Techniques for Implementing Multipliers in Stratix, Stratix GX & Cyclone Devices

August 2003, ver. 1.0 Application Note 306

Introduction

Stratix™, Stratix GX, and Cyclone™ FPGAs have dedicated architectural features that make it easy to implement high-performance multipliers. Stratix and Stratix GX devices feature embedded high-performance multiplier-accumulators (MACs) in dedicated digital signal processing (DSP) blocks. DSP blocks can operate at data rates above 300 million samples per second (MSPS), making Stratix and Stratix GX FPGAs ideal for high-speed DSP applications. In addition to the dedicated DSP blocks, designers can also use the devices' TriMatrix™ memory blocks to implement variable depth/width, high-performance soft multipliers. For example, designers can implement TriMatrix memory blocks as look-up tables (LUTs) that contain partial results from multiplication of input data with coefficients. Cyclone devices have M4K memory blocks which can be used as LUTs to implement variable depth/width high-performance soft multipliers for low cost, high volume DSP applications.

Stratix, Stratix GX, and Cyclone FPGAs can implement the multiplier types shown in Table 1.

Table 1. Supported Multiplier Implementations

Multiplier Type	Description	Stratix	Stratix GX	Cyclone
Soft multiplier	These multipliers are implemented as LUTs in memory, which contains all possible partial results from multiplication. There are five soft multiplier modes: ■ Parallel multiplication ■ Semi-parallel multiplication ■ Sum of multiplication ■ Hybrid multiplication ■ Fully variable multipliers	✓	✓	✓
Multipliers using DSP blocks or logic elements (LEs)	These multipliers are implemented in dedicated DSP blocks or LEs using the `lpm_mult`, `altmult_add`, or `altmult_accum` megafunctions.	✓	✓	-
Hard multiplier	These multipliers are implemented in a combination of DSP blocks and LEs.	✓	✓	-

Tables 2 and 3 show the total number of multipliers available in Stratix and Stratix GX devices using DSP blocks and soft multipliers. Table 4 shows the total number of soft multipliers available in Cyclone devices.

Table 2. Number of Multipliers in Stratix Devices

Device	DSP Blocks (18 × 18)	Soft Multipliers (16 × 16) *Notes (1)*	Total Multipliers *Notes (2), (3)*
EP1S10	24	81	105 (4.38)
EP1S20	40	132	172 (4.30)
EP1S25	40	190	230 (5.57)
EP1S30	48	241	289 (6.02)
EP1S40	56	280	336 (6.00)
EP1S60	72	434	506 (7.03)
EP1S80	88	558	646 (7.34)

Notes to Table 2:
(1) Soft multipliers implemented in sum of multiplication mode. RAM blocks configured with 18-bit data widths and sum of coefficients up to 18 bits.
(2) The number in parentheses represents the increase factor, which is the total number of multipliers with soft multipliers divided by the number of 18 × 18 multipliers supported by DSP blocks only.
(3) The total number of multipliers may vary according to the multiplier mode used.

Table 3. Number of Multipliers in Stratix GX Devices

Device	DSP Blocks (18 × 18)	Soft Multipliers (16 × 16) *Notes (1)*	Total Multipliers *Notes (2), (3)*
EP1SGX10C	24	81	105 (4.38)
EP1SGX10D	24	81	105 (4.38)
EP1SGX25C	40	190	230 (5.57)
EP1SGX25D	40	190	230 (5.57)
EP1SGX25F	40	190	230 (5.57)
EP1SGX40D	56	280	336 (6.00)
EP1SGX40G	56	280	336 (6.00)

Notes to Table 3:
(1) Soft multipliers implemented in sum of multiplication mode. RAM blocks configured with 18-bit data widths and sum of coefficients up to 18 bits.
(2) The number in parentheses represents the increase factor, which is the total number of multipliers with soft multipliers divided by the number of 18 × 18 multipliers supported by DSP blocks only.
(3) The total number of multipliers may vary according to the multiplier mode used.

Table 4. Number of Multipliers in Cyclone Devices

Device	Soft Multipliers (16 × 16) *Notes (1), (2)*
EP1C3	11
EP1C4	14
EP1C6	17
EP1C12	45
EP1C20	56

Notes to Table 4:

(1) Soft multipliers implemented in sum of multiplication mode. RAM blocks configured with 18-bit data widths and sum of coefficients up to 18 bits.

(2) The total number of multipliers may vary according to the multiplier mode used.

This application note describes the dedicated memory and DSP blocks, the supported multiplier types, and includes an example of each type.

Memory Blocks

The Stratix and Stratix GX TriMatrix memories consist of three types of RAM blocks: M512, M4K, and M-RAM. The M512 and M4K RAM blocks are memory blocks with a maximum width of 18 and 36 bits, respectively, and a maximum performance of approximately 300 MHz, which is ideal for implementing soft multipliers.

Tables 5 and 6 show the available TriMatrix memory blocks in Stratix and Stratix GX devices, respectively.

Table 5. Stratix Memory Blocks

Device	M512 RAM (32 × 18 Bits)	M4K RAM (128 × 36 Bits)	M-RAM (4K × 144 Bits)	Total RAM Bits
EP1S10	94	60	1	920,448
EP1S20	194	82	2	1,669,248
EP1S25	224	138	2	1,944,576
EP1S30	295	171	4	3,317,184
EP1S40	384	183	4	3,423,744
EP1S60	574	292	6	5,215,104
EP1S80	767	364	9	7,427,520

Table 6. Stratix GX TriMatrix Memory Blocks

Device	M512 RAM (32 × 18 Bits)	M4K RAM (128 × 36 Bits)	M-RAM (4K × 144 Bits)	Total RAM Bits
EP1SGX10C	94	60	1	920,448
EP1SGX10D	94	60	1	920,448
EP1SGX25C	224	138	2	1,944,576
EP1SGX25D	224	138	2	1,944,576
EP1SGX25F	224	138	2	1,944,576
EP1SGX40D	384	183	4	3,423,744
EP1SGX40G	384	183	4	3,423,744

The Cyclone M4K memory blocks have a maximum width of 36 bits and a maximum performance of approximately 200 MHz. Table 7 shows the number of Cyclone M4K memory blocks.

Table 7. Cyclone M4K Memory Blocks

Device	M4K RAM (128 × 36 Bits)
EP1C3	13
EP1C4	17
EP1C6	20
EP1C12	52
EP1C20	64

Table 8 shows the possible configurations of the M512, M4K, and M-RAM blocks found in Stratix, Stratix GX, and Cyclone devices.

Table 8. M512, M4K & M-RAM Memory Configurations (Part 1 of 2)

M512 RAM Block (32 × 18 Bits)	M4K RAM Block (128 × 36 Bits)	M-RAM Block (4K × 144 Bits)
512 × 1	4K × 1	64K × 8
256 × 2	2K × 2	64K × 9
128 × 4	1K × 4	32K × 16
64 × 8	512 × 8	32K × 18
64 × 9	512 × 9	16K × 32
32 × 16	256 × 16	16K × 36
32 × 18	256 × 18	8K × 64

Table 8. M512, M4K & M-RAM Memory Configurations (Part 2 of 2)		
M512 RAM Block (32 × 18 Bits)	**M4K RAM Block (128 × 36 Bits)**	**M-RAM Block (4K × 144 Bits)**
-	128 × 32	8K × 72
-	128 × 36	4K × 128
-	-	4K × 144

DSP Blocks

Stratix and Stratix GX devices contain dedicated DSP blocks for implementing high-speed multiplication functions within the FPGA. Tables 9 and 10 show the number of DSP blocks in Stratix and Stratix GX respectively.

Table 9. Number of DSP Blocks in Stratix Devices Note (1)

Device	DSP Blocks	9 × 9 Multipliers	18 × 18 Multipliers	36 × 36 Multipliers
EP1S10	6	48	24	6
EP1S20	10	80	40	10
EP1S25	10	80	40	10
EP1S30	12	96	48	12
EP1S40	14	112	56	14
EP1S60	18	144	72	18
EP1S80	22	176	88	22

Note to Table 9:

(1) Each device has either the number of 9 × 9-, 18 × 18-, or 36 × 36-bit multipliers shown. The total number of multipliers for each device is not the sum of all the multipliers.

Table 10. Number of DSP Blocks in Stratix GX Devices (Part 1 of 2) Note (1)

Device	DSP Blocks	9 × 9 Multipliers	18 × 18 Multipliers	36 × 36 Multipliers
EP1SGX10C	6	48	24	6
EP1SGX10D	6	48	24	6
EP1SGX25C	10	80	40	10
EP1SGX25D	10	80	40	10
EP1SGX25F	10	80	40	10
EP1SGX40D	14	112	56	14

Table 10. Number of DSP Blocks in Stratix GX Devices (Part 2 of 2) Note (1)

Device	DSP Blocks	9 × 9 Multipliers	18 × 18 Multipliers	36 × 36 Multipliers
EP1SGX40G	14	112	56	14

Note to Table 10:

(1) Each device has either the number of 9 × 9-, 18 × 18-, or 36 × 36-bit multipliers shown. The total number of multipliers for each device is not the sum of all the multipliers.

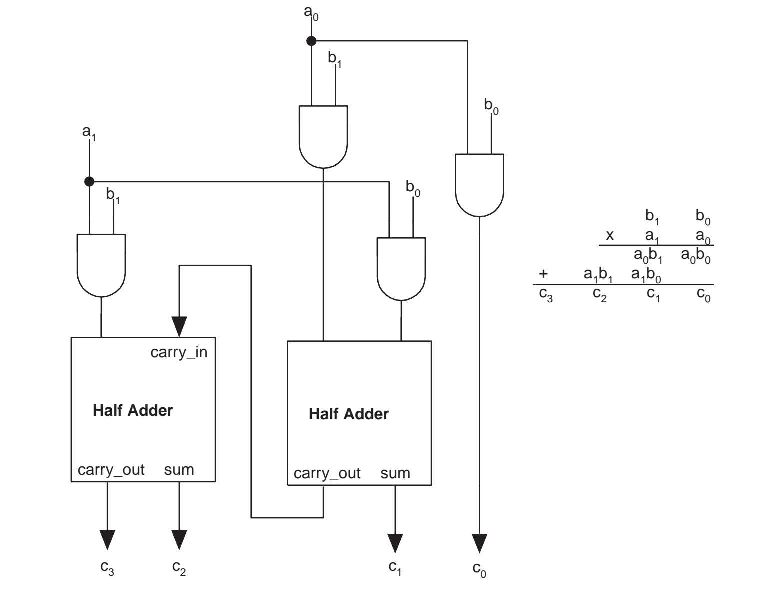
DSP Arithmetic Basics

DSP is a multiplication-intensive technology and to achieve high speeds, these multiplication operations must be accelerated. This section provides basic information on the mathematical theory and algorithms behind common DSP arithmetic implementations.

Multiplication

The base of many DSP algorithms is multiplication in which a multiplier is multiplied to a multiplicand. In this operation, each element of the multiplier is multiplied by each bit of the multiplicand. Then, the partial product of each multiplication is accumulated according to the weight of the partial product, where the weight indicates the location of a bit corresponding to other bits. For example, if a partial product of bits 4 through 7 is added to a partial product of bits 0 through 3, the partial product of 4 through 7 is shifted according to their weight and then accumulated to the partial product of previous stages. Figure 1 shows a simple 2 × 2 multiplication of multiplier a1a0 to multiplicand b1b0.

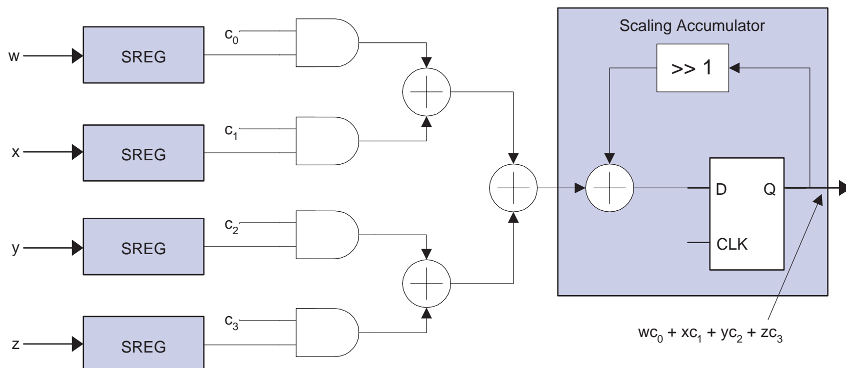
Figure 1. Multiplication of Two 2-Bit Numbers



Distributed Arithmetic

Distributed arithmetic is a method of performing multiplication by distributing the operation over many LUTs. Figure 2 shows a four-product MAC function that uses sequential shift and add to multiply four pairs, and then sums their partial product to obtain a final result. Each multiplier forms partial products by multiplying the multiplicand by one bit of the input data (multiplier) at a time, using an AND gate.

Figure 2. Distributed Arithmetic with Four Constant Multiplicands



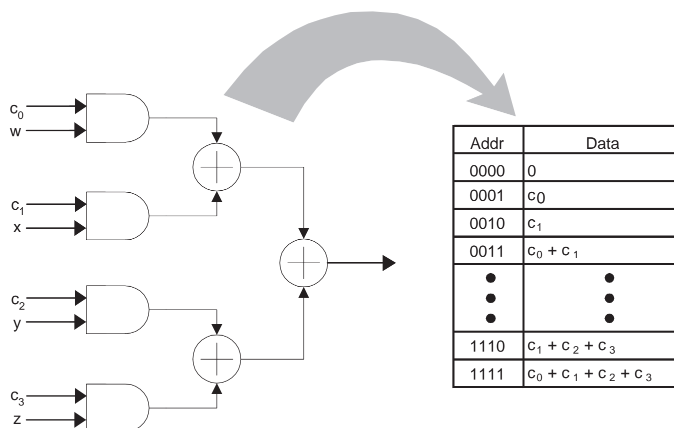
At the end of the process, each partial product result of each input bit is summed prior to the final scaling accumulator stage, which performs a shift-accumulate.

The distributed-arithmetic circuit simultaneously performs four multiplications and sums the results when all of the products are completed. The scaling accumulator shifts the sums of partial products according to the appropriate number of bits and accumulates the result to provide the final multiplier output.

Distributed Arithmetic in LUTs

Figure 3 shows how to implement distributed arithmetic using LUTs: the combined product and adder tree are reduced for the LUT implementation. In this example, the LUT contains the sums of constant coefficients for all possible input combinations to the LUT. The sums of the bits from the LUTs are added together in the scaling accumulator and shifted by the appropriate weights.

Figure 3. Four-Bit Multiplication with Constant Coefficients Note (1)



Note to Figure 3:

(1) c_0 to c_3 are constant coefficients.

The addressing method and data values stored in the LUT in Figure 3 apply to the sum of multiplication operation mode. The addressing method and LUT data values vary depending on the multiplier implementation mode.

Implementing Soft Multipliers Using Memory Blocks

You can use the Stratix and Stratix GX M512 or M4K RAM memory blocks and Cyclone M4K RAM memory blocks as LUTs to implement multiplication for DSP applications. Combinations of the coefficient results are pre-calculated and stored in the M512 or M4K RAM blocks as a LUT. The address port of the RAM block represents one of the multiplication operands. The content of the RAM block at each address represents a unique multiplication result calculated between the input operand and a known coefficient value based on the multiplier mode implemented.

The five soft multiplier modes supported by Stratix and Stratix GX devices are:

- *Parallel Multiplication*—Multiple memories produce one multiplication result every clock cycle. This mode is useful for high-speed data scaling.

- *Semi-Parallel Multiplication*—Each memory produces one multiplication with multi-cycle operation. This mode is useful for coefficient update of least mean squares (LMSs) and coefficient update of equalizers.

- *Sum of Multiplication*—One memory or group of memories produces the sum of multiplication results. This mode is useful in applications such as finite impulse response (FIR) filtering and discrete cosine transforms (DCTs).
- *Hybrid Multiplication*—Combination and optimization of semi-parallel and sum of multiplication modes of operation. This mode is ideal for a complex number of multiplications in complex fast Fourier transforms (FFTs) and infinite impulse response (IIR) filters.
- *Fully Variable Multiplication*—This mode is useful for a soft multiplier implementations in which both the input data and coefficients are varying. This mode is ideal for low-resolution multiplication functions.

The following sections describe each of these modes and provide examples.

Parallel Multiplication

Parallel multiplication involves multiplying all sections of a single input bus or multiplier value with a single multiplicand or coefficient and summing the partial product of each multiplication to obtain the final result. All of the input bits are parallel-loaded into the RAM block address port registers and a new multiplication is completed each clock cycle. For example, a 16-bit input bus can be separated into two groups of eight bits (one group of eight LSB bits and another group of eight MSB bits) and simultaneously shifted into the address ports of two RAM blocks. The output of the RAM blocks indicate the multiplication result for the particular set of bits with the coefficient. Figure 4 represents the decomposition of a 16-bit data input, 10-bit constant coefficient parallel multiplier.

Figure 4. Decomposition of a 16-Bit Input, 10-Bit Coefficient Parallel Multiplier

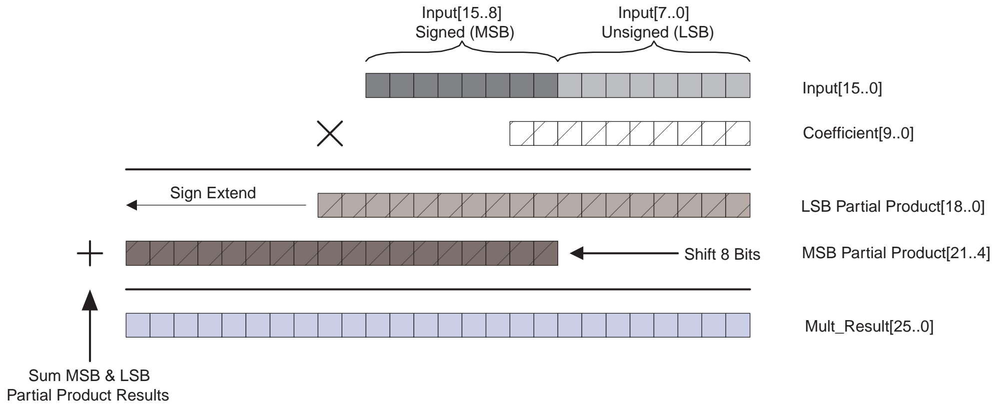
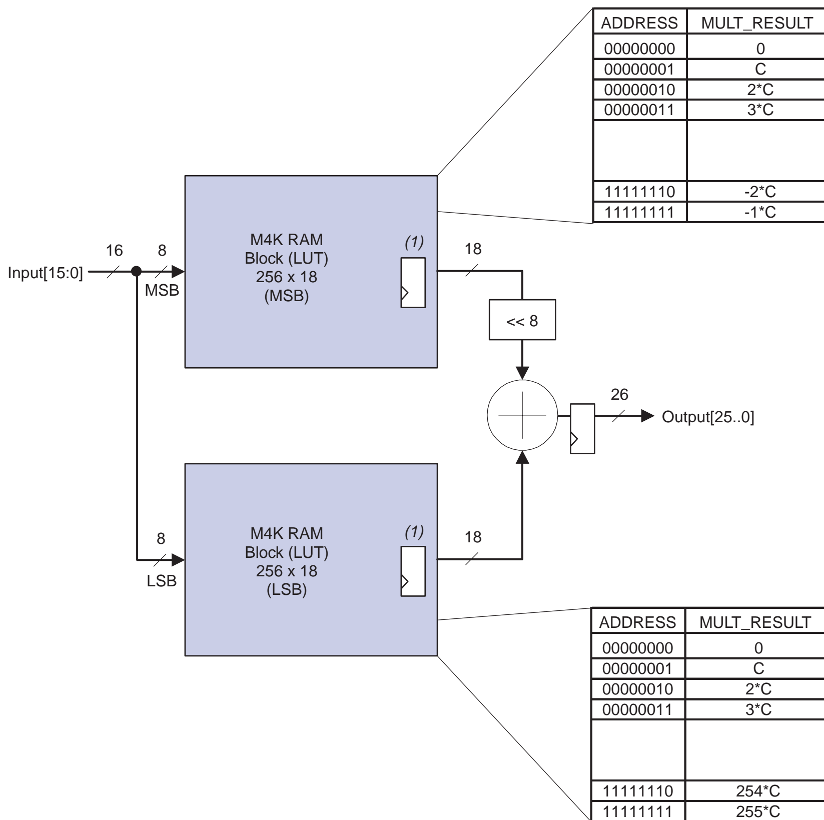


Figure 5 shows the RAM LUT implementation of the parallel multiplier decomposition shown in Figure 4. Because a parallel multiplier accepts a new input every clock cycle, this implementation takes three clock cycles (one to load the input values into the RAM block address ports and two pipeline delays) to compute the final multiplication result. New partial products are obtained from the RAM blocks every clock cycle and the partial products are summed according to their weights. Each partial product multiplication generates an output of 18 bits. At the end of the partial product accumulation, the multiplier generates a 26-bit output.

Figure 5. 16-Bit Input, 10-Bit Coefficient Parallel Multiplication Implementation Using M4K RAM Blocks as LUTs Note (1)



Note to Figure 5:

(1) Optional pipeline register to increase system performance.

Figure 5 shows an implementation for a 16-bit data input, split into two 8-bit sections implemented using two M4K RAM blocks, one for the MSB section and the other for the LSB section. For signed input buses, the M4K RAM block that accepts the MSB bits must contain precalculated coefficient values for signed inputs because the eight MSB bits that feed this RAM block are treated as signed values. The M4K RAM block that accepts the LSB bits must contain precalculated coefficient values for unsigned inputs because the eight LSB bits that feed this RAM blocks are unsigned values.

Because the size for M4K RAM blocks is 256 × 18 bits, the maximum number of bits per section for each M4K RAM block for this coefficient size is eight (2^8 = 256 addresses). The input bus and coefficient size directly affects the number and configuration of RAM blocks used to implement the multiplier. The parallel multiplication mode ensures maximum data throughput (i.e., a new data value every clock cycle).

You can also implement the parallel fixed-coefficient multiplier using the `altmemmult` Quartus II megafunction. You can use the MegaWizard® Plug-In Manager to customize the `altmemmult` megafunction to specify a parallel, fixed coefficient soft multiplier in your design. The input and coefficient bit width settings as well as RAM block selection type determine if the `altmemmult` function implements a semi-parallel or parallel mode soft multiplier, whichever is more efficient. Figures 6 and 7 show the appropriate settings required to implement the both the MSB and LSB M4K RAM blocks respectively, for the 16-bit input, 10-bit parallel multiplier example shown in Figure 14. The coefficient implemented in this example is a constant value of five.

Figure 6. altmemmult MegaWizard Settings for the MSB RAM Block 16-Bit Input, 10-Bit Constant Coefficient Parallel Multiplier

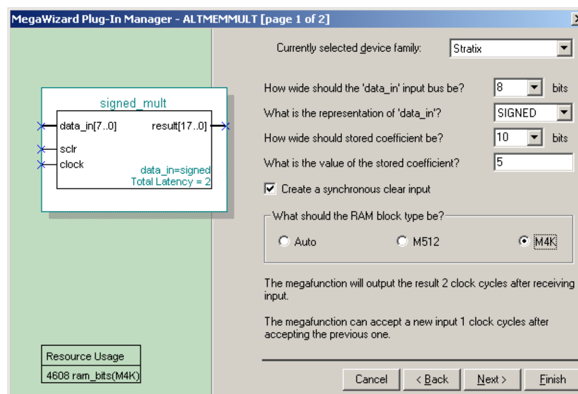
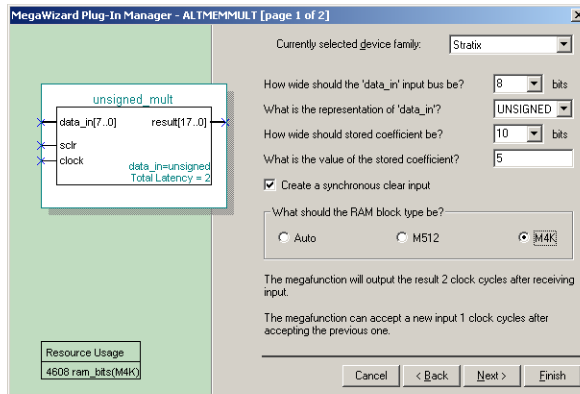


Figure 7. altmemmult MegaWizard Settings for the LSB RAM Block for a 16-Bit Input, 10-Bit Constant Coefficient Parallel Multiplier



The `sload_data` signal and the message located at the bottom right hand corner of the MegaWizard window indicates whether the `altmemmult` function chose to implement a semi-parallel or parallel mode soft multiplier. A parallel soft multiplier does not have the `sload_data` signal and the megafunction can accept a new input every clock cycle. The `altmemmult` megafunction can only implement small parallel mode soft multipliers (i.e., 8-bit input, 10-bit coefficient multipliers). Larger parallel multipliers require multiple `altmemmult` megafunctions to generate partial product results. To obtain the final multiplication result, these partial products must be summed in an end-stage adder implemented externally to the `altmemmult` function.

Fixed-Coefficient Multiplication

Figure 8 shows the simulation results for the example shown in Figure 5. This example multiplies the input, which has a decimal value of 297, with a coefficient, which has a value of 5.

Figure 8. Parallel Multiplication Simulation Results

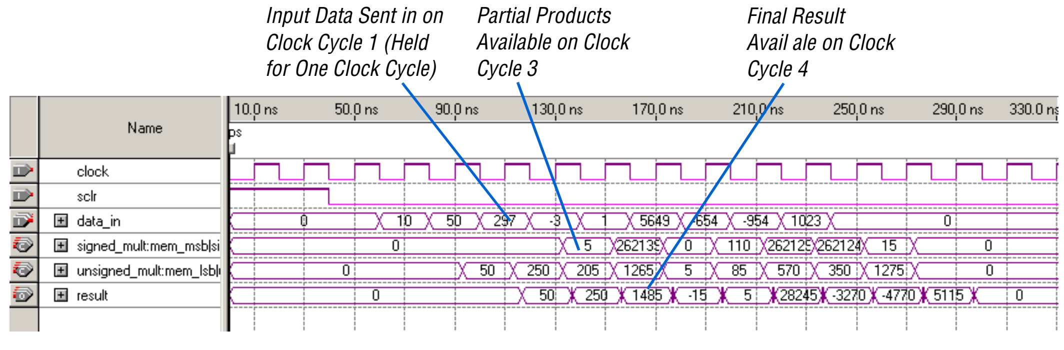


Table 11 shows the implementation result for the parallel fixed coefficient multiplication example shown in Figure 5. The example is implemented using the `altmemmult` megafunction.

Table 11. 16-Bit Input, 10-Bit Constant Coefficient Parallel Multiplication Implementation Results

Device	EP1S10F484C5
Utilization	Logic cells: 26/10,570 (1%) M4K RAM blocks: 2/60 (3%)
Latency *(1)*	3 clock cycles
Throughput	291 megasamples per second
Performance	291.0 MHz

Note to Table 11

(1) Latency is the number of clock cycles required to complete a single multiplication computation.

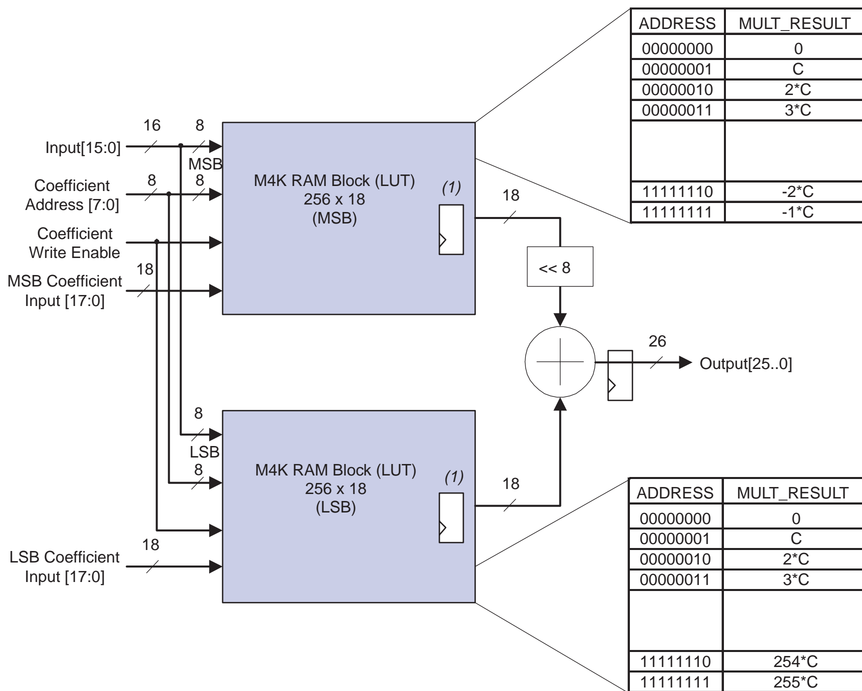
You can download the files (**parallel_fixed.zip**) for the design described in Table 11 from the Design Examples section of the Altera web site at www.altera.com.

Variable Coefficient Multiplication

To perform constant coefficient multiplication, you can implement the Stratix, Stratix GX, and Cyclone memory blocks as ROM. For variable coefficient multiplication, these memory blocks must be implemented as RAM blocks, which allow you to rewrite blocks with new precalculated coefficients. Figure 9 shows an implementation for variable coefficient parallel multiplication implementation using M4K single-port RAM blocks. Using the method shown in Figure 9, the multiplier function is

stalled while the coefficients are updated. But, by implementing multiple sets of RAM blocks for storing different precalculated coefficient sets, you can switch multiplication between two different sets of coefficients in a single clock cycle. One way of doing this is to partition the RAM block to store two unique sets of coefficients and to use the MSB address bit to select which coefficient set to use. Also, with the use of dual-port RAM blocks, you can write/update the values of a set of coefficients in a partition while simultaneously using a different set of coefficients in another partition to perform multiplication.

***Figure 9. 16-Bit Input, 10-Bit Variable Coefficient Parallel Multiplication Implementation Using M4K Single-Port RAM Blocks as LUTs** Note (1)*



Note to Figure 9:

(1) Optional pipeline register to increase system performance.

Table 12 shows the implementation results for a parallel variable coefficient multiplication example.

Table 12. 16-Bit Input, 14-Bit Variable Coefficient Parallel Multiplication Implementation Results

Device	EP1S10F484C5
Utilization	Logic cells: 43/10,570 (<1%) M4K RAM blocks: 2/60 (3%)
Latency *(1)*	3 clock cycles
Throughput	291 megasamples per second
Performance	291.0 MHz

Note to Table 12:
(1) Latency is the number of clock cycles required to complete a single multiplication computation.

You can download the files (**parallel_var.zip**) for the design described in Table 12 from the Design Examples section of the Altera web site at www.altera.com.

Semi-Parallel Multiplication

Semi-parallel multiplication involves multiplying sections of a single input bus or multiplier value with a single multiplicand or coefficient and shift accumulating the partial product of each multiplication to obtain the final result. For example, a 16-bit input bus can be separated into four groups of four bits that are consecutively shifted into the address port of the RAM block once every clock cycle, beginning with the first four LSB bits. The output of the RAM block indicates the multiplication result for a particular set of bits with the coefficient, every clock cycle. Figure 10 shows the decomposition of a 16-bit data input, 14-bit coefficient semi-parallel multiplier.

Figure 10. Decomposition of a 16-Bit Input, 14-Bit Coefficient Semi-Parallel Multiplier

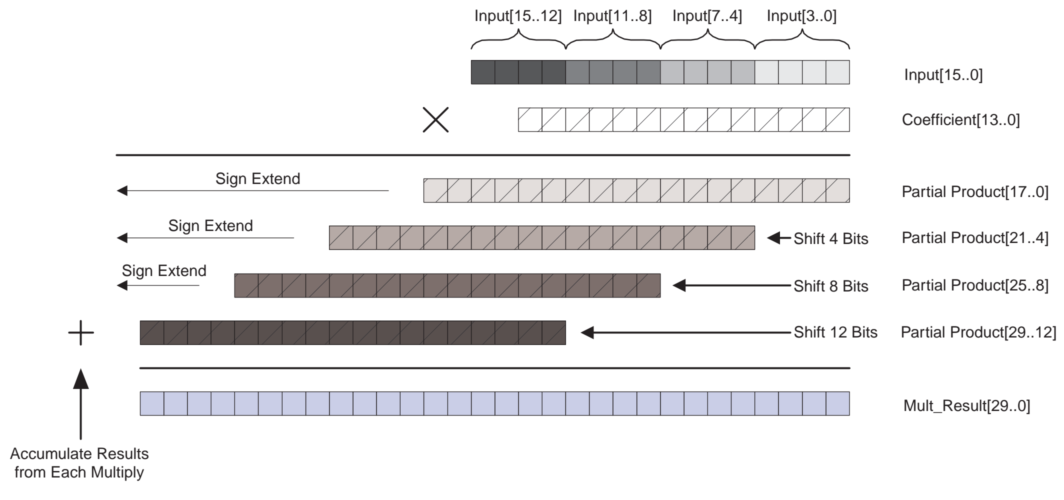
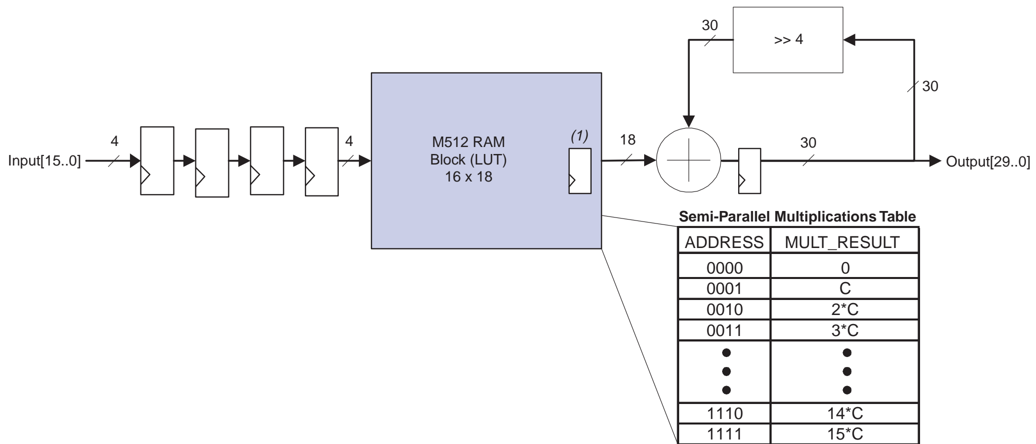


Figure 11 shows the RAM LUT implementation of the semi-parallel multiplier decomposition shown in Figure 10. This implementation loads the input data four bits every clock cycle, taking six clock cycles (four to load the input values into the RAM block plus two pipeline delays) to complete the multiplication operation by shift-accumulating the partial products obtained from the RAM block once per clock cycle, according to their weights. Each shift-accumulation of a partial product generates four extra bits. At the end of the fourth partial product accumulation, the multiplier generates a 30-bit output.

Figure 11. 16-Bit Input, 14-Bit Coefficient Semi-Parallel Multiplication Implementation Using M512 RAM Blocks as LUTs *Note (1)*



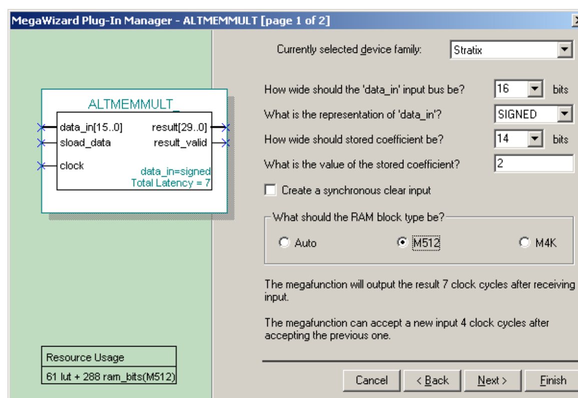
ADDRESS	MULT_RESULT
0000	0
0001	C
0010	2*C
0011	3*C
• • •	• • •
1110	14*C
1111	15*C

Note to Figure 11:
(1) Optional pipeline register to increase system performance.

Figure 11 shows an implementation for a 16-bit data input, split into four 4-bit sections implemented using a single M512 RAM block. In this example, for the same memory block utilization, factors like the input bus size help determine the output bit width and the latency of the multiplier. Increasing the bit width of the sections (i.e., implementing more than 4-bit sections in this case) can reduce the latency of the multiplier. This implementation may require more M512 RAM blocks or that you use M4K RAM blocks.

You can also implement the semi-parallel fixed coefficient multiplier using the `altmemmult` Quartus II megafunction. You can use the MegaWizard Plug-In Manager to customize the `altmemmult` megafunction to specify a semi-parallel, fixed coefficient soft multiplier in your design. The input and coefficient bit width settings as well as RAM block selection type determine whether the `altmemmult` function implements a semi-parallel or parallel mode soft multiplier; it implements whichever is more efficient. Figure 12 shows the settings required to implement the 16-bit input, 14-bit semi-parallel multiplier example shown in Figure 11. The coefficient implemented in this example is a constant value of two.

Figure 12. altmemmult MegaWizard Settings for a 16-Bit Input, 14-Bit Constant Coefficient Semi-Parallel Multiplier



The `sload_data` signal and the message located at the bottom right hand corner of the MegaWizard window indicate whether the `altmemmult` function chose to implement a semi-parallel or parallel mode soft multiplier. A semi-parallel soft multiplier has an `sload_data` signal and can only accept a new input after more than one clock cycle. The semi-parallel multiplier in Figure 11 indicates that the 16-bit input is split into four groups of four bits each. Because it takes four clock cycles to load the entire 16-bits into the RAM block, the current input must remain stable for four clock cycles prior to loading the new input. A high signal on `sload_data` for one clock cycle indicates the start of a new block of input data.

For information on implementing variable coefficient soft multipliers, refer to the "Variable Coefficient Multiplication" on page 15.

Figure 13 shows the simulation results for the example shown in Figure 11. This example multiplies the input, which has a decimal value of 10, with a coefficient, which has a value of 2.

Figure 13. Semi-Parallel Simulation Results

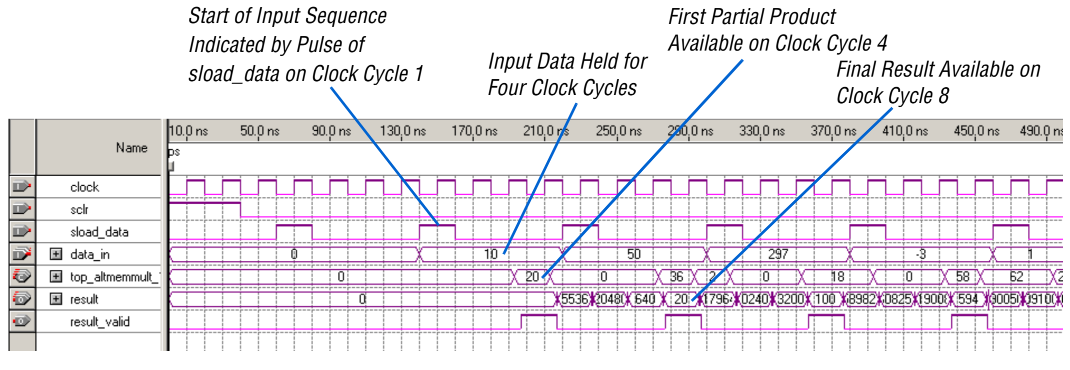


Table 13 shows the implementation result for the semi-parallel fixed coefficient multiplication example shown in Figure 11.

Table 13. 16-Bit Input, 14-Bit Constant Coefficient Semi-Parallel Multiplication Implementation Results	
Device	EP1S10F484C5
Utilization	Logic cells: 61/10,570 (1%) M512 RAM blocks: 1/94 (2%)
Latency *(1)*	7 clock cycles
Throughput	80 megasamples per second
Performance	321.0 MHz

Note to Table 13:

(1) Latency is the number of clock cycles required to complete a single multiplication computation.

You can download the files (**semi_prl_fixed.zip**) for the design described in Table 13 from the Design Examples section of the Altera web site at www.altera.com.

Table 14 shows the implementation results for a semi-parallel variable coefficient multiplication example.

Table 14. 16-Bit Input, 14-Bit Variable Coefficient Semi-Parallel Multiplication Implementation Results

Device	EP1S10F484C5
Utilization	Logic cells: 119/10,570 (1%) M512 RAM blocks: 1/94 (2%)
Latency (1)	7 clock cycles
Throughput	66 megasamples per second
Performance	265.0 MHz

Note to Table 14:
(1) Latency is the number of clock cycles required to complete a single multiplication computation.

You can download the files (**semi_prl_var.zip**) for the design described in Table 14 from the Design Examples section of the Altera web site at www.altera.com.

Sum of Multiplication

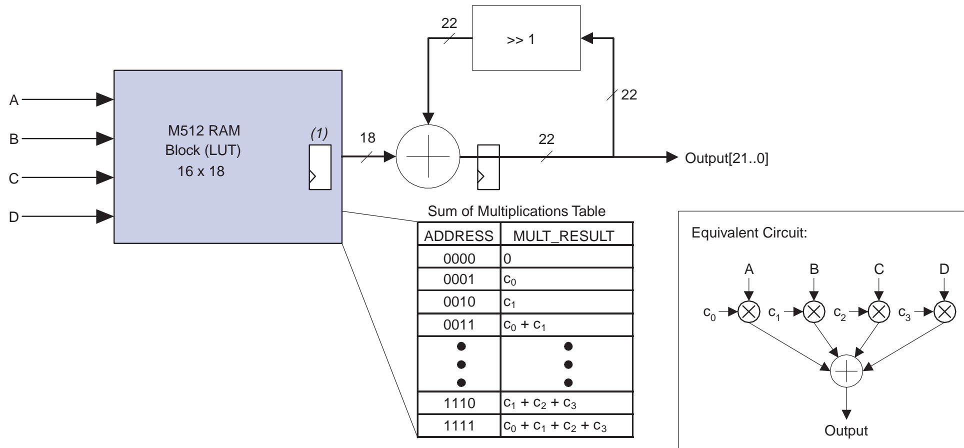
The sum of multiplication mode result is the weighted summation of results produced by multiplying a set of input data (multiplier) to a set of multiplicands. This sum forms the basis of a MAC function that is useful in functions such as FIR filters, where each input data (multiplier) value is multiplied with a particular coefficient (or multiplicand) and summed to provide the final result.

In the sum of multiplication mode, each input bus shifts into the address port of the memory block one bit per clock cycle, starting with the LSB. If there are four inputs (called A, B, C, and D) to the multiplier block, at the first clock cycle, the LSB of inputs A, B, C, and D forms the 4-bit address value to the RAM block. The next clock cycle, the second LSB bit for each input forms the next address value to the RAM block, and so on. For an n-bit input data width, it takes n clock cycles to load in all of the data bits required to compute the multiplication result. The RAM block output indicates the multiplication result for a specific bit position at each clock cycle.

Figure 14 shows the RAM LUT implementation of four 4-bit data inputs and up to 16-bit constant coefficients. This fixed coefficient implementation takes six clock cycles (four to load the input values into the RAM block plus two pipeline delays) to complete the multiplication operation by shift-accumulating the partial products obtained from the RAM block once per clock cycle, according to their weights. Each shift-accumulation of a partial product generates an extra carry bit. At the end

of the fourth partial product accumulation, the multiplier generates a 22-bit output. The size of the input data helps determine the output bit width and the latency of the multiplier.

Figure 14. 4-Input Sum of Multiplication Implementation Using M512 RAM Blocks as LUTs



Note to Figure 14:

(1) Optional pipeline register to increase system performance.

Figure 14 shows an implementation for four 4-bit data inputs. Because M512 RAM blocks are 32 × 18 bits, the maximum number of inputs for each M512 RAM block for this coefficient size is five ($2^5 = 32$ addresses). Depending on the number of inputs, size and number of coefficients, and the required operating speed, the number of RAM blocks used varies. The example shown in Figure 14 requires only one M512 RAM block.

For information on implementing variable coefficient soft multipliers, refer to "Variable Coefficient Multiplication" on page 15.

Figure 15 shows the simulation result for an example based on Figure 14. This example has additional pipeline stages and multiplies input A, which has a binary value of 0001, with the c0 coefficient, which has a value of -3.

You can choose to reduce the number of pipeline stages to reduce the latency, but your design may have reduced f_{MAX} as a result.

Figure 15. Sum of Multiplication Simulation Results

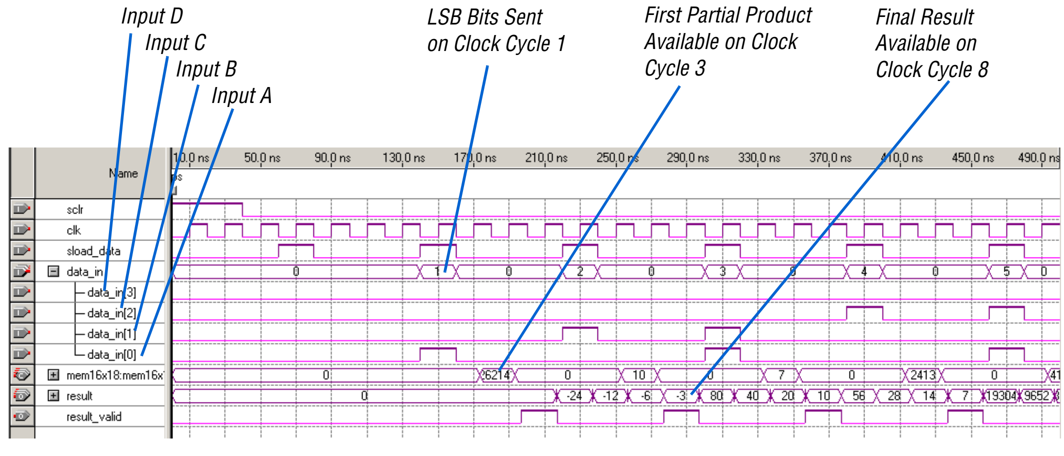


Table 15 shows the implementation results of the four input, 16-bit fixed coefficient sum of multiplication example shown in Figure 14.

Table 15. 4-Input, 16-Bit Fixed Coefficient Sum of Multiplication Implementation Results

Device	EP1S10F484C5
Utilization	Logic cells: 84/10,570 (1%) M512 RAM blocks: 1/94 (2%)
Latency *(1)*	7 clock cycles
Throughput	46 megasamples per second
Performance	184.0 MHz

Note to Table 15:

(1) Latency is the number of clock cycles required to complete an entire sum of multiplication computation.

You can download the files (**sum_mult_fixed.zip**) for the design described in Table 15 from the Design Examples section of the Altera web site at www.altera.com.

Table 16 shows the implementation results of a four input, 16-bit variable coefficient sum of multiplication example.

Table 16. Four Input, 16-Bit Variable Coefficient Sum of Multiplication Implementation Results

Device	EP1S10F484C5
Utilization	Logic cells: 113/10,570 (1%) M512 RAM blocks: 1/94 (1%)
Latency *(1)*	8 clock cycles
Throughput	29 megasamples per second
Performance	117.0 MHz

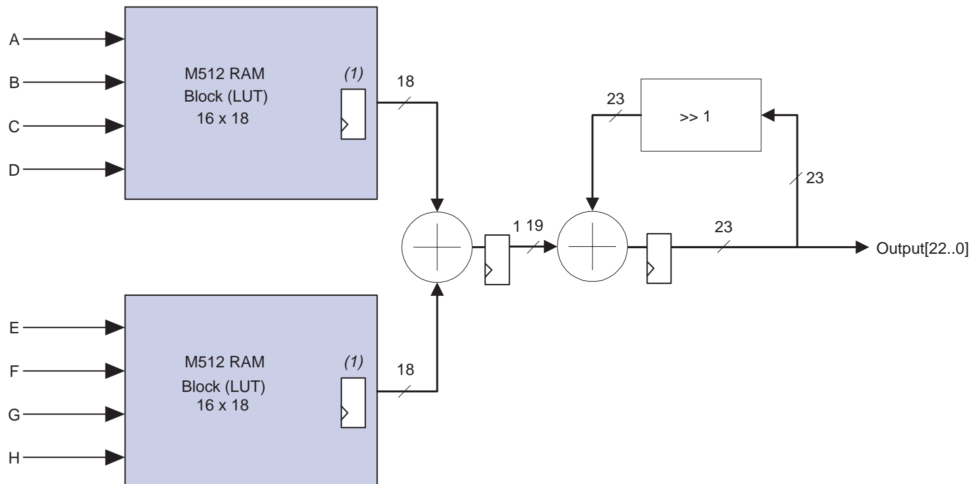
Note to Table 16:

(1) Latency is the number of clock cycles required to complete an entire sum of multiplication computation.

You can download the files (**sum_mult_var.zip**) for the design described in Table 16 from the Design Examples section of the Altera web site at www.altera.com.

You can combine multiple M512 blocks and/or M4K blocks to create larger multiplier structures that are capable of multiplying more data inputs and coefficients simultaneously. Figure 16 shows the multiplication of eight 4-bit data inputs to eight 16-bit constant coefficients in two M512 RAM blocks.

Figure 16. Using Multiple M512 RAM Blocks for an 8-Coefficient Multiplier



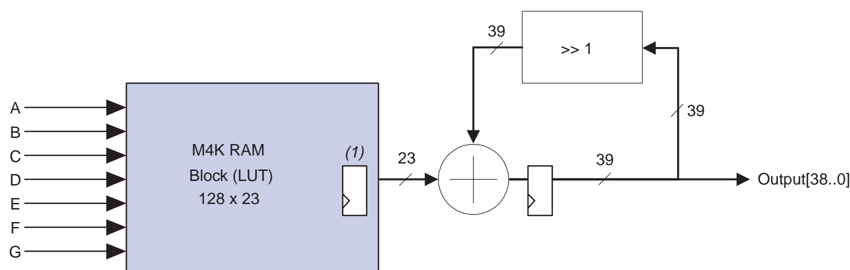
Note to Figure 16

(1) Optional pipeline register to increase system performance.

For information on implementing variable coefficient soft multipliers, refer to "Variable Coefficient Multiplication" on page 15.

You can also create similar implementations using M4K RAM blocks, particularly if the coefficients are larger than 16 bits. Figure 17 shows multiplication of seven 16-bit data inputs to a 20-bit constant coefficient in one M4K RAM block. The 128 addressed lines correspond to seven data inputs or unique coefficients in a M4K RAM block. Performing seven 16 × 20-bit multiplications generates a 23-bit output from a M4K RAM block. It takes 18 clock cycles to complete accumulation of the partial products (16 clock cycles to shift the input values into the address port of the RAM block plus two pipeline delays). After each partial product accumulation, one bit is added to the total number of output bits, making the final output 39 bits wide.

Figure 17. Using a M4K RAM Block for a 7-Coefficient Multiplier



Note to Figure 17:
(1) Optional pipeline register to increase system performance.

For information on implementing variable coefficient soft multipliers, refer to "Variable Coefficient Multiplication" on page 15.

Hybrid Multiplication

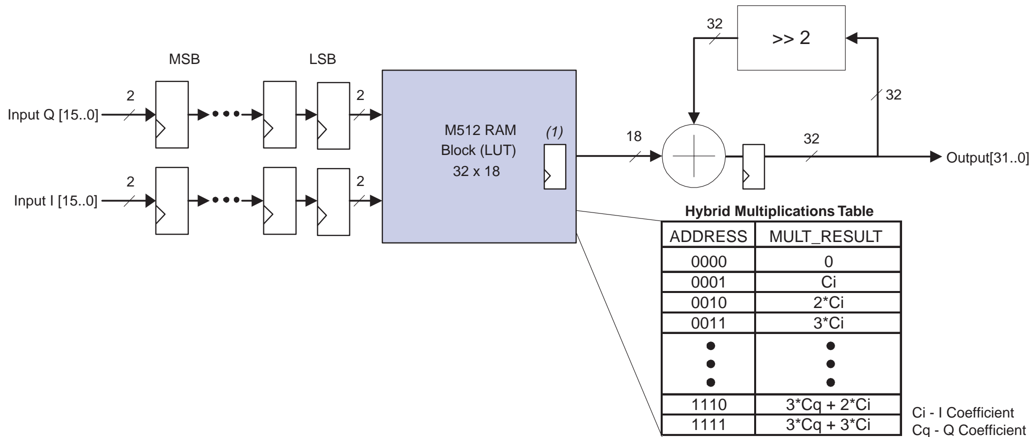
The hybrid multiplication mode is a combination of the semi-parallel and sum of multiplication modes where bit sections from two unique input streams are multiplied with two different coefficients values. This mode is useful in applications that require complex multiplication like fast Fourier transforms (FFTs) where each signal generally has a real and imaginary component that could be multiplied by two unique coefficient values. The partial products obtained from each bit section within the components are shift accumulated to obtain the final result.

In the hybrid multiplication mode, an equal number of bits from each input is concatenated and shifted into the address port of the RAM block every clock cycle, starting with the LSB. If the address port to the RAM

block is four bits wide, each input contributes two bits to the partial product calculation every clock cycle until the entire bit width of the inputs have completely shifted into the RAM block. In this case, for an input bus of 16-bits, it takes 8 clock cycles to shift in all of the data bits of that particular input. The output of the RAM block indicates the sum of multiplication result for a particular set of bits with the coefficients, every clock cycle.

Figure 18 shows the RAM LUT implementation of two 16-bit inputs, each labeled I Input and Q Input, respectively, and up to 15-bit constant coefficients. This implementation takes 11 clock cycles (eight to load the input values into the RAM block plus three pipeline delays) to complete the multiplication operation by shift-accumulating the partial products obtained from the RAM once per clock cycle, according to their weights. Each shift-accumulation of a partial product generates two extra bits. At the end of the last (eighth) partial product accumulation, the multiplier generates a 32-bit output. The size of the input data helps determine the output bit width and the latency of the multiplier.

Figure 18. Two-Input Hybrid Multiplication Implementation Using M512 RAM Blocks as LUTs



Note to Figure 18:

(1) Optional pipeline register to increase system performance.

Figure 18 shows an implementation for two 16-bit data inputs. Even though the 32 × 18-bit configured M512 RAM block can accept five address bits ($2^5 = 32$ addresses), the maximum number of bits equally contributed by each input is two bits (totaling four bits). In this example, for the same memory block utilization, factors such as the input bus size help determine the output bit width and the latency of the multiplier.

Increasing the number of M512 RAM blocks used or moving to larger memory blocks like M4K RAM blocks can reduce the latency of the multiplier an support larger coefficient bit widths.

For information on implementing variable coefficient soft multipliers, refer to "Variable Coefficient Multiplication" on page 15.

Figure 19 shows the simulation results for an example based on Figure 18. This example has additional pipeline stages and multiplies the I and Q inputs, which have values of 300 and 55, respectively, with coefficients Ci and Cq, which have values of 10 and 25, respectively (result = (input_I × Ci) + (input_Q × Cq) = (300 × 10) + (55 × 25) = 4375).

You can choose to reduce the number of pipeline stages to reduce the latency, but your design may have reduced $\mathbf{f_{MAX}}$ as a result.

Figure 19. Hybrid Multiplication Simulation Results

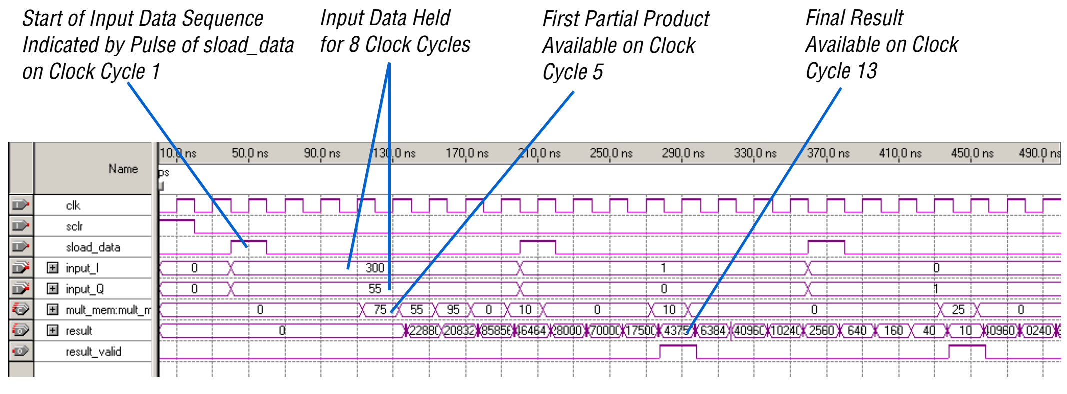


Table 17 shows the implementation results of the two 16-bit input, 15-bit constant coefficient hybrid multiplication example shown in Figure 18.

Table 17. Two Input, 15-Bit Constant Coefficient Hybrid Multiplication Implementation Results	
Device	EP1S10F484C5
Utilization	Logic cells: 185/10,570 (2%) M512 RAM blocks: 1/94 (1%)
Latency *(1)*	12 clock cycles
Throughput	22 megasamples per second
Performance	180.0 MHz

Note to Table 17:
(1) Latency is the number of clock cycles required to complete a single multiplication computation.

You can download the files (**hybrid_fixed.zip**) for the design described in Table 17 from the Design Examples section of the Altera web site at www.altera.com.

Table 18 shows the implementation results for a hybrid variable coefficient multiplication example.

Table 18. Two Input, 15-Bit Variable Coefficient Hybrid Multiplication Implementation Results	
Device	EP1S10F484C5
Utilization	Logic cells: 244/10,570 (2%) M512 RAM blocks: 1/94 (1%)
Latency *(1)*	12 clock cycles
Throughput	24 megasamples per second
Performance	188.0 MHz

Note to Table 18:
(1) Latency is the number of clock cycles required to complete a single multiplication computation.

You can download the files (**hybrid_var.zip**) for the design described in Table 18 from the Design Examples section of the Altera web site at www.altera.com.

Fully Variable Multipliers

The fully variable multiplier mode allows you to implement a soft multiplier in which both the input and the coefficient can vary every clock cycle. The partial product values, which are stored in the RAM blocks, are calculated based on the algebraic expansion of the following equation:

$$(a + b)^2 - (a - b)^2 = a^2 + 2ab + b^2 - (a^2 - 2ab + b^2)$$

$$= 4ab$$

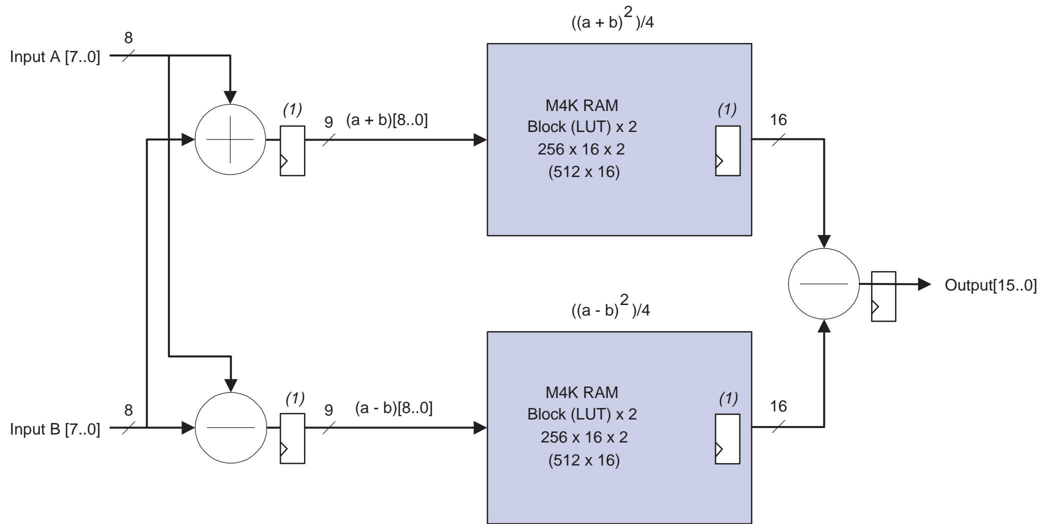
therefore:

$$ab = ((a + b)^2 / 4) - ((a - b)^2 / 4)$$

Where a and b are both variable inputs to the multiplier

Figure 20 shows the RAM LUT implementation of the fully variable multiplier calculated using these equations. Two unique RAM blocks are required, to store the $(a + b)^2/4$ and $(a - b)^2/4$ precalculated values, respectively. The address inputs of $(a + b)$ for the former and $(a - b)$ for the latter RAM block are precalculated in logic prior to the RAM block. The final result of the multiplication is obtained by subtracting the result of the $(a - b)$ RAM block by the result from the $(a + b)$ RAM block. The fully variable multiplier can accept a new input every clock cycle, and takes three clock cycles to compute the final multiplication result.

Figure 20. 8-Bit Fully Variable Multiplier Implementation Using M4K RAM Blocks as LUTs



Note to Figure 20:

(1) Optional pipeline register to increase system performance.

Figure 20 shows an implementation for two 8-bit data inputs. 8-bit inputs result in 16-bit outputs and 9-bit addresses per partial product RAM block. Therefore, for each partial product, two M4K RAM blocks are required in a 256×16 configuration ($2^9 = 512$ addresses). In this multiplier mode, the size of the inputs directly affects the total number of RAM blocks required.

Figure 21 shows the simulation results for the example shown in Figure 20.

Figure 21. Fully Variable Multiplier Simulation Results

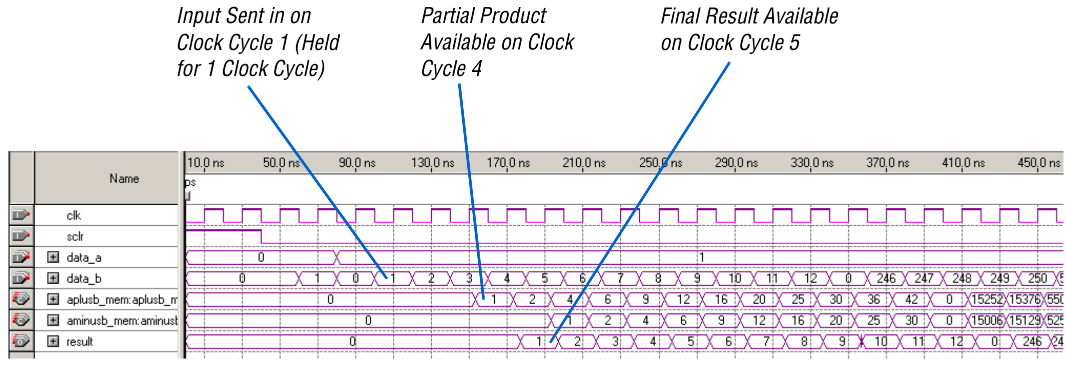


Table 19 shows the implementation results of the 8-bit fully variable multiplier example shown in Figure 20. The fully variable multiplication mode is ideal for low-resolution multiplication in which the input and coefficient bit widths are not too large. Larger input and coefficient bit widths require a significant amount of memory block resources compared to other variable soft multiplier modes of the same size.

Table 19. 8-Bit Fully Variable Multiplier Implementation Results

Device	EP1S10F484C5
Utilization	Logic cells: 35/10,570 (1%) M4K RAM blocks: 4/60 (6%)
Latency *(1)*	4 clock cycles
Throughput	291 megasamples per second
Performance	291.0 MHz

Note to Table 19:

(1) Latency is the number of clock cycles required to complete a single multiplication computation.

You can download the files (**fully_var.zip**) for the design described in Table 19 from the Design Examples section of the Altera web site at www.altera.com.

Implementing Multipliers Using DSP Blocks or LEs

Altera provides three Quartus II megafunctions for implementing various multiply, multiply-accumulate, and multiply-add functions using DSP blocks or LEs:

- lpm_mult—Performs multiply functions only
- altmult_add—Performs multiply or multiply-add functions
- altmult_accum—Performs multiply-accumulate functions only

For more information on using these megafunctions to implement multipliers, refer to *AN 214: Using the DSP Blocks in Stratix & Stratix GX Devices*.

Firm Multipliers

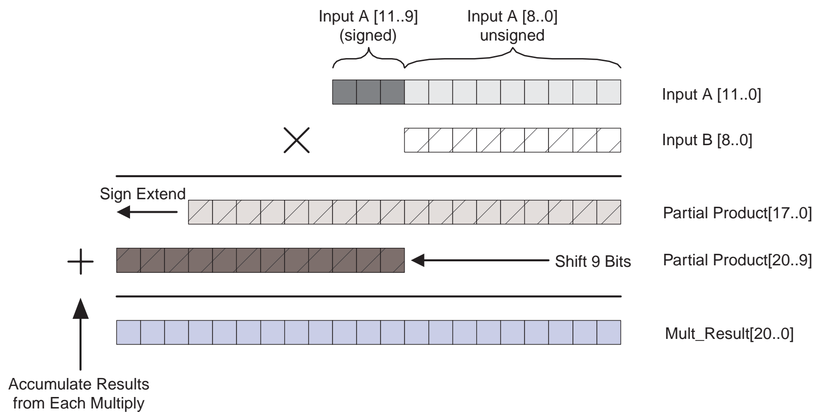
Firm multipliers use a combination of DSP blocks and LEs, enabling you to increase the utilization efficiency of the DSP blocks within your Stratix or Stratix GX device. Stratix and Stratix GX DSP blocks support 9 × 9, 18 × 18, and 36 × 36 multipliers. If you implement a multiplier of a different size, some DSP blocks may be partially used. For example, a 12 × 9 multiplier uses two 9 × 9 DSP blocks because the 12-bit input exceeds the maximum requirement of a single 9 × 9 multiplier. The first 9 × 9 DSP block is fully utilized but the second 9 × 9 multiplier is partially used. Instead of using the partially utilized DSP block for the remaining logic, you can use a firm multiplier to implement it, freeing the DSP block for other use. This method is particularly useful if your design requires a lot of DSP blocks but has LE resources available.

To implement a firm 12 × 9 multiplier, split up the 12-bit input and decompose the multiplication into smaller, partial products that can be implemented in DSP blocks and LEs. To maximize DSP block usage, split the 12-bit input into two sections: a 9-bit section that is multiplied using the DSP blocks and a 3-bit section that is multiplied using LEs. If the 9-bit section consists of LSBs, it becomes an unsigned value while the 3-bit section becomes a signed value and vice versa.

When deciding whether to select the 3-bit section from the MSB or the LSB of the 12-bit input, keep in mind that an LE multiplier is more resource efficient when implemented as a signed multiplier than as an unsigned multiplier. If the 9-bit input is unsigned, the 3-bit section is chosen from the MSB so that the LE multiplier performs signed multiplication. If the 9-bit input is signed, you can choose the 3-bit section from the MSB or LSB because either implementation results in a signed multiplier implemented in LEs.

Figure 22 shows the decomposition of the 12 × 9 firm multiplier.

Figure 22. Decomposition of the 12 × 9 Multiplier



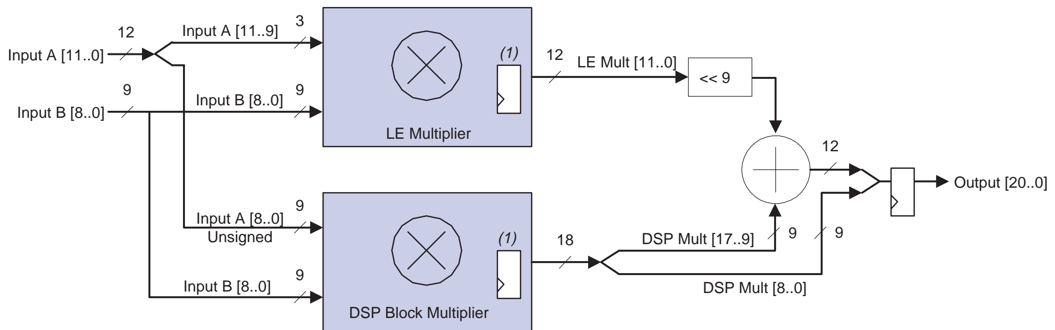
Based on this decomposition, you can build the circuit for the firm multiplier using three main blocks:

- DSP block multiplier—Built using either the `lpm_mult` or `altmult_add` megafunctions
- LE-based multiplier—Built using either the `lpm_mult` or `altmult_add` megafunctions
- End-stage adder—Built using the `lpm_add_sub` megafunction

The DSP block multiplier multiplies the 9-bit input by the 9-bit LSB section of the 12-bit input. The LE-based multiplier multiplies the 9-bit input with the 3-bit MSB section of the 12-bit input. The result of both multipliers is the partial products of the decomposition. The results of the partial products are weighted prior to being summed in the end-stage adder. This weighting and addition restores the bit-alignment of the partial products to ensure proper result values. Based on Figure 22, the 9 × 3 multiplication partial product is weighted by a shift to the left of nine bits. The 12-bit end-stage adder has to accommodate the 12-bit result of the 9 × 3 multiplication and the nine MSBs of the 9 × 9 multiplication, sign extended.

Figure 23 shows the circuit of the 12 × 9 firm multiplier.

Figure 23. 12 × 9 Firm Multiplier Circuit



Notes to Figure 23:

(1) Optional pipeline register to increase system performance.

(2) Using the `altmult_add` megafunction to implement the multipliers allows you to mix signed and unsigned inputs.

Figure 24 shows the simulation results for the example shown in Figure 23.

Figure 24. 12 × 9 Firm Multiplier Simulation Results

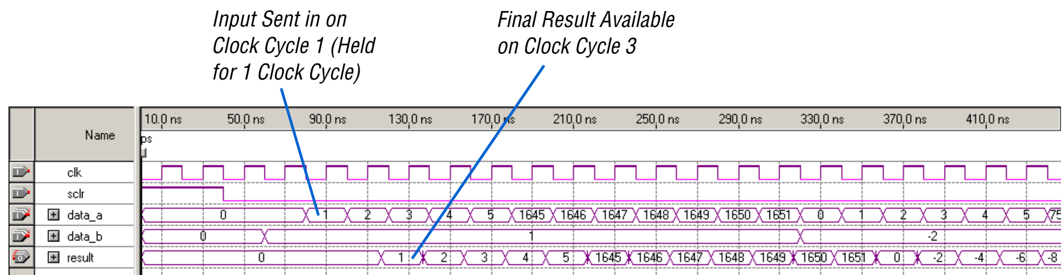


Table 20 shows the implementation results for the 12 × 9 firm multiplier circuit example shown in Figure 23.

Table 20. 12 × 9 Firm Multiplier Implementation Results Note (2)

Device	EP1S10F484C5
Utilization	Logic cells: 68/10,570 (1%) DSP block 9-bit elements: 1/48 (2%)
Latency *(1)*	2 clock cycles
Throughput	270 megasamples per second
Performance	270.0 MHz

Note to Table 20:

(1) Latency is the number of clock cycles required to complete a single multiplication computation

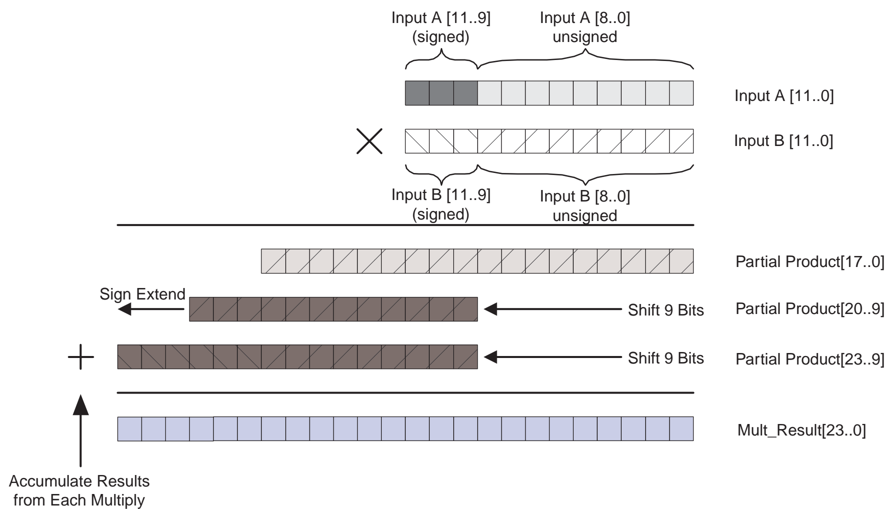
(2) The `altmult_add` megafunction implements both the LE and DSP block multipliers.

You can download the files (**12x9_firm_mult.zip**) for the design described in Table 20 from the Design Examples section of the Altera web site at www.altera.com.

The example shown in Figure 23 is suitable when only one of the multiplier inputs exceeds the 9-bit input width of a single DSP block. When both multiplier inputs exceed 9-bits, as in the case of a 12 × 12 multiplier, the multiplication must be decomposed into three partial products instead of two. The 12-bit inputs must be sectioned to maximize the use of the 9 × 9 DSP blocks and the utilization efficiency of implementing signed multiplication in LEs. Therefore, both inputs should be sectioned into a 3-bit MSB section and a 9-bit LSB section.

Figure 25 shows the decomposition of the 12 × 12 multiplier.

Figure 25. Decomposition of the 12 × 12 Multiplier



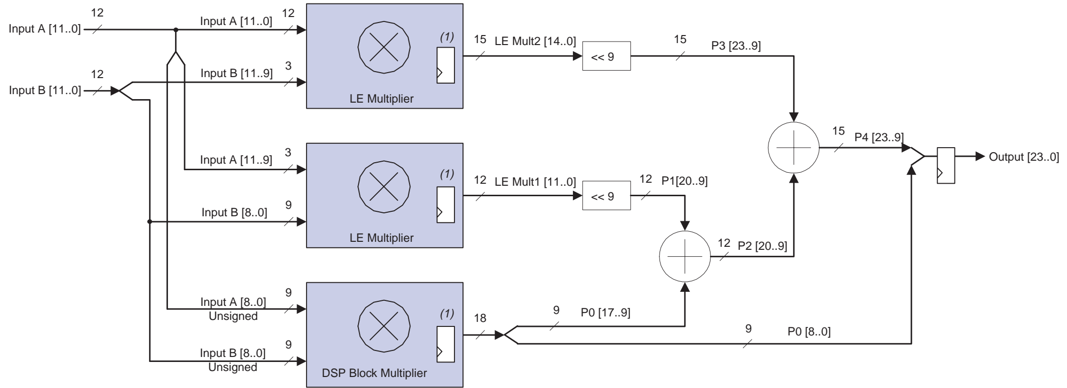
The circuit for the firm multiplier can now be extracted from the decomposition. The firm multiplier circuit consists of five main blocks:

- One DSP block multiplier—Built using either the `lpm_mult` or `altmult_add` megafunctions
- Two LE-based multipliers—Built using either the `lpm_mult` or `altmult_add` megafunctions
- Two adders—Built using the `lpm_add_sub` megafunction

The DSP block multiplier multiplies the two 9-bit LSB sections of the 12-bit inputs. The first LE-based multiplier multiplies the 9-bit LSB section of one 12-bit input with the 3-bit MSB section of the other 12-bit input. The other LE-based multiplier multiplies the 3-bit MSB of one 12-bit input with the entire 12-bits of the other input. The results of these three multipliers are the three partial products of the decomposition. The results of these partial products are summed in two stages (using two adders) prior to producing the final output.

Figure 26 shows the two adder stages within the final circuit of the 12 × 12 firm multiplier.

Figure 26. 12 × 12 Firm Multiplier Circuit



Notes to Figure 26:

(1) Optional pipeline register to increase system performance.

(2) Using the `altmult_add` megafunction to implement the multipliers allows you to mix signed and unsigned inputs.

Figure 27 shows the simulation results for the example shown in Figure 26.

Figure 27. 12 × 12 Firm Multiplier Simulation Results

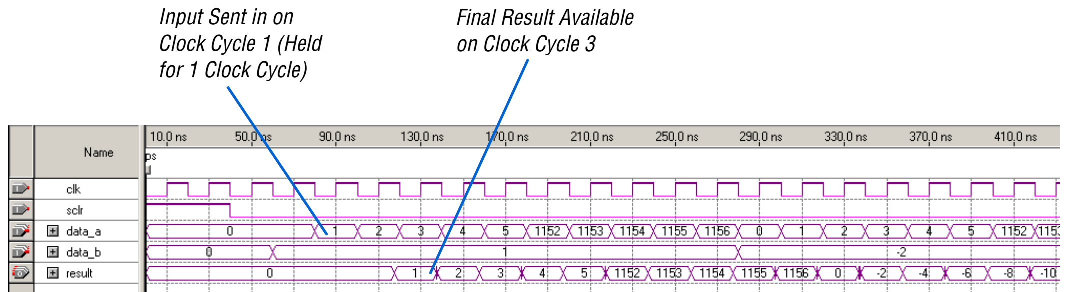


Table 21 shows the implementation results for the 12 × 12 firm multiplier example shown in Figure 26.

Table 21. 12 × 12Firm Multiplier Implementation Results *Note (2)*

Device	EP1S10F484C5
Utilization	Logic cells: 145/10,570 (1%) DSP block 9-bit elements: 1/48 (2%)
Latency *(1)*	2 clock cycles
Throughput	181 megasamples per second
Performance	181.0 MHz

Note to Table 21:

(1) Latency is the number of clock cycles required to complete a single multiplication computation

(2) The `altmult_add` megafunction implements both the LE and DSP block multipliers.

You can download the files (**12x12_firm_mult.zip**) for the design described in Table 20 from the Design Examples section of the Altera web site at www.altera.com.

Conclusion

Although Stratix and Stratix GX DSP blocks are useful for implementing DSP applications, you can also use Stratix and Stratix GX TriMatrix blocks (M512 or M4K RAM blocks) or Cyclone M4K RAM blocks for designs that need more multipliers than are available using DSP blocks alone. For example, using soft multipliers, you can increase the number of 16 × 16 multipliers in a Stratix E1S80 device by a factor of more than 7 see Table 9 on page 5). Another example, the fully variable soft multiplier is an ideal implementation for applications requiring smaller multipliers with frequently varying coefficients. Other soft multiplier modes are more resource efficient and better suited for applications that do not require frequent coefficient updates. The firm multiplier allows you to balance the use of DSP block multipliers with LE-based multipliers, allowing more efficient use of the Stratix and Stratix GX DSP blocks.

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
www.altera.com
Applications Hotline:
(800) 800-EPLD
Literature Services:
lit_req@altera.com

Copyright © 2003 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.