Project 2 Report

Introduction:

In this project we were assigned to design and implement a shift and add 8-bit signed sequential multiplier and implement it on the Basys 3 FPGA board. This project provided a great learning opportunity for us to implement our knowledge in digital circuit design and Verilog into a real-world application. We faced a lot of challenges during the research, design, and implementation phases of the project. Indeed, these challenges increased our understanding of the material and provided us with a real-life example of trying to manage a project within a team during limited time. We provide below our design outline, implementation issues, and validation activities.

Design:

Datapath:

Our first step in this project was to design the DataPath. The Datapath was designed similar to the unsigned shift and add sequential multiplier but with the added feature of detecting whether the product will be negative or positive. The multiplier and multiplicand are first loaded into their respective shift registers. The som button (start of multiplication) gives us the signal to start our algorithm, if the button was pressed, we check the most significant bit of both the multiplier and the multiplicand. If it is one, then the loaded number is negative. We then proceed to get their two’s complement and load them into separate registers. We also perform and xor operation on the most significant bit of the multiplier and the multiplicand before getting the two’s complement. If the result from the xor operation is one, then we know our product must be negative since we have a negative number multiplied by a positive one or vice versa. Thus, if the sign flag is one, we make sure to output a negative sign next to the product at the end of multiplication. To sum up, we eventually compute the absolute value of the product of the multiplier and multiplicand and then decide on the sign when outputting the product. After finishing the loading phase of both the multiplier and multiplicand, we proceed to our shift and add algorithm. We have the product register initialized with zeroes. Noting that the product register is 16 bits because the product of multiplying 8-bits by 8-bits can reach a number that is only representable by 16-bit. We first examine the least significant bit of the multiplier register if it is one then we add the product with the multiplicand and save the result in the product register, this gives us our first partial product. Afterwards, we shift the multiplier to the right one-bit and the multiplicand to the left one-bit, and examine the least significant bit of the multiplier if it is one we add the multiplicand to the previous partial product stored in the product register, else if the least significant bit is zero we don’t add anything to the product register, the same principle applies when we calculated our first partial product. After repeating this algorithm several times, we need to stop, and according to our design we stop when the multiplier register is empty, ie fully loaded with zeroes. We set the flag eom (end of multiplication) to one when our multiplier register is empty, this flag is set to the output of all the bits of the multiplier register going through a nor gate, thus the eom flag will only be one if the multiplier register is zero. Of course, what controls these operations of when to stop, when to add, when to shift, and when to load is the control unit and it will be discussed thoroughly in the next section.

Control unit:

Binary to BCD:

After getting the output in the product register, we needed to display it, but we were faced with the challenge of segmenting the bits into units, tenths, hundredth, thousands, and ten thousand. We thought of multiple ideas, but they all seemed hard to implement. Until we found the infamous shift add 3 algorithms. The algorithm converts binary representations of a number into a decimal representation. It works using a shift and add method. First, we consider each four digits as a representation of units, tenths, hundredth and so on in the final output of the shift add 3 methods from right to left. We then shift our binary digit one bit to the left, we now have one bit in the digits row, now the algorithm goes as follows for every row, if the 4-bits in that row are greater than or equal five we add three to the digits and then we proceed with shifting left one bit, if they bits don’t represent a number greater than or equal to five then we shift to the left one bit regardless. If we exceeded four bits because of our shifting left operations, then we are now in the tenths row. We now examine the bits in the tenth row. Of course one or two bits in the tenths row will never be equal to exceed five and thus we shift until the tenths row contains at least three bits and now we examine the bits, if they represent a number greater than or equal five (from left to right) we add three and shift, if they don’t represent a number greater than five we shift to the left one bit regardless. We continue in this fashion until we have one digit in the ten-thousand row for the 16-bit input or in other words until we have done exactly 15 shifts to the product represented in binary. Thus, we now have in every row our digits for the units, tenths, hundredth, thousands, and ten-thousands. We had to choose between two ways to implement this algorithm, either using a sequential circuit with shift registers or a combinational circuit with add three circuits that decides if each four bits are greater than or equal to five or not. We chose the combinational option as it was easy to implement and debug. Moreover, after we get our output from our Binary to BCD converter, we then display each relevant digit (depending on the scrolling feature, which will be discussed) onto a seven sevent segment display using a BCD to Seven segment decoder, which we previously implemented in the lab and were familiar with it.

Scrolling:

Implementation issues:

One major issue we faced during implementation was that the buttons wouldn’t start the multiplication. We examined the whole code from beginning to end but couldn’t quite figure it out. Until we noticed that we hadn’t included a reset button in the module that initializes the buttons, this in turn affected our denouncer, synchronizer and edge detector modules. We also faced the issue of the scrolling being too fast, so we had to include a clock divider to the shifter so that it shifts the digits slower.

Contributions: