CSCE 2301

Project Report
8x8 SPM

Contents:

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

SPM Implementation

Figure 4. Bit Serial (Carry Save) Adder

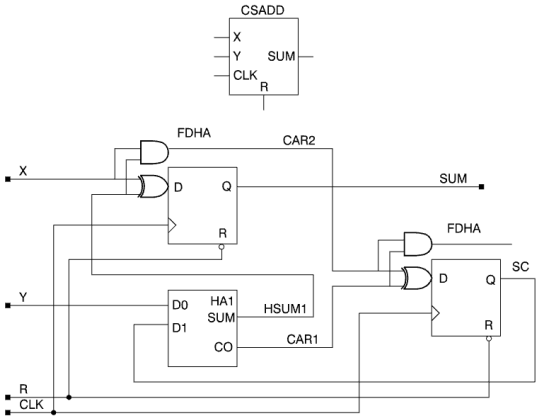


Figure 5. Two's Complement

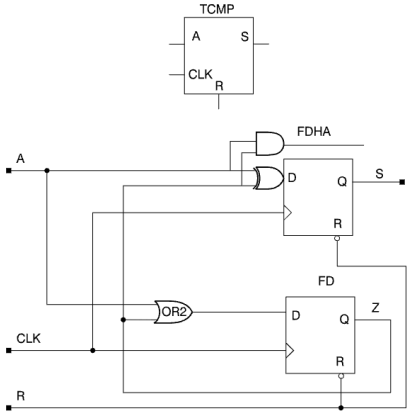
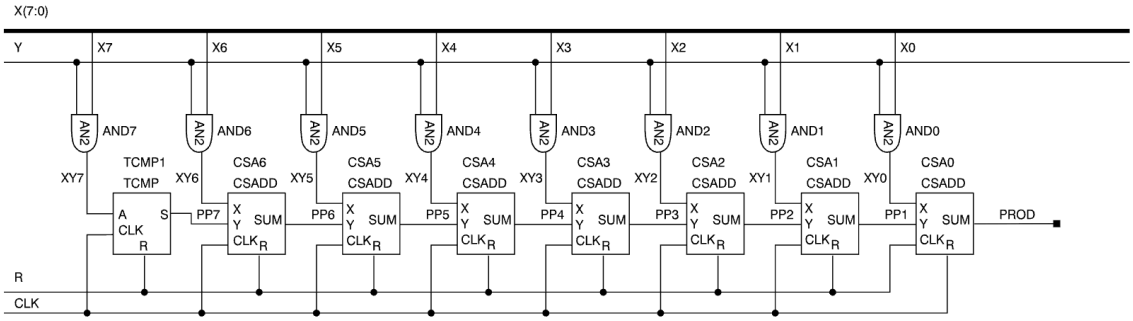


Figure 6. 8-bit Serial-parallel Multiplier



The internal structure of our SPM followed this schematic diagram here. The serial-parallel multiplier we built takes two 8-bit signed numbers and produces a 16-bit product. One input (X) is treated in parallel, while the other (Y) is processed serially — one bit at a time. The whole operation is spread across multiple clock cycles, which helps reduce hardware complexity.

At a high level, the circuit works by taking each bit of the serial input Y and multiplying it with all bits of X in parallel. This gives us a row of partial products for each bit of Y. These partial products are then added together using a chain of carry-save adders (CSAs), which let us delay carry propagation and simplify the addition process over time.

Because we're working with signed numbers, we also had to handle negative inputs. To do this, we used a two's complement block at the start of the multiplier. This block detects if the number is negative (based on the most significant bit) and, if so, converts it using two’s complement logic to make sure the math works correctly.

All the partial sums are stored in registers as we move through each bit of Y. With every clock cycle, we update the sum and shift everything into place. After eight clock cycles (one for each bit of Y), the final 16-bit result is ready at the output.

This design keeps things simple and efficient by avoiding complex parallel multipliers. It also makes good use of clock cycles and minimal hardware, which is ideal for FPGA implementations or any setup where saving space and resources matters.

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 × Positive** cases (ex 5 × 6)
- **Negative × Positive** (ex. -3 × 7)
- **Positive × Negative** (ex 8 × -4)
- **Negative × 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.

CSCE 2301

Contributions

Serial Parallel Multiplication Implementation- Omar Osama & Omar Khalil

Binary Output To Display Implementation- Kareem Rashed