

Project Report

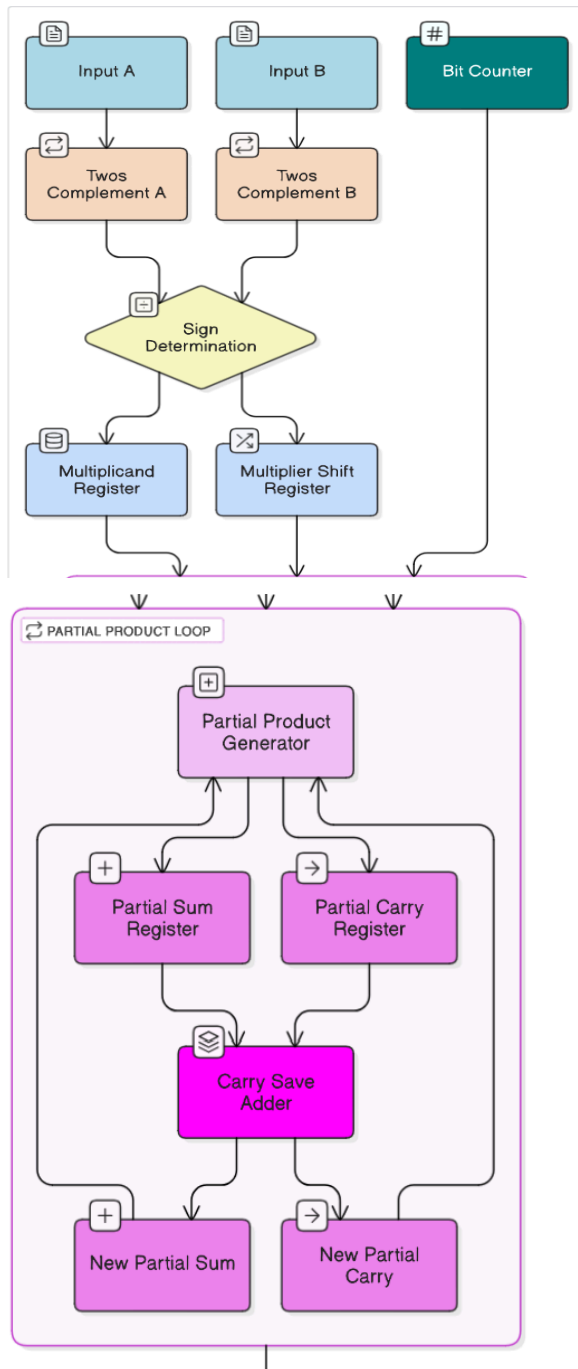
8x8 SPM

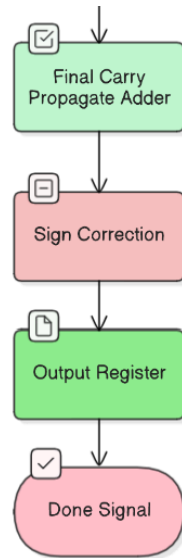
Contents:

1. SPM Implementation
2. Display Implementation
3. Validation and Testing
4. Contributions

SPM Implementation

Signed Serial-Parallel Multiplier Flow Chart





The internal structure of our Serial-Parallel Multiplier (SPM) follows the schematic diagram shown. This multiplier is designed to take two 8-bit signed inputs and produce a 16-bit signed product. It processes one input (**X**) in parallel and the other (**Y**) serially — handling one bit of **Y** at a time. This sequential approach distributes the computation across multiple clock cycles, significantly reducing hardware complexity.

At a high level, the circuit multiplies each bit of the serial input **Y** with all bits of the parallel input **X** in one clock cycle, generating a row of **partial products**. Over successive cycles, these rows are accumulated using **carry-save adders (CSAs)**, which allow intermediate sums and carries to be stored without immediate carry propagation. This strategy enhances performance by simplifying addition and deferring carry resolution until the end.

To properly handle signed inputs, we incorporate **two's complement logic** at the front of the circuit. This block checks the sign bit (most significant bit) of each input and converts negative values into their two's complement form to ensure correct binary multiplication behavior.

Throughout the process, partial sums and carries are stored in dedicated registers. With each clock cycle, the multiplier shifts and updates these values in response to the next bit of **Y**. After eight cycles — one for each bit of the serial input — the final 16-bit result is produced.

This serial-parallel architecture strikes a balance between efficiency and simplicity. By spreading computation across time and minimizing hardware usage, it's particularly well-suited for **FPGA implementations** and other resource-constrained environments where low area and power consumption are critical.

Display Implementation

Once we obtain the multiplication output, displaying a 16 bit number on a 4 segment display is far from trivial. To begin, we must convert the 16 bit decimal to 5 BCDs (Binary Coded Decimals) to be able to connect each number to a 7-segment display. To convert the 16-bit binary output of the serial-parallel multiplier into Binary Coded Decimal (BCD) format, the Double Dabble algorithm was implemented. Double Dabble, also known as the shift-and-add-3 algorithm, is an efficient and hardware-friendly method for converting binary numbers to BCD.

The algorithm works by performing a series of left shifts on the binary input while adjusting any BCD digit that becomes 5 or greater. This ensures proper decimal digit representation as bits are shifted into place.

Steps of the algorithm:

1. Initialize a BCD register large enough to hold the resulting decimal digits (typically 5 digits for 16-bit input).
2. Shift the binary input left into the BCD register one bit at a time, starting from the most significant bit.
3. Before each shift, check each BCD digit (4-bit groups). If a digit is 5 or greater, add 3 to it.
4. After all 16 shifts, the BCD register contains the correct decimal representation of the binary input.

This method was chosen due to its simplicity, suitability for hardware implementation, and minimal resource usage. It allowed for an accurate and reliable conversion of the multiplication result to a human-readable decimal format. This method is quite impressive since it is the only method that does not require division or modulo which are very costly in hardware.

We then reach another obstacle. The BASYS 3 FPGA has only 4 seven segment displays, whereas our largest case scenario is $-128 * -128$ which is a 5 digit number. Not only that, but we also need to show when a product is negative. Hence our worst case is a 5 digit negative number, needing a total of 6 displays where we only have 4. Hence we implemented a scrolling mechanism using push-buttons right and left.

Our approach was to build an FSM as follows:

Ex. -16723

State A:

Sign and left most digits- -167

If Right pressed- Go to state B

Else Stay

State B:

Sign and left most digits- -672

If Right pressed- Go to state C

If Left pressed- Go to state A

Else Stay

State C:

Sign and left most digits- -723

If Left pressed- Go to state B

Else Stay

Validation & Testing

To make sure the serial-parallel multiplier was working correctly, we tested it with a wide range of input values — both positive and negative — and carefully observed the output at each stage. We started by simulating the design using basic testbenches that fed in known values of X and Y and checked whether the output matched the expected 16-bit product.

We made sure to include:

- **Positive \times Positive** cases (ex 5×6)
- **Negative \times Positive** (ex. -3×7)
- **Positive \times Negative** (ex 8×-4)
- **Negative \times Negative** (ex -2×-9)
- **Edge cases**, like multiplying by 0 or by -128 (the most negative 8-bit value)

Each case was chosen to test a different part of the logic — for example, signed multiplication handling, overflow detection, or proper shifting in the final product. For every test, we checked the binary output, then passed it through the Double Dabble block to convert it to BCD. Finally, we verified that the correct decimal digits were shown on the 7-segment display.

We also compared the hardware output against results calculated in software to confirm correctness. To catch subtle bugs, we slowed down the clock in simulations and stepped through the multiplier cycle by cycle to see how partial products were built up over time.

This combination of manual tests, edge cases, and waveform simulation helped ensure that the design was reliable and accurate under all expected conditions.

Contributions

Serial Parallel Multiplication Implementation- Omar Osama & Omar Khalil

Binary Output To Display Implementation- Kareem Rashed