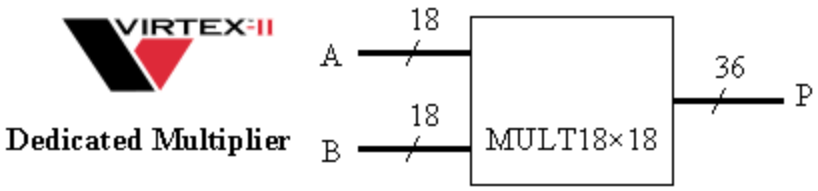
techXclusives

Expanding Virtex-II™ Multipliers

By Ken Chapman
Staff Engineer, Core Applications - Xilinx UK

Introduction

I'm sure you have all been busy looking at Virtex-II™ data sheets and Application Notes recently. Certainly, those of you with DSP projects in mind will have been most interested in the dedicated multipliers these devices have to offer; the potential is enormous, with anything from 4 to 192 available per device.

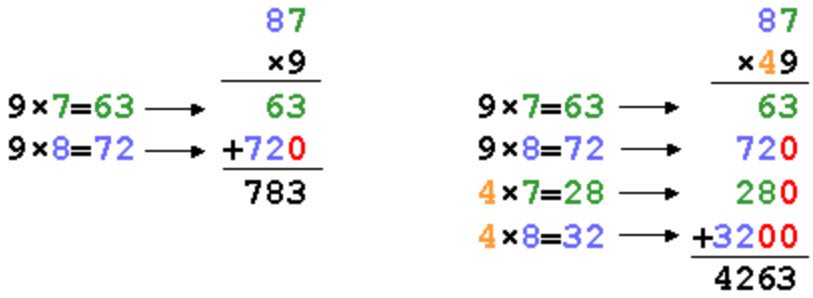


Each multiplier supports up to 18-bit by 18-bit signed inputs, providing support for a huge range of applications. Whilst many people exploit the configurable nature of Xilinx FPGA devices by reducing bit-widths, and therefore reducing product cost, I also see an exciting trend towards extending arithmetic 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 Virtex™ devices exceeds that of ASIC implementations, whilst at the same time providing a standard product solution to applications where the volume simply could not entertain the ASIC development costs (NRE).

So, in this article we will look expanding the natural bit-width capability of the dedicated multipliers in a way that will make best use of the complete Virtex-II resources.

Multiplication Revision

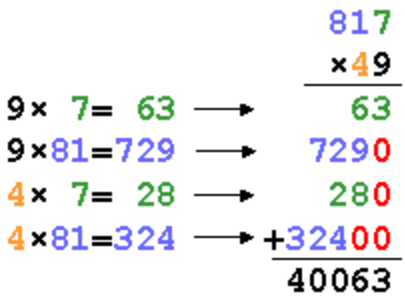
Since we are going to exceed the bit-widths supported by a single dedicated multiplier, we are ultimately going to need to decompose the multiplication process into smaller sub-processes. In fact, we do this every time the battery runs out in our favourite calculator (and it's too dark for the solar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



In each of the examples above, 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 (e.g., in the 87×49 example, the last partial product is 4×8=32, but this really means 40×80=3200).

The Weird Split!

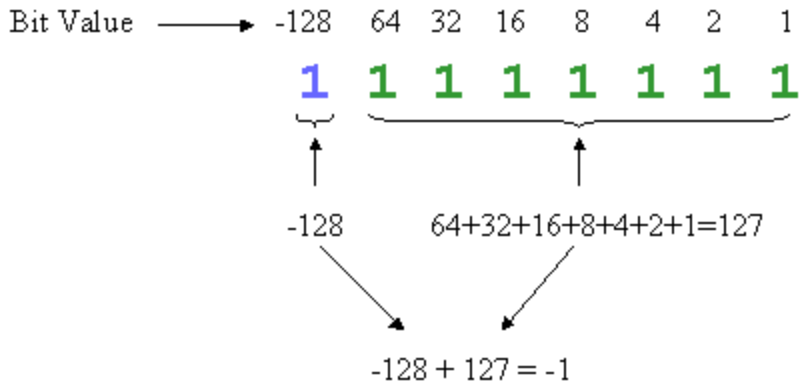
As "normal people", we learn to deal with numbers in powers of ten; during our long-hand multiplication process, we naturally split the numbers into individual digits. However, the rules still work if we split numbers in weird ways, even if it doesn't make the mental task easier...



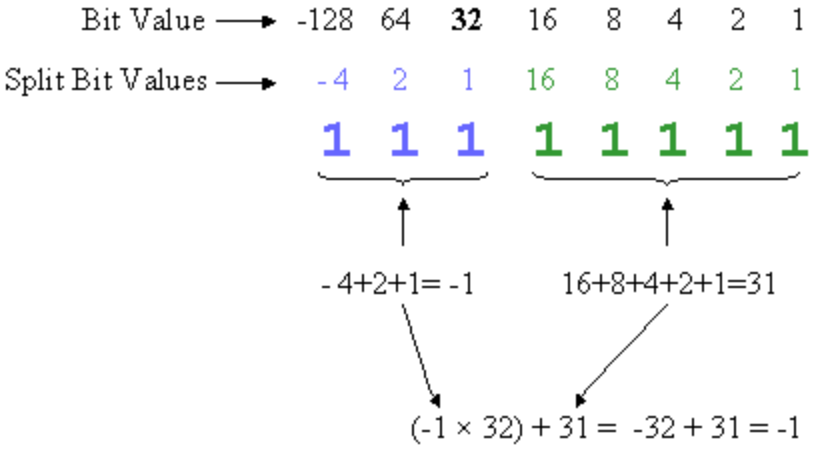
We see here that the multiplicand has again been split into two parts; in this case, however, the left-hand part consists of 2 digits. Since we do not naturally know our "81' times-table, we find it hard to work out 9×81=729 (I bet you split it into separate digits or use a calculator!); but so long 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...



Here, 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 MSB is taken into consideration...



Once again, we take the 8-bit pattern of "11111111", but this time, we split it into 3-bits and 5-bits. It certainly 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'complement number in its own right. The offset weighting of the most significant section must be restored at some stage.