Fast Integer Multipliers

This application example was prepared by Ken Chapman, a Xilinx Field Applications Engineer based in England. An abbreviated version appeared in EDN Magazine's Design Ideas column in March, 1993, and was recently chosen the overall 1993 Design Idea winner. Congratulations, Ken!

Digital multipliers are needed in many system applications, including digital filters, correlators, and neural networks. These multipliers typically are required to handle operands of up to 16 bits, and need to provide results in less than 50 ns (20 MHz systems). Figure 1 is a common module found in many signal processing applications. (For example, with the output 'y' applied to an activation function, this would be a simple neuron.)

The increasing density of programmable logic devices has led to complete systems and sub-systems being implemented in one FPGA device. However, digital multipliers generally are considered too slow when implemented in FPGAs, or too large to make effective use of a programmable part. As an alternative, dedicated multiplier devices are connected to the main system, often resulting in a performance degradation caused by the delays inherent in getting the data to and from the multiplier device. (Often, these data movements are time-multiplexed to reduce the large I/O requirement.) Thus, techniques for implementing a compact and fast digital multiplier in a programmable part are needed.

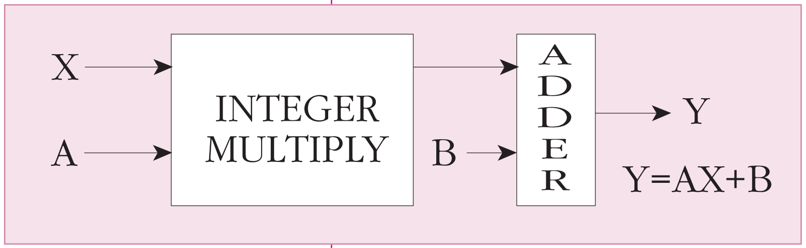


Figure 1

Implementation Techniques

There are three main implementations of digital multipliers:

1) **Shift and Add** — One operand is shifted to the left by one bit each cycle and applied to an accumulator when the corresponding bit in the second operand is high.
2) **Look-up table** — The operands are applied as addresses to a pre-programmed memory that outputs the result.
3) **Logical tree** — Each of the resultant bits are a logic function of the relevant bits of each operand.

Using a 16-bit x 16-bit multiplier as an example, let's consider each technique.

The **shift and add** implementation is compact but very slow. The result is obtained after 16 clock cycles and the accumulator must be 32-bits wide. The accumulator will determine the maximum clock rate dependent on the carry logic chain. This implementation also precludes any new operands from being applied until the calculation has been completed, read, and cleared.

The speed of the **look-up table** solution depends on the speed of the memory used, but rapidly becomes unwieldy as operand size increases. This 16x16 example requires a 4,294,967,296 x 32-bit memory! Small multipliers work well this way, such as a 4-bit x 4-bit multiply implemented in a byte-wide ROM.

Some very complex implementations of **logical trees** have been developed employing product sharing, and are to be found in many dedicated arithmetic devices. The gate count is high and can be considered a reduced version of the ROM table. The logic involved tends to have high fan-in requirements (up to 32 inputs).

The XC4000 FPGA architecture includes the types of logic resources needed for digital multipliers:

- High-density devices are available (currently, up to 25,000 gates), providing ample density for a multiplier and substantial additional logic.
- Fast, dedicated carry logic circuitry provides for compact and high-speed simple arithmetic functions such as addition.
- The XC4000 architecture includes the ability to configure custom blocks of RAM and ROM using the look-up-table-based function generators.

Having all these features available means that any of the described techniques may be implemented.

Fast and Compact Multipliers

If the multiplier needs to be both fast and compact, the choice of multiplier implementations can be difficult.

The compact **shift and add** technique may be too slow for many applications, but is easily implemented in the XC4000 architecture. The accumulator can make excellent use of the fast carry logic, with a reasonably low cycle time (even for a result of 32 bits).

A **look-up table** built from a custom ROM block is ideal for small multipliers. Since any configurable logic block (CLB) can be configured as either two 16 x 1 memories or a single 32 x 1 memory, it is easy to see how small look-up tables can be implemented. Unfortunately, the larger memory requirements of larger operands can use up the available CLB resources rapidly. Operands with more than four bits soon become difficult to handle.

Logic trees can be implemented, but again, as the operand width increases, the fan-in requirement of the logic soon exceeds the nine inputs available to a CLB. As a result, functions have to be split. This prevents a compact design, and impacts performance due to the multiple block delays between the operands and the result. However, the XC4000 device architecture does offer an easy way to pipeline any design, using the two registers in each CLB. Once again, operands of more than four bits become an escalating problem to handle.

“The real strengths of the XC4000 FPGA architecture in this application are its ability to handle small look-up tables, provide logic functions of less than nine inputs, and form fast adders of any size.”

The real strengths of the XC4000 FPGA architecture in this application are its ability to handle small look-up tables, provide logic functions of less than nine inputs, and form fast adders of any size. Therefore, the ideal solution may be to use a hybrid technique tailored to these properties: using small look-up tables for partial products and combining the results by addition.

Anatomy of a Hybrid Multiplier

Partial products are formed by splitting the first operand into sections and multiplying each section by all of the second operand. These products contain all the information about the multiplication that must then be combined by addition, while at the same time restoring the bit weighting of the sections from the first operand.

Figure 2 is a block diagram of the implementation of an 8 x 8 multiplier that multiplies the two operands called ‘X_OPERAND’ and ‘A_OPERAND’. (X-BLOX modules could be used to enter this design easily in a block diagram format.) The first operand is split into two nibbles. Both nibbles produce a 12-bit partial product after multiplication by the second operand. These products are applied to a

Continued on the next page

Continued from the previous page

16-bit combining adder to form the final 16-bit result.

To restore the weighting factor of the two nibbles, the 12-bit partial products are expanded into 16-bit operands. The product from the low order nibble, with a weighting of one, is converted to 16 bits by padding four additional MSBs with the value of zero. The product from the high order nibble, with a weighting of 16, is effectively shifted by four bits using four additional LSBs, again with value of zero. The resulting 16-bit operands form the inputs to the adder.

It is not possible for the adder to overflow because it is known that an integer multiply of two operands with ‘n’ and ‘m’ bits (respectively) will generate an ‘n+m’ bit result. In this 8-bit x 8-bit example, the maximum result occurs when multiplying FF x FF = FE01 (hex).

To create a compact, fast and easy-to-implement design, make the look-up tables for the partial products as small as possible. The simple 16 x 1 memory elements are combined to expand the width of the memory to that required for the partial products. In this example, 16 x 12 memories are synthesized using just six CLBs, each containing two 16 x 1 tables.

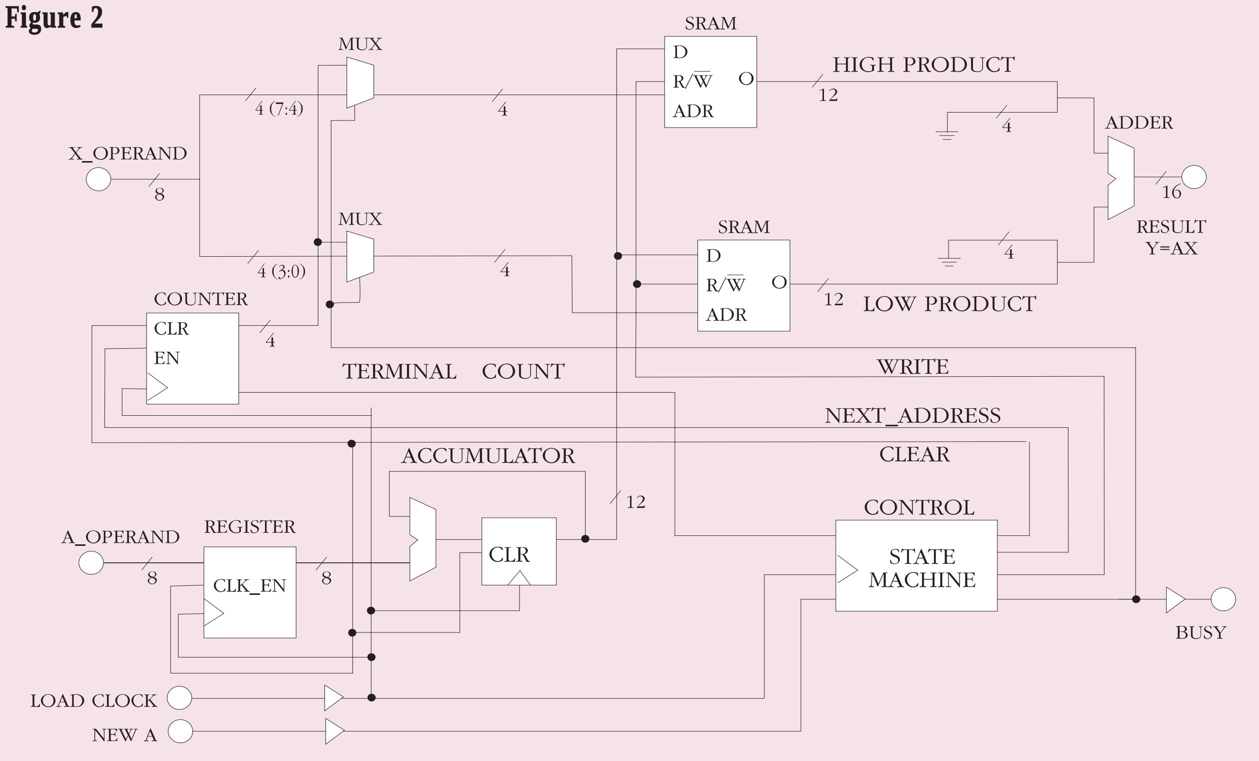
A memory with 16 addresses has 4 address lines, meaning that only the nibble section from the X_OPERAND can be applied. The need to connect the A_OPERAND has been totally eliminated in this design by ensuring that only those partial products that can be obtained by a single value of A_OPERAND are available at any one time. In other words, at any given time, the X_OPERAND can only be multiplied by a constant, as determined by the contents of the look-up tables.

By preloading the look-up tables with only those partial products that can be obtained from the present value of A_OPERAND, the size of the look-up table is greatly reduced. These partial products in the look-up table are, quite simply:

0, 1 x A_OPERAND, 2 x A_OPERAND, ... , 15 x A_OPERAND

These 16 partial products must be recalculated and reloaded for each new value of A_OPERAND. Building the look-

Figure 2



up tables from RAM means that the 16 partial products relating to a given value of A_OPERAND can be modified. This is performed using an accumulator, address counter and small state machine that accesses each location of the memory and stores a new product. Both tables are modified to contain the same data, so each nibble of the X_OPERAND is multiplied by the same value.

Clearly, the multiplier can not be used for the period during which the table contents are being modified, and results cannot be obtained quickly when changing the value of A_OPERAND. In contrast, changes in the X_OPERAND will give a result with minimal delay, and the data path can easily be pipelined to optimize performance in a clocked system.

Most applications can tolerate the time spent modifying the look-up tables for a new value of the A_OPERAND. Consider the module of Figure 1. In many cases, only the ability to modify 'A' is required in order to tune a system. Another example is video processing, where 'A' is changed only at the end of each page, during the screen fly-back.

For some applications, 'A' will be fixed from the design concept and throughout the life of the system. (Computer simulations may provide the optimum values for multiplicands and offsets in the system.) In these cases, the look-up tables can be implemented as ROMs in the XC4000 with the partial products pre-defined, thereby eliminating the overhead logic needed to program the tables and increasing performance by removing the address multiplexers from the data path. The reconfigurable nature of XC4000 devices would still permit different contents to be loaded into the look-up tables (representing different values for A_OPERAND) by using multiple device configuration bitstreams.

The technique expands easily by units of four bits on the X_OPERAND, where each additional nibble connects to a separate look-up table. The size and contents of each table are determined by the second operand (A_OPERAND). Further levels of combining adders are required for more than two look-up tables. These will impact performance, but their effect can be minimized with a pipelined design (registers at the table outputs and each adder). Negative values in two's-complement format can be converted using simple 'invert and add 1' pipeline stages where necessary, handling the sign bits separately.

For example, a 16 x 16 multiplier of this type uses four tables to generate 20-bit partial products corresponding to each nibble. Two 24-bit combining adders are used to form partial products for the high and low order bytes of the input data. A final 32-bit adder combines these products, restoring the weighting of 256 to the high order product (a shift of 8 bits).

Performance

The figures shown in Table 1 are worst case for an XC4000-5 device. The designs were each processed using the Xilinx automatic tools without user intervention.

The combinatorial speed is taken as a device pin-to-pin delay, including 10 nanoseconds of I/O delays. Since the multiplier is generally imbedded in the system, these time values may be deducted in estimating in-system performance.

Pipelined speed is determined by the speed of the slowest element, which is the largest adder in the system. ◆

Style	**8 x 8-bit RAM Look-up**	**8 x 8-bit ROM Look-up**	**16 x 16-bit RAM Look-up**	**16 x 16-bit ROM Look-up**
CLB Count	39	22	117	84
Combinatorial Delay	56 ns	46 ns	96 ns	88 ns
Pipelined Performance	39.9 MHz	41.3 MHz	25 MHz	25.5 MHz

Table 1: Performance of hybrid multipliers implemented in an XC4000-5 FPGA