WP277 (v1.0) May 22, 2008

Expanding Dedicated Multipliers

By: Ken Chapman

Those of you with DSP or computationally intensive projects in mind will be most interested in the dedicated multipliers Xilinx® Spartan®-3 generation FPGA devices have to offer; the potential is enormous, with anything from 3 to 104 available per device.

This white paper describes methods for expanding the natural bit-width capability of dedicated multipliers in a way that will make best use of the complete FPGA resources. This information is applicable to the Spartan-3 generation FPGA families. The same dedicated multiplier is found in the Virtex®-II FPGAs and Virtex-II Pro FPGAs, with up to 444 multipliers available.

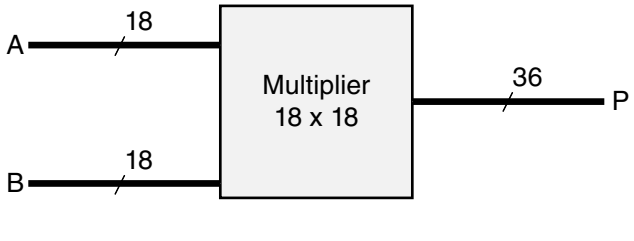

Figure 1: **Dedicated Multiplier**

Each multiplier (Figure 1) supports up to 18-bit by 18-bit signed inputs, providing support for a huge range of applications. While many people exploit the configurable nature of Xilinx FPGA devices by reducing bit widths and therefore reducing product cost, I also see an exciting trend towards extending arithmetic precision (using more bits) to improve the quality of results and even make certain algorithms practical for the first time. This is particularly interesting in those cases where the processing performance made available by FPGAs exceeds that of ASIC implementations, while at the same time providing a standard product solution to applications where the volume simply could not entertain the ASIC development costs.

In this white paper we will look at expanding the natural bit-width capability of dedicated multipliers in a way that will make best use of the complete FPGA resources. This information is applicable to the Spartan-3 generation FPGA families. The same dedicated multiplier is also found in the Virtex-II FPGAs and Virtex-II Pro FPGAs, with up to 444 multipliers available.

Note: Newer Virtex families such as the Virtex-4 and Virtex-5 FPGAs, and the Spartan-3A DSP family, replace the multipliers with complete DSP blocks. For more information on using these dedicated DSP blocks, see the *DSP User Guide* for each family.

Multiplication Revision

Since we are going to exceed the bit-widths supported by a single dedicated multiplier, we are going to need to decompose the multiplication process into smaller sub-processes. In fact, we do this every time the battery runs out in our favorite calculator (and it's too dark for the solar cell to work!). Then, we revert to good old pencil and paper and perform long hand multiplication (Figure 2).

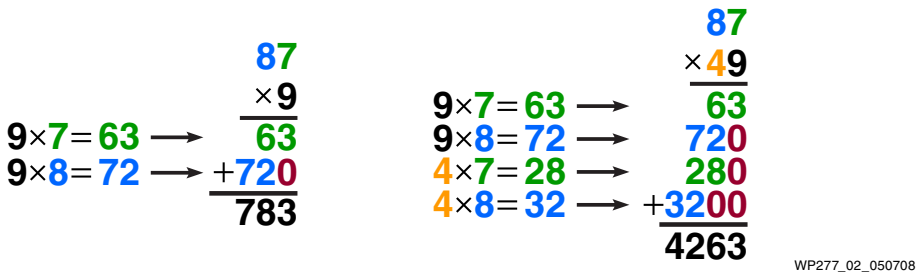

Figure 2: **Long Hand Multiplication with Individual Digits**

In each of the examples in Figure 2, we can see that a number can be split into separate digits. “87” has been split into an “8” (meaning 80) and “7”, and “49” has been split into a “4” (meaning 40) and “9”. Partial products are then formed by multiplying the

individual digits of the multiplicand with the individual digits of the multiplier. Once all combinations have been completed, the partial products are summed to form the final result. Care must be taken to ensure that the weighting of each partial result is applied. We achieve this in long hand multiplication by inserting the "0" to offset our partial product result (for example, when performing 87×49, the last partial product is 4×8=32, but this really means 40×80=3200).

The Weird Split!

As "normal people", we learn to deal with numbers in powers of ten; during our long hand multiplication process, we naturally split the numbers into individual digits. However, the rules still work if we split numbers in weird ways, even if it doesn't make the mental task easier (Figure 3).

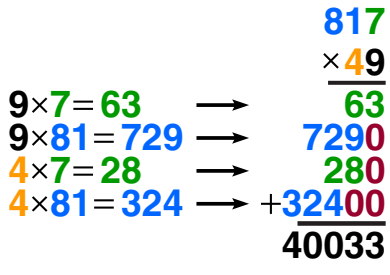

Figure 3: **Long Hand Multiplication with 2 Digits**

We see in Figure 3 that the multiplicand has again been split into two parts; in this case, however, the left-hand part consists of 2 digits. Since we do not naturally know our "81" times-table, we find it hard to work out 9×81=729 (I bet you split it into separate digits or used a calculator!); but so long as the partial product is appropriately weighted during the summation, then the final result is good.

Splitting 2's Complement Numbers

A 2's complement number is an encoded binary representation of a signed value. We tend to learn about 2's complement as some form of "invert and add one" procedure that enables negative values to be represented; however, it is also possible to evaluate a negative value more directly by splitting the number (Figure 4).

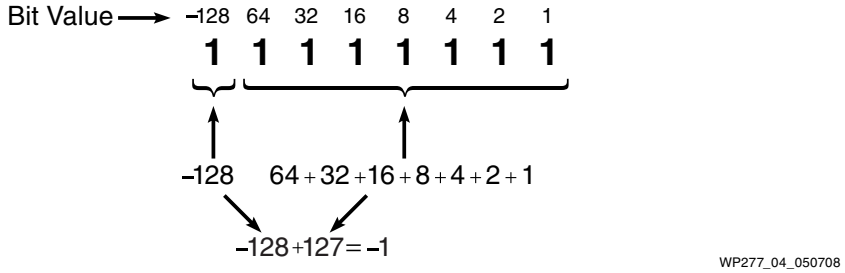

Figure 4: **2's Complement Split into 1 sign Bit and 7 value Bits**

In Figure 4 the all-"11111111" pattern of an 8-bit number is used to represent the value "-1". Rather than invert all bits and add one (00000000+1=00000001), we can see that the least significant bits can be considered to represent a positive value (+127), and the most significant bit, a large negative value (-128). The net effect of this is to form the

value -1. The interesting thing is that we can split the binary representation at any point, provided that the negative weighting associated with the Most Significant Bit (MSB) is taken into consideration (Figure 5).

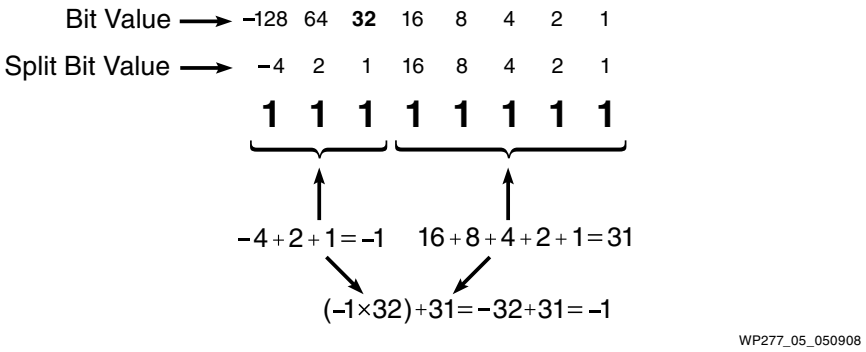

Figure 5: **2's Complement Split into 3 Bits and 5 Bits**

Once again, we take the 8-bit pattern of "11111111", but this time, we split it into *3 bits* and *5 bits*. It doesn't make it easier to work out the value from a human perspective, but the rules still apply. The least significant 5 bits are interpreted as a positive number since all of the original bit values were positive.

However, the most significant 3 bits must be interpreted as a signed value because the MSB has a negative weighting. In this example, the 3 bits have the potential to represent the range of values -4 to +3. With the value of the most significant 3 bits established, it can be added to the value of the 5 bits. We must remember to restore the weighting of the most significant bits, which is a factor of 32 in this example. (Note that -128+64+32=-32 in just the same way that (-4+2+1)×32=-32.)

So, if we split any 2's complement number into two sections, the least significant bits will be a positive unsigned number, and the most significant bits will be a signed 2's complement number in its own right. The offset weighting of the most significant section must be restored at some stage.

Splitting a Dedicated Multiplier

Hopefully, we are now ready to consider how to support a multiplier that is larger than the bit width provided by the dedicated multipliers (without simply reverting purely to CLB-based logic multipliers). Let's have a look at two case studies that are typical of those I frequently see:

Case Study 1

In this first case, we will consider that only one input exceeds the 18-bit limit of the dedicated multiplier (Figure 6).

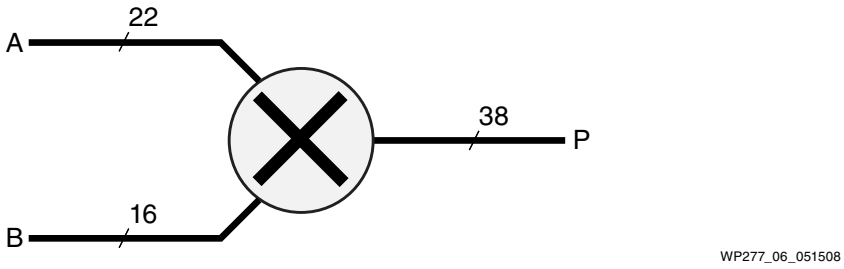

Figure 6: **22x16 Multiplication**

Using the CLB logic to implement this multiplier would require 197 “slices” and 4 levels of logic. Being able to replace this with a dedicated multiplier would clearly save a lot of CLB resources and offer a faster combinatorial process.

Unfortunately, the 22-bit input exceeds the 18 bits supported by the dedicated multiplier; so we need to look at decomposing the multiplication process and splitting the larger input (Figure 7).

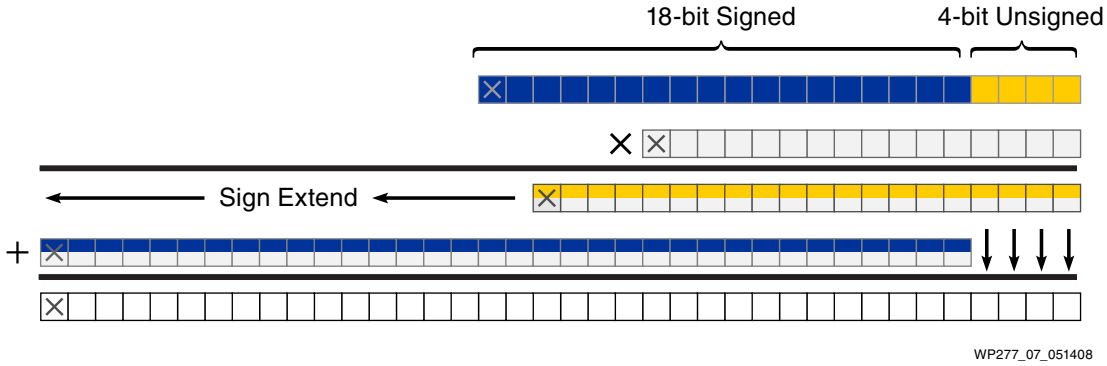

Figure 7: **Splitting 22-bit Input**

Although we could select virtually any point to split the 22-bit input word, it makes sense to take a maximum 18 bits in one section to maximize the use of a dedicated multiplier. As we already know, the most significant section becomes a signed value in its own right, but the least significant section is unsigned. Since the dedicated multipliers are signed multipliers, it is vital that we assign the most significant section to the dedicated multiplier, then consider the options for the support of the smaller unsigned least significant section later.

Two partial products are formed. The first is a 20-bit signed product, which is the result of the multiplying the 16-bit signed value by the 4-bit unsigned section. The second is a 34-bit signed product formed by the multiplication of 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.

So let's have a look at the structure that's required (Figure 8).

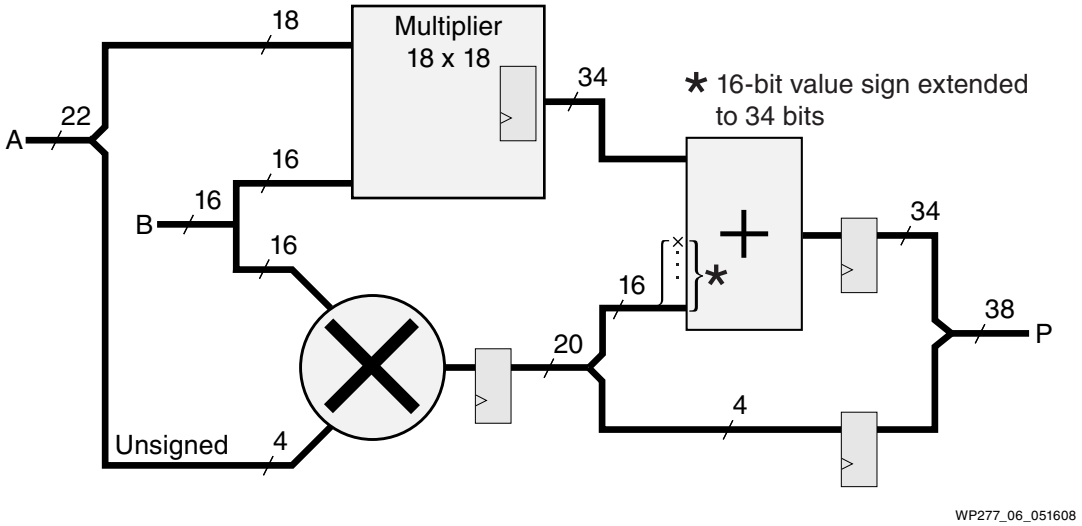

Figure 8: **22x16 Multiplication using One Dedicated Multiplier**

The circuit shows that the dedicated multiplier can be well utilized. A 34-bit adder is required and will be supported by 17 "slices". The question is, how should the second 16-bit signed by 4-bit unsigned multiplier be implemented? There are two options:

1. Use a second dedicated multiplier. Of course, it will not be used particularly efficiently in this case, but if you have no other use for a dedicated multiplier (i.e., you have enough available in your target device), then inefficient use is better than complete waste.
2. Implement this smaller multiplier using CLB logic. In this case, a 16×4 multiplier (note that this is not a 4×16 multiplier) requires 27 slices and 2 levels of logic.

Performance and Synthesis Tools

Clearly, the structure of this decomposed multiplier has more than one level of logic, and as a combinatorial multiplier, this must add to the delay and reduce system performance. Fortunately, we can pipeline multipliers in most algorithms and structures, and the flip-flops required to do this are available.

The flip-flop symbols in Figure 8 indicate where pipeline registers can be inserted. The dedicated multiplier in the Spartan-3 family can internally support a single pipeline stage, while the Spartan-3E and Spartan-3A/3AN multipliers support both input and output pipeline stage options. The 16×4 multiplier will also require a single pipeline stage to balance the circuit. Using a second dedicated multiplier makes this easy, but care must be taken when using CLB logic to ensure that only one register is inserted, even though there are two levels of logic.

Of course, the CLB multiplier could be fully pipelined using two registers, but this would require an additional register to be included in the upper dedicated multiplier (this would require CLB based flip-flops for the Spartan-3 devices). Some experimentation may be necessary to determine just how much pipelining is needed to meet your particular requirements.

Synthesis tools are good at decomposing large multiplier structures automatically; however, it is worth investigating the rules that your synthesis tool follows. It is highly likely that if your tool is set to target the dedicated multipliers, it will use these for each partial product multiplier (option 1). This is acceptable if you have adequate

multipliers, but you may need to become involved when you run out of multipliers and yet still have 75% of the CLBs unoccupied!

When performance is your dominating factor, then don't forget that "y <= a*b;" describes a combinatorial multiplier (or a combinatorial multiplier followed by a single register when described in a clocked process), and that the decomposed multiplier structure will therefore incur greater delay than a single dedicated multiplier. Again, taking control of the decomposition yourself will also enable you to pipeline the structure and achieve your required performance.

Case Study 2

In this second case, we extend the second input to also exceed the 18-bit limit of the dedicated multiplier (Figure 9).

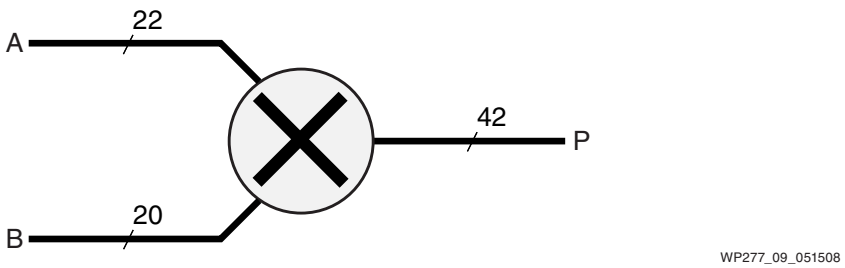

Figure 9: **22x20 Multiplier**

The multiplication must now be decomposed into 4 partial products. Once again, we can maximize the use of the signed 18×18 bit dedicated multiplier by selecting our split points of each word (Figure 10).

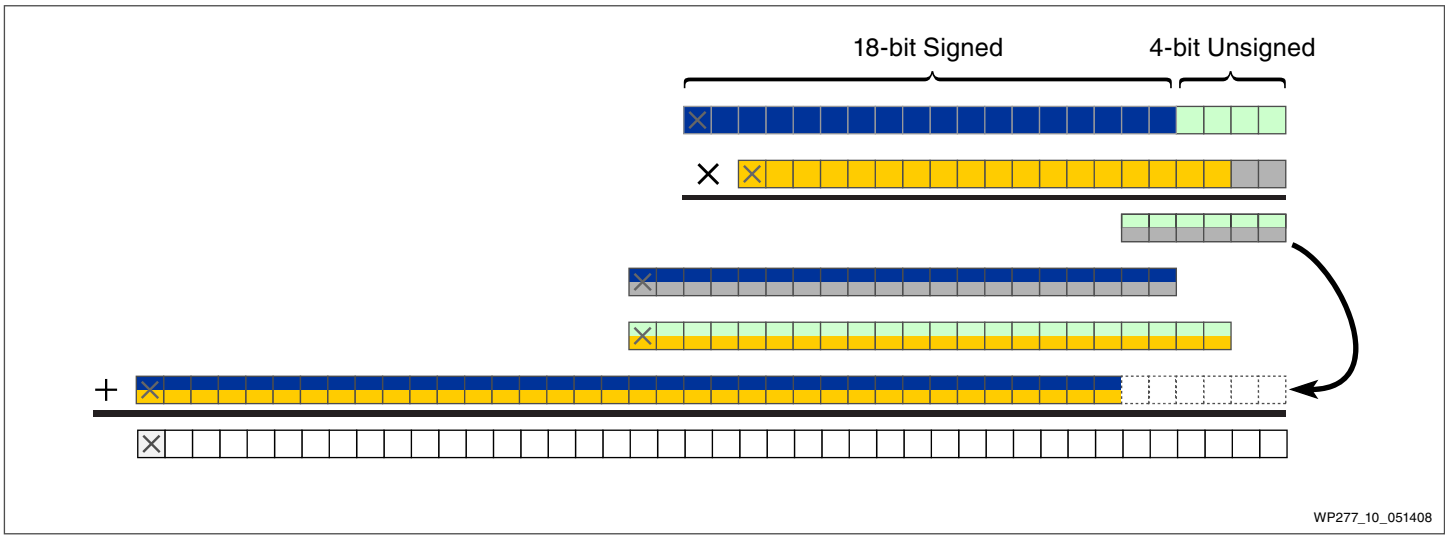

Figure 10: **Splitting 22-bit and 20-bit Inputs**

The partial products require the following multipliers:

1. A 4-bit unsigned × 2-bit unsigned multiplier producing a 6-bit unsigned product (3 "slices" of logic).
2. An 18-bit signed × 2-bit unsigned multiplier producing a 20-bit signed product (10 "slices" of logic).

3. An 18-bit signed × 4-bit unsigned multiplier producing a 22-bit signed product (30 "slices" of logic, or possibly a dedicated multiplier).
4. An 18-bit signed × 18-bit signed multiplier producing a 36-bit signed product (dedicated multiplier).

As you can see, some of the multipliers are really so small that use of dedicated multipliers in these positions just feels wrong. Once again, make sure that you implement an 18×2 (one level of logic) and not a 2×18 (horrendous!) multiplier.

Although 4 partial products must be summed, the first purely unsigned 6-bit product can simply be concatenated with the last 36-bit signed product. The 20-bit and 22-bit products can be added to form a 23-bit value using a 21-bit adder. Finally, the 42-bit result is formed using a 40-bit adder. With some multipliers followed by two levels of addition logic, there is greater reason to consider the structure and insert pipeline registers for higher performance.

Summary

Although it is ideal to adjust the algorithm to fit within the 18-bit limit of the dedicated multipliers, larger multipliers are possible, making good use of the dedicated multipliers as building blocks. Remember the effect that decomposing a multiplier will have on performance, and look to insert pipeline registers when required.

Very large multipliers will obviously benefit from using the dedicated multipliers for all partial products, but a mixture of dedicated multipliers and CLB-based multipliers will often enable the selection of a smaller device. For more details see

UG331 Spartan-3 Generation User Guide, Chapter 11 *Using Embedded Multipliers*.

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
05/22/08	1.0	Initial Xilinx release. Originally published as a TechXclusive

Notice of Disclaimer