Optimized 8-bit Signed Shift-Add Multiplier (Basys3)

Mohab Bahnassy

Omar Youssef

Hassan Mohamed

Peter Aziz

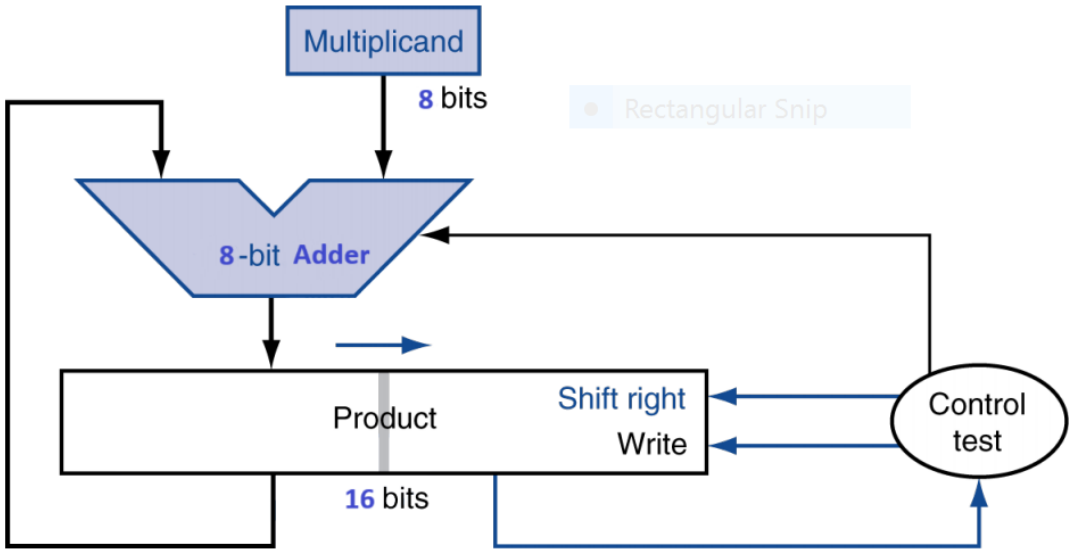
The American University in Cairo

Digital Design I

Table of Contents:

1. Design Outline

Our implementation of the shift add multiplier involved a single 8-bit register holding the multiplicand, an 8-bit carry select adder, and a 16-bit shift register holding both the product and the multiplier. The control unit also controls the interactions between all the components and signals the end of the multiplication. Since the components output the product in binary form, we also implemented a double dabble component that converted the binary number to a form that can be displayed in the seven-segment display.



The efficient implementation of multiplication operations is critical for a wide range of applications. One common approach we used is the shift-add multiplier, which relies on shifting and adding operations to compute the product of two numbers. In this essay, we will explore the design and implementation of an optimized 8-bit shift-add multiplier in the Verilog hardware description language, utilizing the Vivado design suite.

The primary objective of the design is to create an 8-bit multiplier that holds both the multiplier and product in a single 16-bit register. This design choice helps conserve resources and streamline the hardware implementation. The multiplier and product are both 8 bits, and the final result is stored in a 16-bit register.

The design begins by checking whether the multiplier and the multiplicand are negative or positive numbers. If any of the numbers is negative, its magnitude is then produced and used for the rest of the operations. The sign of both numbers is used to get the sign of the product that is then displayed on the right most 7-segment display.

The design begins by initializing the first 8-bits (starting from the most significant bit) of the 16-bit register to zero and the remaining 8-bits are assigned according to the multiplier . This register serves as the container for both the multiplier and the product. The use of a single register is advantageous in terms of resource efficiency.

The core of the multiplier lies in its ability to process each bit of the multiplier iteratively. The process starts from the least significant bit (LSB) and progresses towards the most significant bit (MSB). For each bit of the multiplier, the design checks if the current bit is set to 1. If so, it adds the current value of the product to the 16-bit register. Following this addition, the entire register is shifted left by 1 bit. This shift replicates the effect of multiplying by 2. Simultaneously, the multiplier is shifted to the right by 1 bit.

As for the final result, after processing all eight bits of the multiplier, the 16-bit register holds the final product. This product is the result of the optimized shift-add multiplication algorithm implemented in Verilog. The decision to use a single 16-bit register to store both the product and the multiplier in the design of the 8-bit shift-add multiplier is a strategic optimization aimed at enhancing efficiency and conserving hardware resources. This design choice offers several advantages over alternative approaches, making it a better-suited solution for specific applications.

The first is resource efficiency. Utilizing a single 16-bit register instead of separate registers for the multiplier and product minimizes the overall consumption of resources. In FPGA (Field-Programmable Gate Array) designs, where resources are often limited, this optimization is particularly valuable. It reduces the number of required registers, resulting in a more compact and resource-efficient implementation.

The second is reduced hardware overhead. Separate registers for the multiplier and product would necessitate additional logic and routing to manage data flow between them. By consolidating both values into a single register, the design simplifies the data path and reduces the hardware overhead associated with inter-register communication. This streamlined architecture contributes to faster signal propagation and potentially lowers power consumption.

The third is simplified control logic. A unified register facilitates a more straightforward control logic implementation. The shift-add multiplication algorithm inherently requires iterative bitwise operations, and maintaining both the multiplier and product in the same register simplifies the control signals needed to orchestrate these operations. The reduction in control complexity can lead to faster clock speeds and more straightforward design verification.
We also have ease of integration. Integrating the multiplier into larger systems or incorporating it into more extensive designs becomes more seamless when the product and multiplier share a common storage location. This simplicity in integration is especially valuable in complex digital systems where multiple modules need to communicate efficiently.

The consolidated register approach aligns with the principles of optimization in digital design. It minimizes the critical path length by avoiding unnecessary interconnections and enables more straightforward pipelining or parallelization strategies. As a result, the implementation can achieve better performance in terms of speed and throughput.

2. Implementation Decisions

First, we decided to use the 2's complement for the negative numbers in both the multiplier and the multiplicand. As the 2's complement method uses the most significant bit to which the sign of the number, the two most significant bits of the multiplier and the multiplicand were used to calculate the sign of the product by doing a simple single-bit XOR between them. If one of the numbers was negative, the result of the XOR is 1, then the product is negative and a negative sign is displayed. On the other hand if both numbers are positive or both of them are negative the result will be zero and the product is positive so nothing will be displayed on the right most 7-segment display.

We decided to start the program by displaying 3 dashes in the three seven segment displays. This indicates that multiplication has not yet started. Once the user specifies the two binary numbers they wish to multiply using the switches, they press the middle button to multiply the numbers. An end of multiplication LED lights up and the number is displayed in the seven segment display.

The seven segment display initially shows the least significant digits of the product, and the buttons on both sides of the middle button can be used to navigate to the rest of the digits. The maximum number of digits needed is 5, and the maximum value is 16,129. The leftmost display is reserved for the negative sign, in case the product is negative.

Among our implementation decisions also was to automatically right-shift the 16-bit register, without depending on a control signal from the control unit. However, when the end of multiplication is reached, an EOM signal is sent from the control unit to halt the sifting. The reason for this decision was to reduce the coupling between the components to the minimum and thus reduce the probability of errors.

Since we used an optimized version of the shift-add multiplier, the number of clock cycles needed to finish the multiplication is 8. The shifting and adding is therefore completed in a single clock cycle. This is enabled by the fact that the addition is done through combinational logic.

Implementing an 8-bit shift-add multiplier in Verilog involves several key implementation decisions. These decisions include architectural choices, data representation, and control logic design. Here are some implementation decisions we needed to make:

Data Representation: In this case, both the multiplier and multiplicand are 8 bits. We ensured that the data width is appropriate for our application. Register Width: In this case, a 16-bit register is used. Clock Edge and Reset Handling: We determine whether the multiplier operates on a rising clock edge. Additionally, we handled resets, ensuring proper initialization of internal registers. Shift and Add Operations: We implemented arithmetic and logical shift operations based on our requirements, ensuring the Verilog implementation correctly incorporates these shifts.

Addition Operation: We implemented the addition operation that occurs when a bit in the multiplier is set. Control Logic: We designed the control logic to orchestrate the sequential operations within the multiplier. This includes managing shifts, additions, and the overall loop structure. We used two finite-state machines to coordinate the operations, and control the displayed digits on the seven-segment display.

Testbench Development: We developed a comprehensive testbench to verify the functionality of the multiplier under various scenarios, along with test benches for each component, ensuring that it produces correct results for a range of inputs and edge cases.Synthesis and Optimization: We synthesized the Verilog code using tools like Vivado and analyzed the synthesized results to ensure that the design meets timing constraints and optimize as needed. Resource Utilization: We monitored resource utilization, including the number of flip-flops and logic elements, to ensure that the design fits within the constraints of the target FPGA.

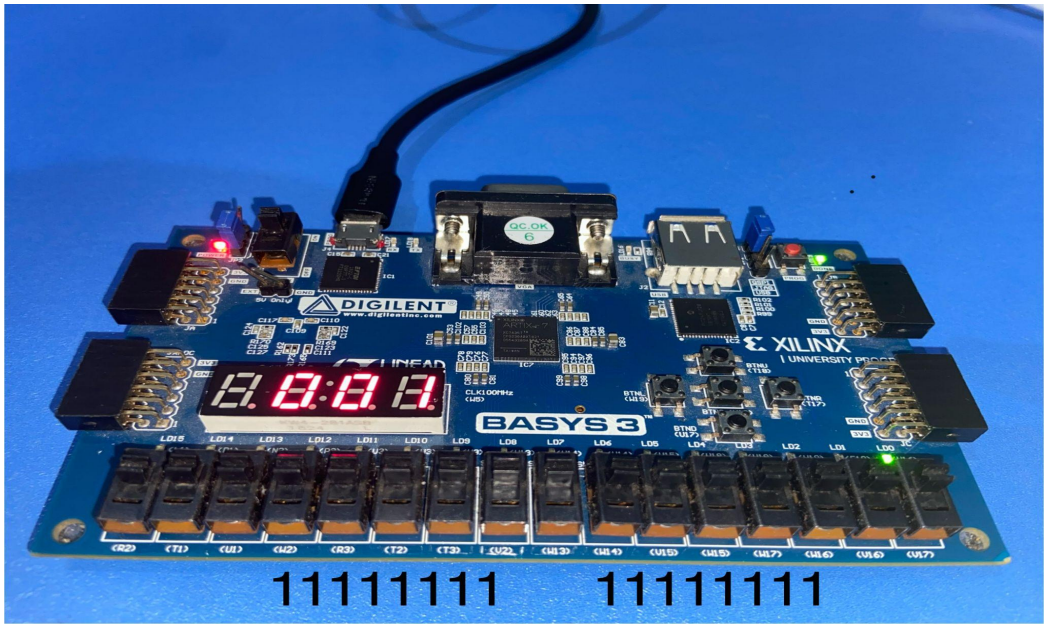
In our implementation, we used two finite-state machines. The first of those was to implement the shifting between digits in the seven-segment display. Since we have only three seven-segment displays reserved for the actual product, we needed three states to each represent the digits to be shown. The movement between the states depends on the push buttons, which could be left or right. As for the second finite state machine, it was used to implement the double dabble algorithm.

---

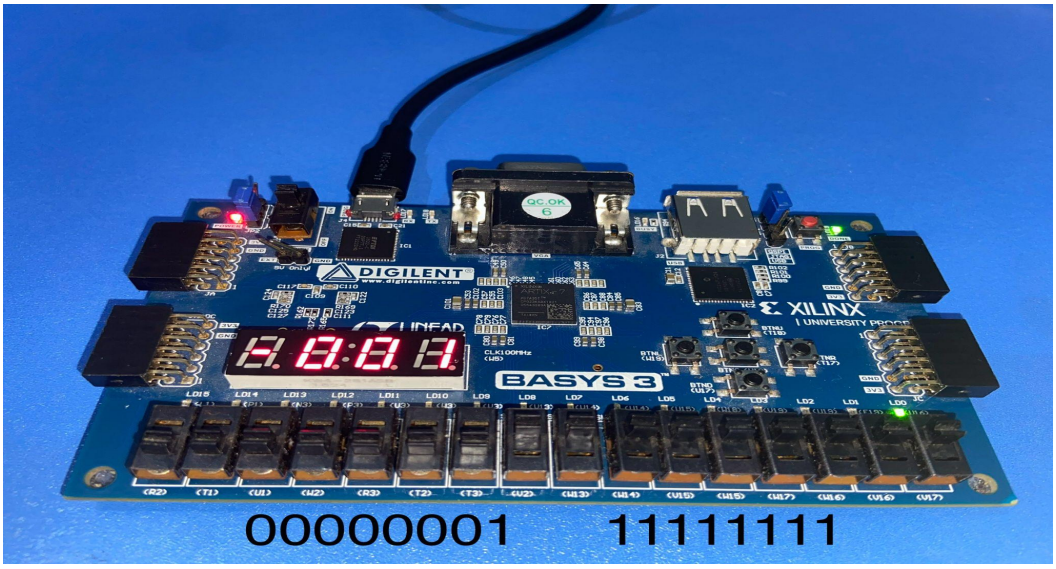
3. Validation:

To test and validate the program, we tried some random numbers along with all the corner cases we could think of:

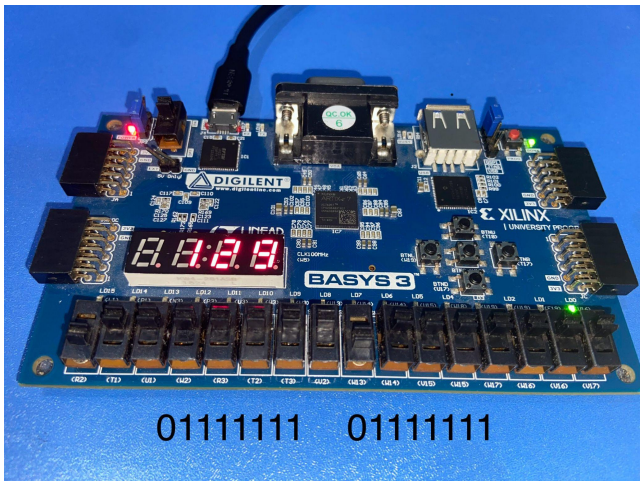
Test 1 (-1 × -1 = 1):



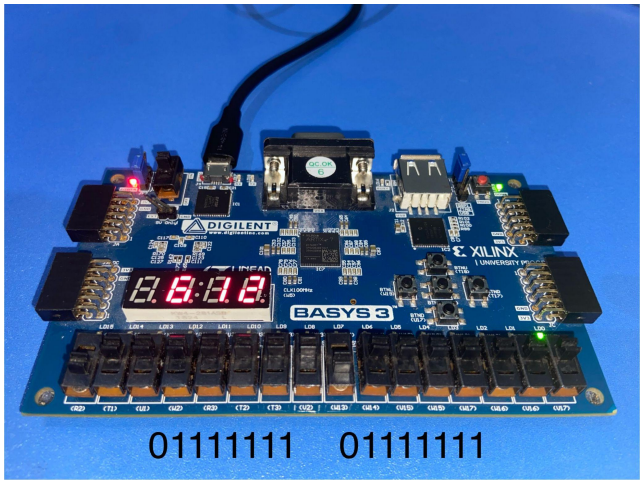
Test 2 (1 × -1 = -1):



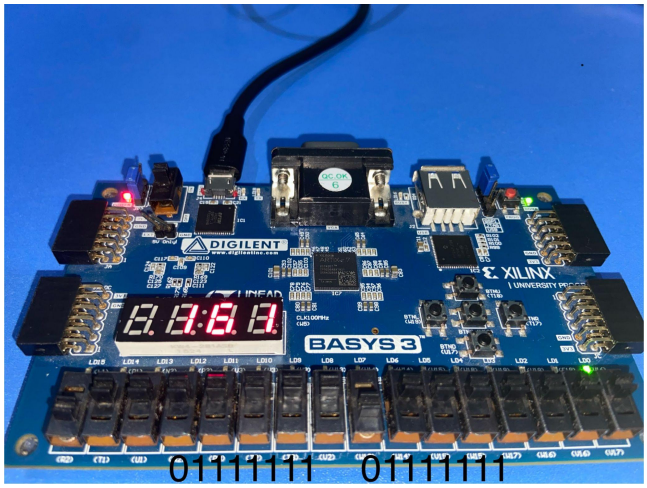
Test 3 (127 × 127 = 16129):



16[129]

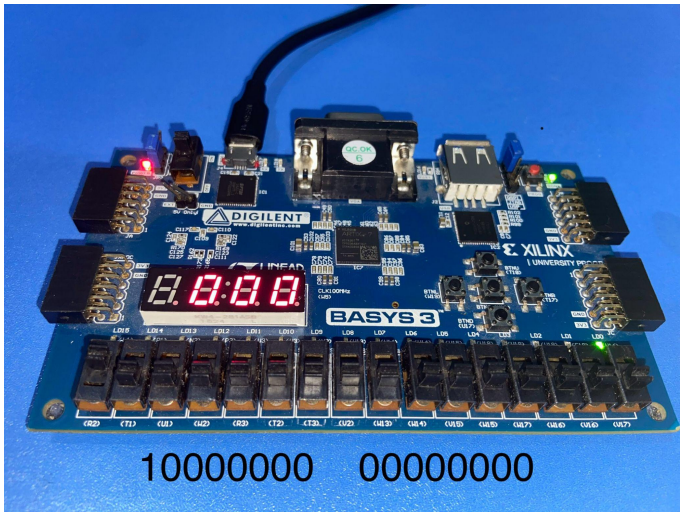


1[612]9



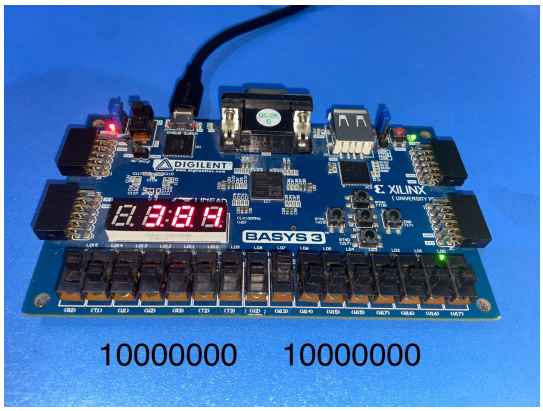
[161]29

Test 4 (-128 × 0 = 0):

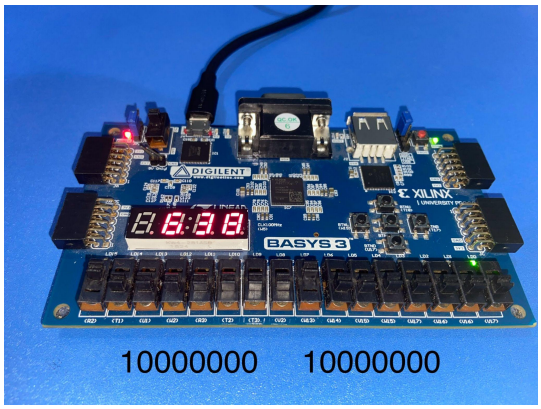


Here, we handled the - 0 case when a negative number is multiplied by a zero we need not get a -0, but rather a 0. So, here it is handled.

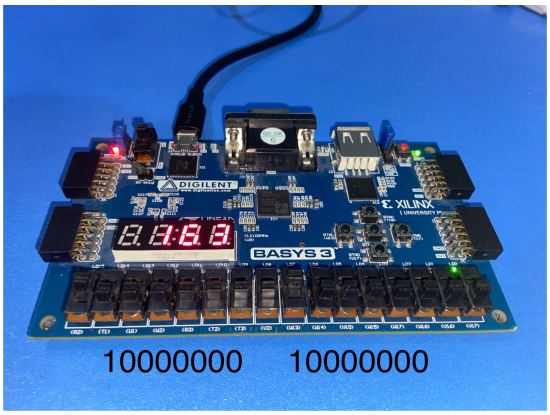
Test 5 (-128 × -128 = 16384):



16[384]

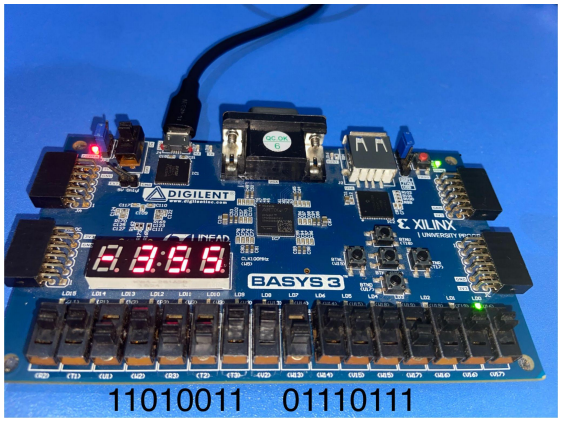


1[638]4

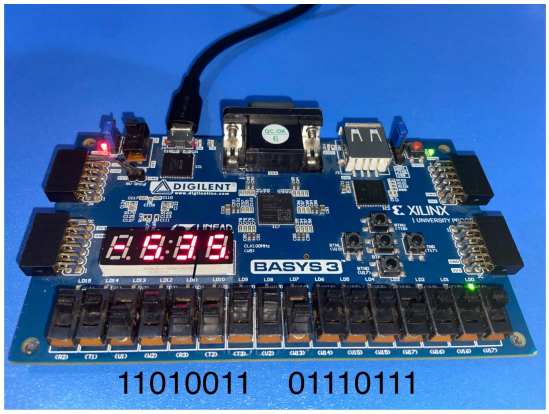


[163]84

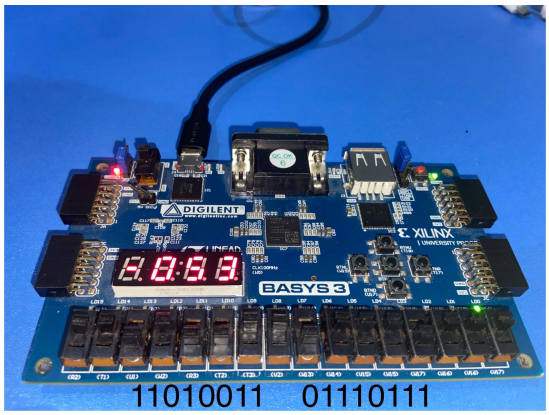
Test 6 -random number- (-45 × 119 = -05355):



-05[355]



-0[535]5



-[053]55

We tried many other numbers while implementing and testing the program. However, these are the ones we recorded. We also implemented a few test cases using the test bench simulator on Vivado. Unfortunately, we did not take any screenshots. However, their codes are in the main file.

4. Member Contributions

Milestone I:

Name	Contribution
Hassan Ahmed Mohamed	2's complement, Display
Omar Ahmed Yossuf	Main, Control Unit, Display, Multiplication, seven segment
Peter Hany	Lucid chart, shifting left/right
Mohab Bahnassy	Display, double dabble

Milestone II:

Name	Contribution
Hassan Ahmed Mohamed	Double Dabble, Display, Left/Right Shifting,Main, Seven segment
Omar Ahmed Yossuf	Main, Control Unit, Display, Left/Right Shifting, seven segment
Peter Hany	Multiplier, Main, Display
Mohab Bahnassy	Multiplier, Main, Display

5. Usage Instructions

Step 1: represent the two decimal numbers that you want to multiply into 8-bit binary with the MSB being the sign bit, for example (-2 x 4), (11111110 x 00000100).

Step 2: in the FPGA. The first eight switches represent the multiplicand and the following eight represent the multiplier.. Turn the switches ON or OFF depending on the two 8 bit binary numbers that were represented in step 1 where OFF represents a zero and ON represents a one.

Step 3: Press the middle button in the FPGA to start the multiplication. Once the Multiplication is complete an LED light will turn on indicating that the multiplication is finished.

Step 4: Initially three numbers will be shown on the seven segments accompanied by the sign, use the right and left buttons to shift the display to view the full number. The initial numbers shown will be the three least significant numbers (units, tens, hundreds) and to show the higher significance numbers (thousands, ten-thousands) you need to press the left button (BTNL) once for the thousands and twice for the ten-thousands.