock [✓ Independent Read and Write Clocks]

RAM Type
[✓ Two Port] Dual Port

Reset
[✓ Active Low] Active High None

Write Depth 256
Write Width 16
Read Depth 256
Read Width 16

Write Clock
[✓ Rising] Falling

Read Clock
[✓ Rising] Falling

Write Enable
Active Low [✓ Active High]

Read Enable
[✓ Active Low] Active High

Write Mode A
[✓ Retain Output Data] Pass Write Data to Output

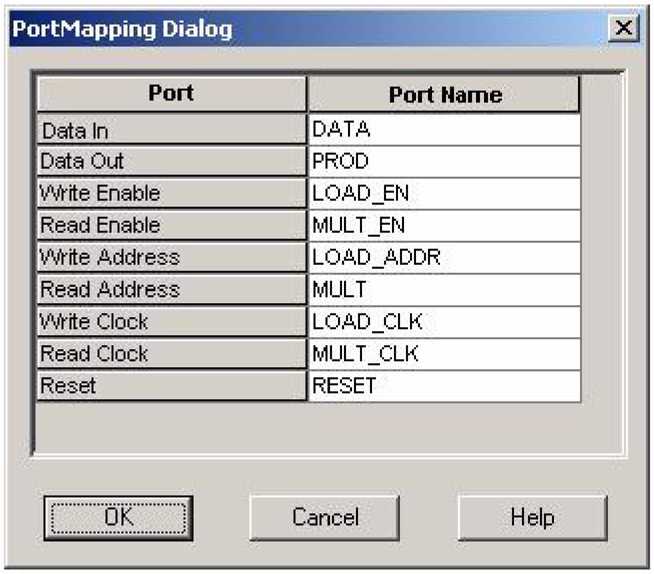
Read Pipeline A
[✓ No] Yes

Write Mode B
Retain Output Data Pass Write Data to Output

Read Pipeline B
No Yes

Figure 9 • **RAM16x8**

In Figure 10, the following port map is used in order to make the signals more meaningful as a multiplier.



Port	Port Name
Data In	DATA
Data Out	PROD
Write Enable	LOAD_EN
Read Enable	MULT_EN
Write Address	LOAD_ADDR
Read Address	MULT
Write Clock	LOAD_CLK
Read Clock	MULT_CLK
Reset	RESET

Figure 10 • **Port Mapping Dialog**

http://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