XILINX®

XAPP467 (v1.1) May 13, 2003

Using Embedded Multipliers in Spartan-3 FPGAs

Summary

Dedicated 18x18 multipliers speed up DSP logic in the Spartan™-3 family. The multipliers are fast and efficient at implementing signed or unsigned multiplication of up to 18 bits. In addition to basic multiplication functions, the embedded multiplier block can be used as a shifter or to generate magnitude or two's-complement return of a value. The multipliers can be cascaded with each other or CLB logic for larger or more complex functions.

Introduction

Spartan-3 FPGAs have a number of features to fortify the chip's arithmetic capabilities. Carry logic and dedicated carry routing continues to be provided as in past generations. Dedicated AND gates in the CLBs accelerate array multiplication operations. The newest and most significant addition is the dedicated 18x18 two's-complement multiplier block. With 4 to 104 of these dedicated multipliers in each device, fast arithmetic functions can be implemented with minimal use of the general-purpose resources. In addition to the performance advantage, dedicated multipliers require less power than CLB-based multipliers.

The embedded multipliers offer fast, efficient means to create 18-bit signed by 18-bit signed multiplication products. The multiplier blocks share routing resources with the Block SelectRAM™ memory, allowing for increased efficiency for many applications. Cascading of multipliers can be implemented with additional logic resources in local Spartan-3 slices.

Applications such as signed-signed, signed-unsigned, and unsigned-unsigned multiplication, logical, arithmetic, and barrel shifters, two's-complement and magnitude return are easily implemented.

The 18-bit x 18-bit multipliers can be quickly created using the CORE Generator™ system, or they can be instantiated (or inferred) using VHDL or Verilog.

Two's-Complement Signed Multiplier

Data Flow

Each embedded multiplier block (MULT18X18 primitive) supports two independent dynamic data input ports: 18-bit signed or 17-bit unsigned. The two inputs are referred to as the multiplicand and the multiplier, or the factors, while the output is the product. The MULT18X18 primitive is illustrated in Figure 1.

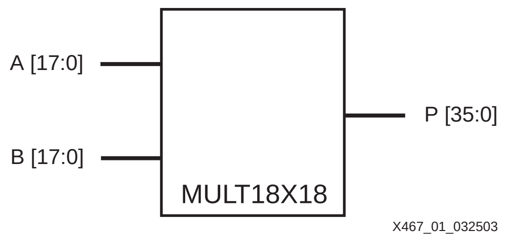


Figure 1: **Embedded Multiplier**

© 2003 Xilinx, Inc. All rights reserved. All Xilinx trademarks, registered trademarks, patents, and further disclaimers are as listed at http://www.xilinx.com/legal.htm. All other trademarks and registered trademarks are the property of their respective owners. All specifications are subject to change without notice.

NOTICE OF DISCLAIMER: Xilinx is providing this design, code, or information "as is." By providing the design, code, or information as one possible implementation of this feature, application, or standard, Xilinx makes no representation that this implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of the implementation, including but not limited to any warranties or representations that this implementation is free from claims of infringement and any implied warranties of merchantability or fitness for a particular purpose.

In addition, efficient cascading of multipliers up to 35-bit x 35-bit signed can be accomplished by using four embedded multipliers, one 36-bit adder, and one 53-bit adder. See Figure 6.

Binary multiplication is similar to regular multiplication with the multiplicand multiplied by each bit of the multiplier to generate partial products, and then the partial products added together to create the result. The Xilinx multiplier block uses the modified Booth algorithm, in effect using multiplexers to create the partial products.

Timing Specification

The result is generated faster for the LSBs than the MSBs, since the MSBs require more levels of addition, so timing specifications are different for each of the 36 multiplier outputs. Designs should use only as many output bits as are necessary. For example, if two unsigned numbers will never have a product of 2^{35} or higher, the P[35] output is always zero. For any pair of signed numbers of n bits, if you will never have -2^{n-1} x -2^{n-1}, then the MSB is always identical to the next lower-order bit (P[2n-1] = P[2n-2]). Also consider that if some outputs must have longer routing delays, they should be put on the output LSBs to balance with the MSB delays.

For the same reason, the data input setup time for the pipelined multiplier will be shorter for the MSBs than the LSBs, but the timing parameters do not differentiate between pins for setup time. For additional safety margin in a design, slower inputs should be put on the MSBs. The Reset and Clock Enable inputs have much faster setup times than any of the data inputs, and all have zero hold times. The timing parameter name "$t_{MULIDCK}$" (MULtiplier Input Data to ClocK) is used for both the data and control inputs, but will have different values for each type.

Library Primitives

Two library primitives are available for the embedded multipliers. Table 1 describes these primitives.

Table 1: **Multiplier Primitives**

Primitive	A Width	B Width	P Width	Signed/Unsigned	Output
MULT18X18	18	18	36	Signed (Two's Complement)	Combinatorial
MULT18X18S	18	18	36	Signed (Two's Complement)	Registered

The registered version of the multiplier adds a clock input C, an active-High Clock Enable CE, and a synchronous Reset R (see Figure 2). The registers are implemented in the multiplier itself and do not require any other resources. The control inputs C, CE, and R all have built-in programmable polarity. The data inputs, clock enable, and reset all must meet a setup time before the clock edge, and the data on the P outputs changes after the clock-to-output delay.

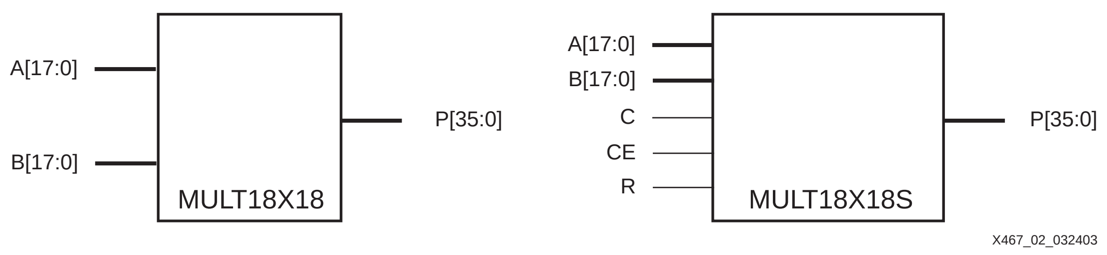


Figure 2: **Combinatorial and Registered Multiplier Primitives**

The pin names used in the Xilinx implementation tools, such as the FPGA Editor, are identical to those used in the library primitives.

VHDL Instantiation Template

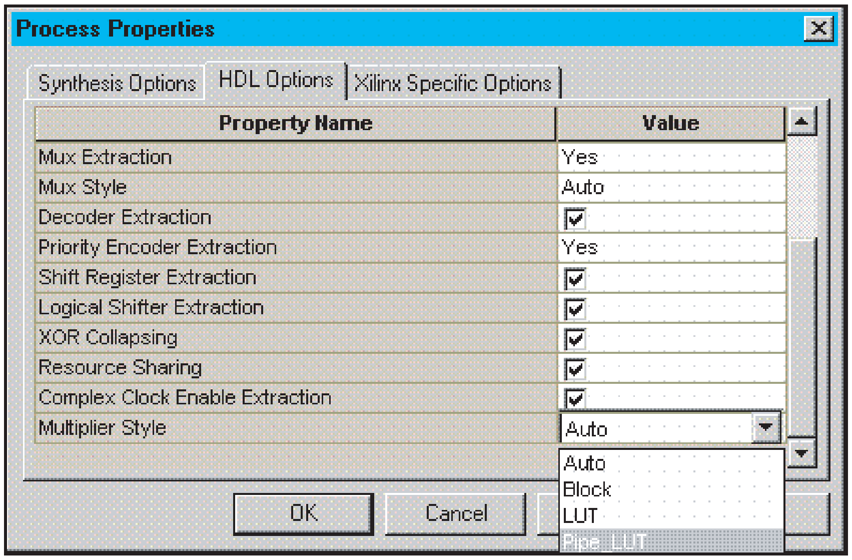
```
-- Component Declaration for MULT18X18 should be placed
-- after architecture statement but before begin keyword
component MULT18X18
  port ( P : out STD_LOGIC_VECTOR (35 downto 0);
         A : in STD_LOGIC_VECTOR (17 downto 0);
         B : in STD_LOGIC_VECTOR (17 downto 0));
end component;
-- Component Attribute specification for MULT18X18
-- should be placed after architecture declaration but
-- before the begin keyword
-- Attributes should be placed here
-- Component Instantiation for MULT18X18 should be placed
-- in architecture after the begin keyword
MULT18X18_INSTANCE_NAME : MULT18X18
  port map (P => user_P,
            A => user_A,
            B => user_B);
```

Verilog Instantiation Template

```
MULT18X18 MULT18X18_instance_name (.P (user_P),
                                   .A (user_A),
                                   .B (user_B));
```

MULT_STYLE Constraint

The MULT_STYLE constraint controls the implementation of the MULT18X18 primitives. In the Project Navigator (see Figure 3), the default is that the Xilinx Synthesis Tool (XST) will select the best type of implementation. To ensure that the embedded multipliers are used, set MULT_STYLE = Block or select "Block" for the "Multiplier Style" property in the Project Navigator. The MULT_STYLE constraint can also be applied globally at the XST command line or attached to a MULT18X18 primitive. For the MULT18X18S, attach the MULT_STYLE constraint to the component, not the output bus. See the Constraints Guide for more information.



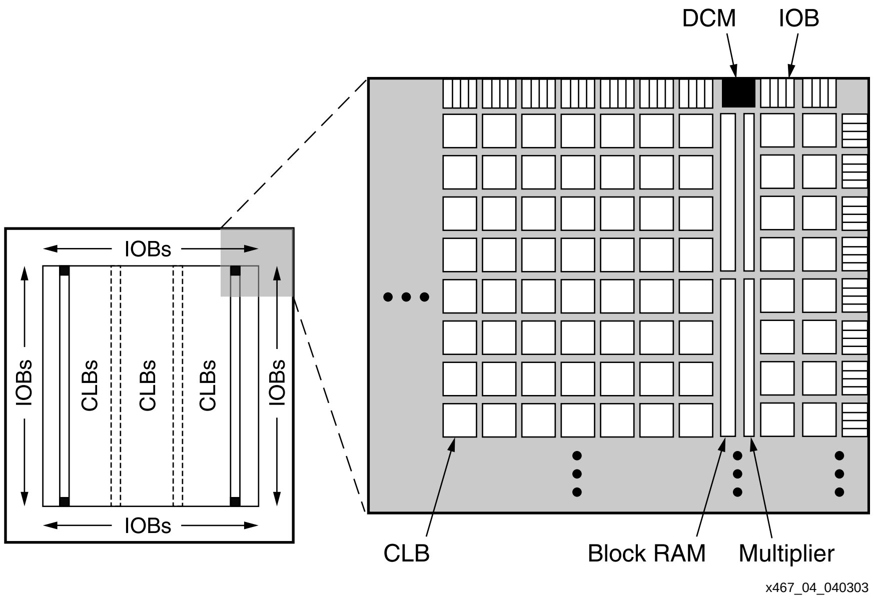
Property Name	Value
Mux Extraction	Yes
Mux Style	Auto
Decoder Extraction	✓
Priority Encoder Extraction	Yes
Shift Register Extraction	✓
Logical Shifter Extraction	✓
XOR Collapsing	✓
Resource Sharing	✓
Complex Clock Enable Extraction	✓
Multiplier Style	Auto

X467_03_032403

Figure 3: **Setting Multiplier Style in Project Navigator Process Properties**

Multipliers in the Spartan-3 Architecture

The multipliers are located adjacent to the block RAM, making it convenient to store inputs or results in the block memory (see Figure 4). There are two or four columns of multipliers in each device. Where there are two columns, they have two rows of CLBs between them and the edge, allowing the multiplier to be easily driven by CLB or IOB logic. There are four CLBs, or 16 slices and 32 LUTs, on either side of a given multiplier block, allowing 32 input and output signals to be connected immediately adjacent to the multiplier block. One possible high-speed layout is to put A[15:0] on one side, B[15:0] on the other side, and intersperse the P[31:0] outputs on both sides. For a full-size 18x18 multiplier, the extra inputs and outputs can connect to the next CLB column. For best performance, pipeline the inputs with registers in the adjacent CLBs.



Notes:
1. The two additional block RAM/multiplier columns of the XC3S4000 and XC3S5000 devices are shown with dashed lines. The XC3S50 device has a single column of block RAM/multipliers along the left edge.

Figure 4: **Location of Multipliers in Spartan-3 Architecture**

The 18-bit width of the Spartan-3 multiplier is unusual but matches with the 18-bit width of the block RAM, which includes parity bits. Standard 8-bit or 16-bit multipliers can be created by using part of the multiplier block, or a 32-bit multiplier can be created via cascading. The Xilinx architecture allows any non-standard bit width to be implemented, exactly matching the needs of the application. Unused multiplier inputs are connected automatically to zero via connections to unused LUTs that are set to zero.

Table 2: **Number of Multipliers per Spartan-3 Device**

Device	Multiplier Columns	Multipliers
XC3S50	1	4
XC3S200	2	12
XC3S400	2	16
XC3S1000	2	24
XC3S1500	2	32
XC3S2000	2	40
XC3S4000	4	96
XC3S5000	4	104

Expanding Multipliers

Multiplication using inputs with more than 18 bits is possible by decomposing the multiplication process into smaller subprocesses. The binary representation of either input can be split at any point, provided the proper weighting and sign of the MSBs is taken into account. Splitting off the 18 MSBs of the input makes the best use of the 18-bit signed multipliers.

For example, Figure 5 shows how a 22x16 multiplier could be implemented. The 22-bit value is decomposed into an 18-bit signed value and a 4-bit unsigned value from the LSBs. Two partial products are formed. The first is a 20-bit signed product, which is the result of multiplying the 16-bit signed value by the 4-bit unsigned section. The second is a 34-bit signed product, formed by multiplying the 16-bit signed value by the 18-bit signed section. The addition process restores the weighting of the products (note the least significant bits of the first product bypass the addition) and forms the final 38-bit product. Since the first product is signed, the 20-bit value needs to be sign-extended before addition. The adder itself only needs to be 34 bits, requiring 17 slices.

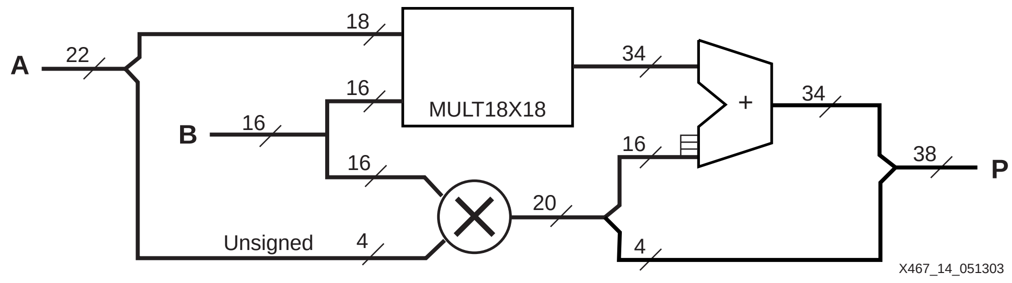


Figure 5: **22x16 Multiplier Implementation**

The implementation can vary depending on the performance needs and available resources. The second multiplier can be implemented in the MULT18X18 resource or in CLBs if it is small. Pipelining can be added to improve performance, using the built-in capabilities of the dedicated multipliers. If both inputs are greater than 18 bits, then four partial products are formed, but the purely unsigned result from the LSBs simply can be concatenated with the 36-bit signed product of the MSBs and added to the other two results.

Figure 6 represents the cascaded scheme used to implement a 35-bit by 35-bit signed multiplier utilizing four embedded multipliers and two adders.

The fixed adder is 53 bits wide (17 LSBs are always 0 on one input).

The 34-bit by 34-bit unsigned submodule is constructed in a similar manner with the most significant bit on each operand being tied to logic Low.

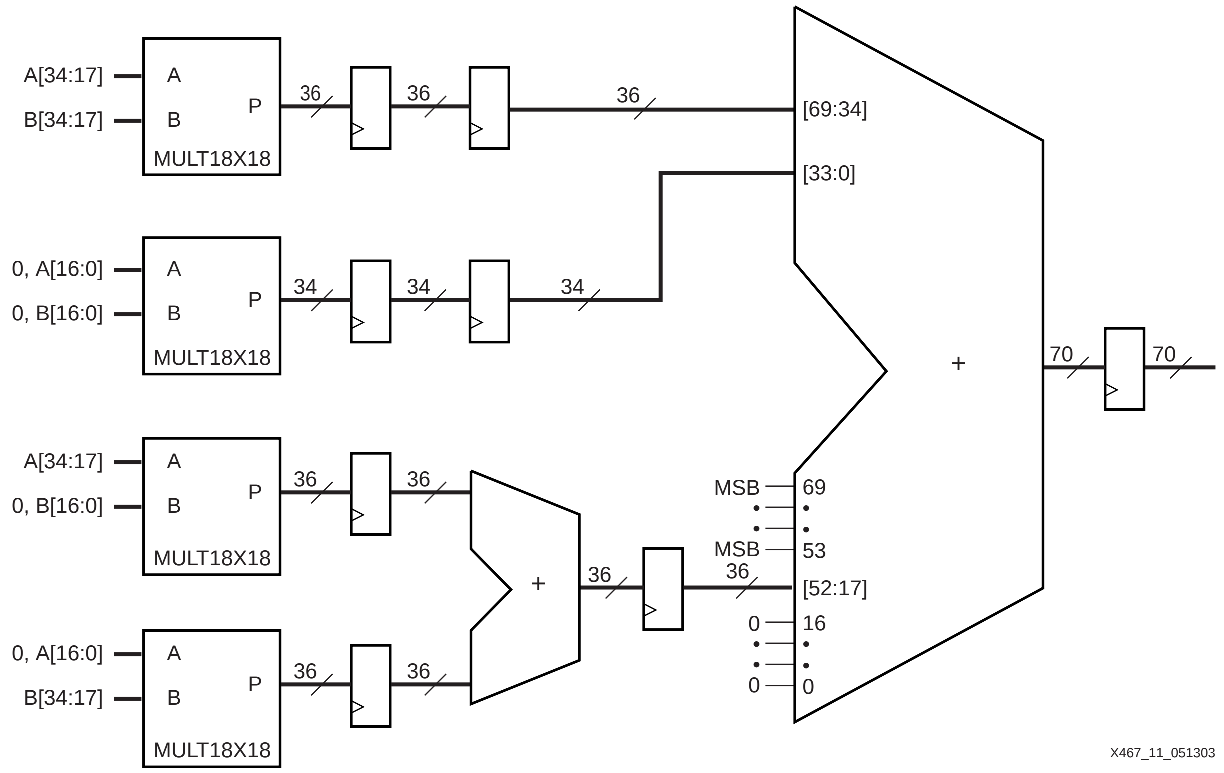


Figure 6: **35x35 Signed Multiplier**

Two Multipliers in a Single Primitive

The dedicated multiplier can be used to multiply two smaller numbers at the same time. By putting one value on the LSBs and one on the MSBs, two independent results can be obtained as long as the results do not overlap with each other on the outputs. Shifting one of the values n positions to the MSBs is the same as multiplying it by 2^n. If the value shifted to the MSBs is X, then the new value is $X * 2^n$. If the value on the LSBs is Y, then the complete multiplier input is $X * 2^n + Y$.

For simplified illustration purposes, an assumption of two squares being implemented in the same MULT18X18 primitive is used. The following equation shows the form of the multiplication.

Two Multipliers per Primitive:

$$(X * 2^n + Y)(X * 2^n + Y) = (X^2 * 2^{2n}) + (XY * 2^{n+1}) + (Y^2)$$

For values 0 on X or Y, the equation becomes:

$X^2 * 2^{2n}$ {Y=0} (X^2 on the output MSBs)

Y^2 {X=0} (Y^2 on the output LSBs)

0 {X=0, Y=0}

With both X and Y at non-zero values, care must be taken to avoid overlap between the results on the MSBs and LSBs and the middle term $(XY * 2^{n+1})$. Two multipliers can coexist in one MULT18X18 primitive, if the conditions in the following inequalities are met when neither X nor Y are 0.

Inequality Conditions for Two Multipliers per Primitive:

$$(X^2 * 2^{2n})_{min} > (XY * 2^{n+1})_{max}, (XY * 2^{n+1})_{min} > (Y^2)_{max}$$

Table 3 shows values for X and Y where these conditions are met.

Table 3: **Two Multipliers per MULT18X18 Allowable Sizes**

X * X		Y * Y	
Signed Size	**Unsigned Size**	**Signed Size**	**Unsigned Size**
7 X 7	6 X 6	-	4 X 4
6 X 6	5 X 5	-	5 X 5
5 X 5	4 X 4	3 X 3	6 X 6
4 X 4	3 X 3	3 X 3	7 X 7
3 X 3	2 X 2	4 X 4	8 X 8

Figure 7 represents the MULT18X18 connections for calculating the square of both a 6-bit signed number and a 5-bit unsigned number.

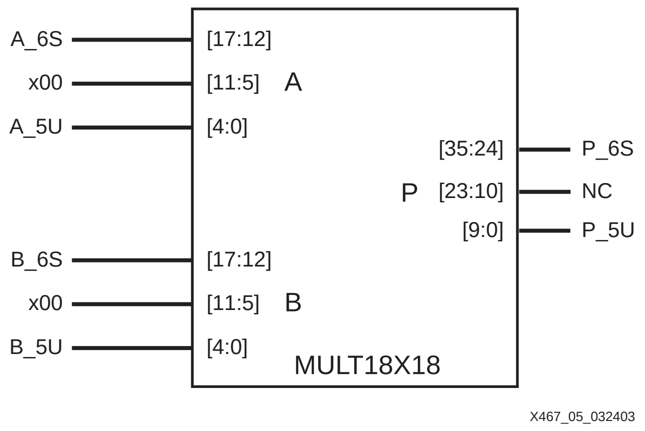


Figure 7: **Two Multipliers in One Primitive**

Design Entry

There are many options for including the Spartan-3 multiplier in a design. The library primitives MULT18X18 and MULT18X18S described earlier can be instantiated in the schematic or HDL code. Synthesis tools can infer a multiplier block from the multiply operator, including Xilinx XST, Synplicity Synplify, and Mentor LeonardoSpectrum. They will infer the MULT18X18S when the operation is controlled by a clock for a synchronous multiplier.

LeonardoSpectrum features a pipeline multiplier that involves putting levels of registers in the logic to introduce parallelism and, as a result, use CLB resources instead of the dedicated multipliers. A certain construct in the input RTL source code description is required to allow the pipelined multiplier feature to take effect. See the Synthesis and Simulation Design Guide for more information.

The following VHDL example will infer the MULT18X18S using XST or Synplify:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity mult18x18s is
  port ( a : in std_logic_vector(7 downto 0);
         b : in std_logic_vector(7 downto 0);
         clk : in std_logic;
         prod : out std_logic_vector(15 downto 0));
end mult18x18s;
architecture arch_mult18x18s of
```

```
  mult18x18s is
begin
process(clk) is begin
  if clk'event and clk = '1' then
    prod <= a*b;
  end if;
end process;
end arch_mult18x18s;
```

The following is a Synchronous Multiplier VHDL Example coded for LeonardoSpectrum:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity mult18x18s is
  port(  clk: in std_logic;
         a: in std_logic_vector(7 downto 0);
         b: in std_logic_vector(7 downto 0);
         prod: out std_logic_vector(15 downto 0));
end mult18x18s;
architecture arch_mult18x18s of
  mult18x18s is
signal reg_prod : std_logic_vector(15 downto 0);
begin
process(clk)
begin
  if(rising_edge(clk))then
    reg_prod <= a * b;
    prod <= reg_prod;
  end if;
end process;
end arch_mult18x18s;
```

The following is a Synchronous Multiplier Verilog Example coded for Synplify and XST:

```
module mult18x18s(a,b,clk,prod);
  input [7:0] a;
  input [7:0] b;
  input clk;
  output [15:0] prod;
  reg [15:0] prod;
  always @(posedge clk) prod <= a*b;
endmodule
```

The following is a Synchronous Multiplier Verilog Example coded for LeonardoSpectrum:

```
module mult18x18s (a,b,clk,prod);
  input [7:0] a;
  input [7:0] b;
  input clk;
  output [15:0] prod;
  reg [15:0] reg_prod, prod;
  always @(posedge clk) begin
  reg_prod <= a*b;
  prod <= reg_prod;
endmodule
```

Using the CORE Generator System

Multipliers that make use of the embedded Spartan-3 18-bit x 18-bit two's-complement multipliers can be easily generated using v6.0 of the CORE Generator Multiplier module. This core is available with version 5.1i and later of the CORE Generator system. Features of the Multiplier Generator include:

- Generates parallel multipliers using the dedicated multiplier blocks
  - Also can use other resources for parallel multipliers or generate sequential/serial-sequential, and fixed/reloadable constant coefficient multipliers
- Supports two's-complement signed/unsigned modes
- Supports inputs ranging from 1 to 64 bits wide
- Supports outputs ranging from 1 to 129 bits wide
- Generates purely combinatorial and fully pipelined implementations
- Provides optional registered output with optional clock enable and asynchronous and synchronous clears
- Provides optional handshaking signals

Figure 8 shows the logic symbol for the Core Multiplier Generator. The RFD (Ready For Data) output goes High to indicate the multiplier is ready to accept data. The ND (New Data) input can be asserted to indicate new data is available on the multiplier inputs. The RDY (Ready) signal indicates that the output is the current product. LOADB and SWAPB are used in constant coefficient multipliers.

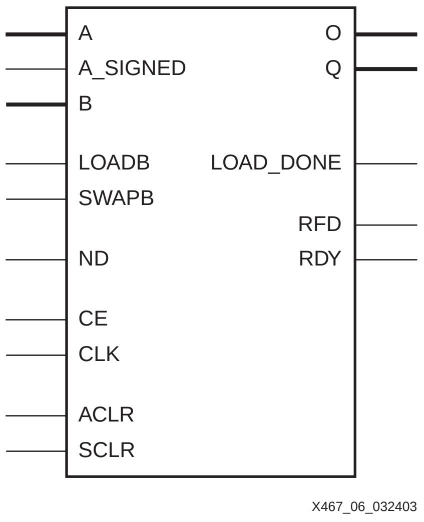


Figure 8: **Core Multiplier Generator Symbol**

The CORE Generator system uses the embedded multiplier for the default Parallel multiplier type. The Multiplier Construction option gives the user the choice to implement the function in look-up tables instead.

Figure 9 shows the timing diagram for the Multiplier Generator.

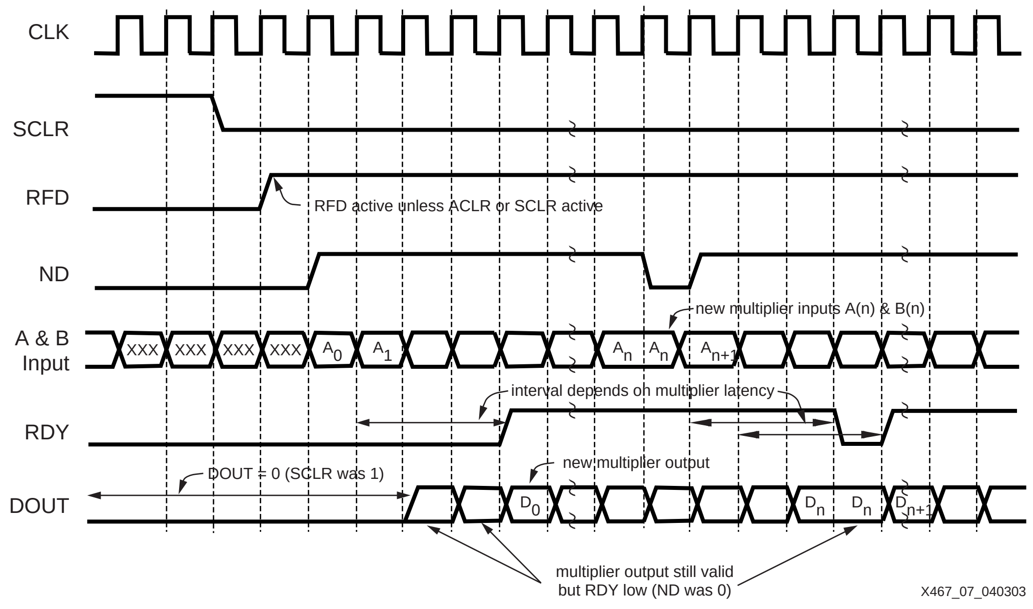

CLK
SCLR
RFD
ND
A & B Input
RDY
DOUT
RFD active unless ACLR or SCLR active
new multiplier inputs A(n) & B(n)
XXX
XXX
XXX
XXX
A0
A1
An
An
An+1
interval depends on multiplier latency
DOUT = 0 (SCLR was 1)
new multiplier output
D0
Dn
Dn
Dn+1
multiplier output still valid but RDY low (ND was 0)
X467_07_040303

Figure 9: **Multiplier Generator Timing Diagram**

System Generator

The Multiplier Generator is used by the System Generator for DSP when the MULT block is used. System Generator presents a high level and abstract view of the design, but also exposes key features in the underlying silicon, making it possible to build extremely high-performance FPGA implementations. The System Generator also provides blocks for compiling MATLAB® M-code into synthesizable HDL code. The System Generator uses the embedded multiplier when a parallel multiplier is selected and the use of the dedicated multiplier is checked in the System Generator interface.

MAC Cores

The CORE Generator system and the System Generator can also implement more complex functions using the multiplier as a building block. The Multiply Accumulator (MAC) core supports up to 32-bit inputs and optional user-defined pipelining. The options of an Embedded or LUT Based implementation control whether the dedicated multipliers or CLB resources are used for the function. The MAC implementation uses relatively few CLB resources beyond the dedicated multipliers and provides flexibility that is key to matching a design to the lowest density and lowest cost solution possible.

The MAC and MAC-based FIR filters include an automatic pipeline control which is based on required system clock performance. Levels of pipeline will automatically be inserted based on the design requirement for a perfect speed/area trade-off.

Multiplier Submodules

This section describes several example submodules that can be used in a Spartan-3 design. Table 4 lists multipliers and two's-complement return functions that utilize one MULT18X18 primitive and are not registered.

Table 4: **Embedded Multiplier Submodules — Single MULT18X18**

Submodule	A Width	B Width	P Width	Signed/Unsigned
MULT17X17_U	17	17	34	Unsigned
MULT8X8_S	8	8	16	Signed
MULT8X8_U	8	8	16	Unsigned
MULT4X4_S	4	4	8	Signed
MULT4X4_U	4	4	8	Unsigned
TWOS_CMP18	18	-	18	-
TWOS_CMP9	9	-	9	-
MAGNTD_18	18	-	17	-

Figure 10 and Figure 11 represent 4-bit by 4-bit signed multiplier and 4-bit by 4-bit unsigned multiplier implementations, respectively.

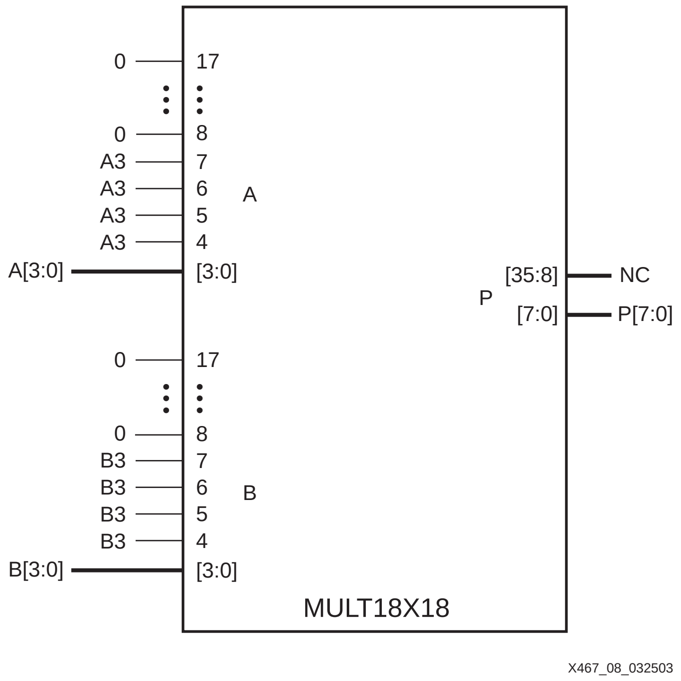


Figure 10: **MULT4X4_S Submodule**

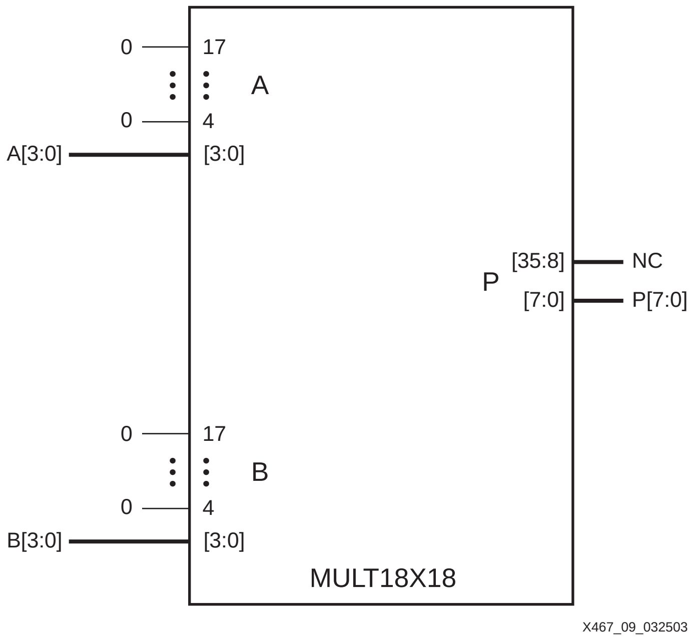


Figure 11: **MULT4X4_U Submodule**

Submodule MAGNTD_18 performs a magnitude return (i.e., absolute value) of a two's-complement number. An incoming negative number returns with a positive number, while an incoming positive number remains unchanged. Submodules TWOS_CMP18 and TWOS_CMP9 perform a two's-complement return function. Additional slice logic can be used with these submodules to efficiently convert sign-magnitude to two's-complement or vice-versa.

Figure 12 shows the connections to a MULT18X18 to create the submodule TWOS_CMP9.

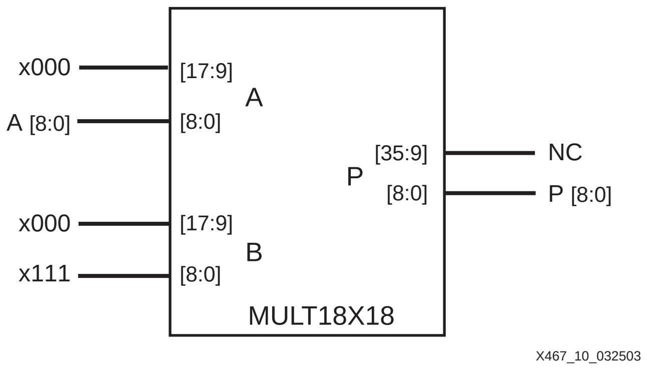


Figure 12: **TWOS_CMP9 Submodule**

VHDL and Verilog Instantiation

VHDL and Verilog instantiation templates are available as examples of primitives and submodules (see **VHDL and Verilog Templates**, page 13).

In VHDL, each template has a component declaration section and an architecture section. Each part of the template should be inserted within the VHDL design file. The port map of the architecture section should include the design signal names.

Port Signals

Data In — A

The data input A provides new data (up to 18 bits) to be used as one of the multiplication operands.

Data In — B

The data input B provides new data (up to 18 bits) to be used as one of the multiplication operands.

Data Out — P

The data output bus P provides the data value (up to 36 bits) of two’s-complement multiplication for operands A and B.

Location Constraints

MULT18X18 embedded multiplier instances can have LOC properties attached to them to constrain placement. MULT18X18 placement locations differ from the convention used for naming CLB locations, allowing LOC properties to transfer easily from array to array.

The LOC properties use the following form:

LOC = MULT18X18_X#Y#

For example, MULT18X18_X0Y0 is the bottom-left MULT18X18 location on the device.

VHDL and Verilog Templates

VHDL and Verilog templates are available for the primitive and submodules.

The following is a template for the primitive:

- SIGNED_MULT_18X18 (primitive: MULT18X18)

The following are templates for submodules:

- UNSIGNED_MULT_17X17 (submodule: MULT17X17_U)
- SIGNED_MULT_8X8 (submodule: MULT8X8_S)
- UNSIGNED_MULT_8X8 (submodule: MULT8X8_U)
- SIGNED_MULT_4X4 (submodule: MULT4X4_S)
- UNSIGNED_MULT_4X4 (submodule: MULT4X4_U)
- TWOS_COMPLEMENTER_18BIT (submodule: TWOS_CMP18)
- TWOS_COMPLEMENTER_9BIT (submodule: TWOS_CMP9)
- MAGNITUDE_18BIT (submodule: MAGNTD_18)

The corresponding submodules have to be synthesized with the design.

Templates for the SIGNED_MULT_18X18 module are provided in VHDL and Verilog code as an example.

VHDL Template

```
-- Module: SIGNED_MULT_18X18
-- Description: VHDL instantiation template
-- 18-bit X 18-bit embedded signed multiplier (asynchronous)
--
-- Device: Spartan-3 Family
---------------------------------------------------------------------
-- Components Declarations
component MULT18X18
  port(
       A :  in std_logic_vector (17 downto 0);
       B :  in std_logic_vector (17 downto 0);
       P : out std_logic_vector (35 downto 0)
  );
end component;
--
-- Architecture Section
--
U_MULT18X18 : MULT18X18
  port map (
    A => , -- insert input signal #1
    B => , -- insert input signal #2
    P =>   -- insert output signal
  );
```

Verilog Template

```
// Module: SIGNED_MULT_18X18
// Description: Verilog instantiation template
// 18-bit X 18-bit embedded signed multiplier (asynchronous)
//
// Device: Spartan-3 Family
//-------------------------------------------------------------------
// Instantiation Section
//
MULT18X18 U_MULT18X18
  (
    .A () , // insert input signal #1
    .B () , // insert input signal #2
    .P ()   // insert output signal
  );
```

Alternative Applications to Multiplication

Since binary multiplication by 2^n is the same as shifting the value n places, a multiplier can be used as a shifter or other general-purpose resource. These can be considered in applications that otherwise would not need the large number of available multipliers.

Shifter

A multiplier can be used as a shifter. One operand is routed to the output, shifted by n positions, if the other operand is a power of two (2^n). Since the sign-bit (MSB) cannot be used to control the shift, the 18x18 two's-complement multiplier can shift by 0 to 16 positions.

Of the 36 output lines, those less significant than the shifted data lines are automatically filled with zeros; those more significant than the shifted data are filled with zeros or ones, depending on the state of the MSB input. This is the natural result of the two's-complement multiplication.

The user can either perform a logic shift of 17 input bits by holding the MSB input Low, or perform an arithmetic shift of an 18-bit two's-complement number, effectively sign-extending the MSB.

A conventional CLB-based shifter would use an array of n multiplexers, each with n inputs, and require a large amount of routing resources. Multiplier-based shifters larger than 18 bits, and barrel shifters of any length, require external OR gating of the outputs, but use far fewer CLB resources.

Magnitude Return

To generate the absolute value of a number by using multiplication, multiply by 1 if it is positive (MSB is zero), and multiply by -1 if it is negative (MSB is one). In two's-complement notation, 1 is all zeros ending in a one as the LSB, and -1 is all ones, including the LSB. Therefore, a magnitude return or absolute value generator can be implemented by multiplying by a value with a one as the LSB and the MSB of the input value in all the other bit positions. Figure 13 shows a magnitude return generator.

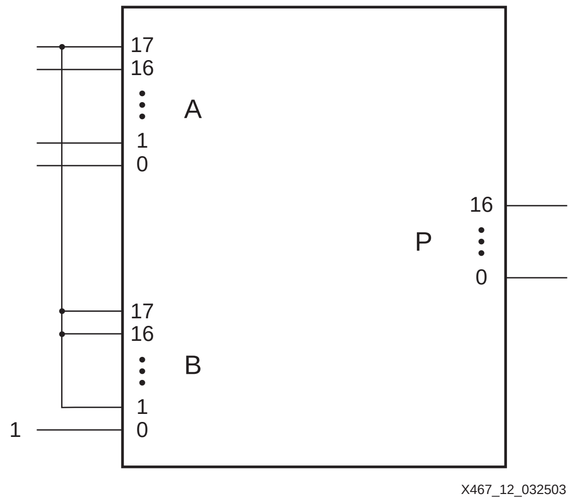


Figure 13: **Magnitude Return**

Two's-Complement Return

Generating the two's complement of a number typically requires only one LUT per bit with the carry logic used for larger numbers. However, if LUTs are heavily used, the multiplier can be used to return the two's complement of the input. Multiplying an input number by an equivalent length number of all ones generates the two's complement of the number over the same length of the output bits. Any extraneous higher-order bits are ignored. Figure 14 shows a two's complement return generator.

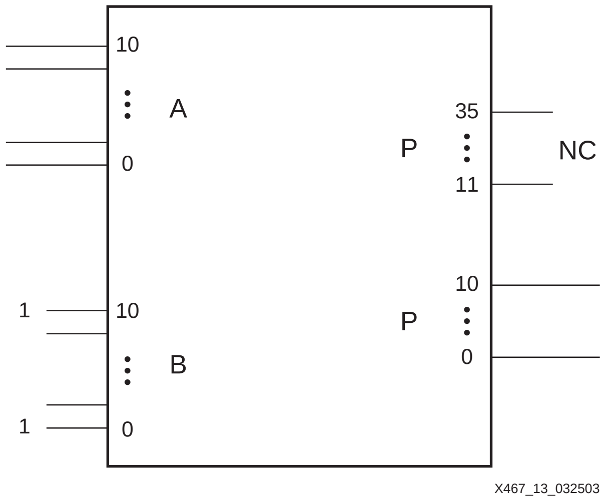


Figure 14: **Two's-Complement Return**

Complex Multiplication

Complex multiplication is multiplication of complex numbers, which contain real and imaginary components with the imaginary unit i equal to the square root of -1. Complex multiplication can be carried out using only three real multiplications: ac, bd, and (a + b)(c + d). The real part of (a + ib)(c + id) is ac - bd, and the imaginary part is (a + b)(c + d) - ac - bd. The large number of multipliers in the Spartan-3 architecture makes it convenient to do even complex multiplication.

Time Sharing in Matrix Multiplication

Many pipelined functions in the computer graphics and video fields are expressed in matrix mathematics. A 3 x 3 matrix multiplication would require 27 multiplies and 18 adds to generate the 3 x 3 matrix result. Color conversion can be described as a 3 x 3 matrix multiplication by a constant, which requires nine multiplies and six adds to generate the three results.

The high-speed capability of a Spartan-3 device allows the user to "time share" the multipliers. Instead of nine multipliers, the design feeds nine sets of inputs resulting in nine sets of results at nine times the clock rate of the system, reducing the multiplier count to one. The adder logic is implemented in CLB resources, and at every third clock, the adder output is stored in output registers to capture the three results. See XAPP284 for more information.

Floating-Point Multiplication

Floating-point values add an exponent to the number and sign bit used in binary multiplication. A 32-bit floating-point multiplier can be implemented using four of the dedicated multiplier blocks and CLB resources. Such multipliers are available from Xilinx AllianceCORE™ partners.

Related Materials and References

- Spartan-3 Family Data Sheet
  Architectural description and timing parameters.
  DS099-1, ***Spartan-3 1.2V FPGA Family:*** **Introduction and Ordering Information** (Module 1)
  DS099-2, ***Spartan-3 1.2V FPGA Family:*** **Functional Description** (Module 2)
  DS099-3, ***Spartan-3 1.2V FPGA Family:*** **DC and Switching Characteristics** (Module 3)
  DS099-4, ***Spartan-3 1.2V FPGA Family:*** **Pinout Tables** (Module 4)
- DSP Central (**http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Xilinx+DSP**)
  Information that will enable you to achieve the maximum benefit from our DSP solutions.
- IP Center (**http://www.xilinx.com/ipcenter**)
  Xilinx and Alliance partner core solutions.
- Xilinx Software Documentation
  (**http://www.xilinx.com/support/sw_manuals/xilinx5/download/**)
  Libraries Guide MULT18X18/S descriptions, Synthesis and Simulation Design Guide instantiation examples for HDL.
- **XAPP284 Matrix Math, Graphics, and Video**
  Uses one multiplier running at 9x the clock rate to provide the nine results for a 3x3 matrix multiplication in one system clock cycle.
- **XAPP636 Optimal Pipelining of the I/O Ports of Virtex-II Multipliers**
  Describes a high-speed, optimized implementation of the dedicated multiplier resulting from pipelined inputs and outputs and effective placement and routing constraints.
- TechXclusives (**http://www.xilinx.com/support/techxclusives/techX-home.htm**)
  See "Using Leftover Multipliers and Block RAM" by Peter Alfke and "Expanding Virtex-II Multipliers" by Ken Chapman.

Conclusion

FPGAs have a significant advantage over general-purpose DSP chips because their logic can be customized for the specific application. Some functions can run over 100 times faster and require much less expense in an FPGA. A key feature to take advantage of is the dedicated multiplier block. Take advantage of the automatic optimization of multiplication logic, and the user controls when necessary to get the exact results desired. The CORE Generator system can create simple multipliers or combine them into more complex functions such as MACs.

Appendix A: Two's-Complement Multiplication

Two's-complement representation allows the use of binary arithmetic operations on signed integers, yielding the correct two's-complement results. Positive two's-complement numbers are represented as simple binary. Negative two's-complement numbers are represented as the binary number that when added to a positive number of the same magnitude equals zero. To calculate the two's complement of an integer, invert the binary equivalent of the number by changing all of the ones to zeros and all of the zeros to ones (also called one's complement), and then add one. The MSB (left-most) bit indicates the sign of the integer; therefore it is sometimes called the sign bit. If the sign bit is zero, the number is positive. If the sign bit is one, the number is negative. To extend a signed integer to a larger width, duplicate the MSB on the left side of the number.

Two's-complement multiplication follows the same rules as binary multiplication, which are the same as the truths of the AND gate:

0 x 0 = 0

0 x 1 = 0

1 x 0 = 0

1 x 1 = 1, and no carry or borrow bits

For example,

$$\begin{array}{ll} & 1111\ 1100 = -4 \\ \times & \underline{0000\ 0100} = \underline{+4} \\ & 1111\ 0000 = -16 \end{array}$$

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
04/06/03	1.0	Initial Xilinx release.
05/13/03	1.1	Updated multiplier information for the XC3S50 device in the **Multipliers in the Spartan-3 Architecture** section. Added new section entitled **Expanding Multipliers**. Added TechXclusives reference to **Related Materials and References** section. Made minor edits for clarification.