Verification of 8, 16, & 32-bit Multiplier

Introduction

The multiplier module's interface uses variable N-bit inputs a and b & output product which is a 2N-bit value allowing it to be done for 8,16, and 32-bit multiplication with the same function and handles overflow. The input values for the module are clk, reset_n, start, a, and b. The output values are product, and busy. The clk input is used for the storing clock signal. reset_n input is the resets of the module values. The start input is the start signal for when to start the multiplication process. busy is used to track when multiple processes are complete. These inputs and outputs are 1-bit values. The a and b inputs are the values to be multiplied. These inputs are N-bit values. The product output is the result of multiplication.

Inputs: clk, reset_n, start, a, b
Outputs: product, busy

Implementation

There are 3 different methods of implementation, each with its own set of tradeoffs. All the versions utilize recursive pipelining. At each positive clock edge, a specific bit value of variable b is processed giving the algorithm a time complexity equal to the number of bits in b (O(log_2(b)); b=2^N). For Version 1 variable a is shifted left by the bit position of b and added to the product. For the Version, 2 variable to avoid an extra indexing variable, which is expensive in terms of hardware, variable a is shifted left by 1 bit and added to the product while variable b is shifted right by 1 bit after each clock cycle. For Version 3 to avoid concatenating variable a and having 2N bit addition, the product is shifted right by 1 bit before variable a is added to the upper half of the product and then set as the new product with extra space for carryover. Essentially it uses the upper N bits of the product to store the result and shift right to provide space for carry-over during addition.

8-bit Multiplier Testing

The testing strategy for the 8-bit multiplier was to exhaustingly test all possible input combinations for the 8-bit multiplier. The testbench was written using nested for loops providing a value for variables a and b for each clock cycle. The testbench also checks for the correct output and busy signal. For the 8-bit multiplier, 65536 (256x256) unit tests were applied. The tests were done using assert statements in the testbench. When the assert statement is false the testbench prints out the values of a, b, product, and busy.

16-bit Multiplier Testing

The testing strategy for the 16-bit multiplier was to exhaustingly test all possible input combinations for the 16-bit multiplier. The testbench was written using nested for loops providing a value for variables a and b for each clock cycle. The testbench also checks for the correct output and busy signal. For the 16-bit multiplier, 4294967296 (65536x65536) unit tests were applied. The tests were done using assert statements in the testbench. When the assert statement is false the testbench prints out the values of a, b, product, and busy.

32-bit Multiplier Testing

The testing strategy for the 32-bit multiplier was to randomly test a good number of input combinations for the 32-bit multiplier. The testbench was written using nested for loops and urandom() which provided a random 32-bit value for variables a and b. The testbench also checks for the correct output and busy signal. For the 32-bit multiplier 10000 unit tests were applied. The tests were done using assert statements in the testbench. When the assert statement is false the testbench prints out the values of a, b, product, and busy.

Results

The results from the testing were unsuccessful. While I was able to assign all values using my testbench module, I was unable to get the correct product value. It seems as though the changes to the input value for the multiplier had no effect on the product value provided by the model. Unfortunately, I was unable to figure out why this was happening.