THE AMERICAN UNIVERSITY IN CAIRO

Computer Science and Engineering Department

Sequential Multiplier

Kareem Sayed, ID:900212697

Mohamed Ahmed, ID:900211305

Andrew Aziz, ID:900213227

Raef Hany, ID:900213194

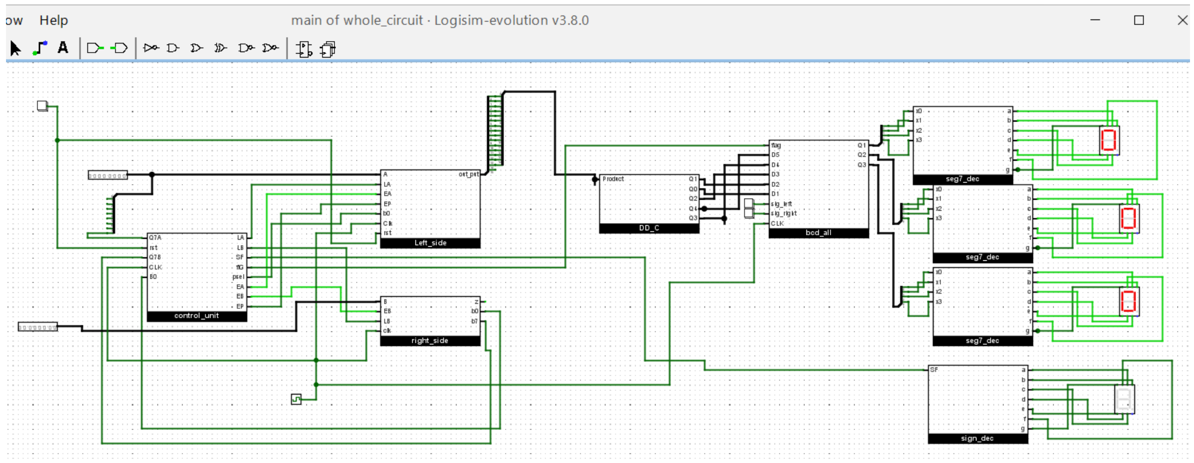
May 26, 2023

Abstract

This report aims to provide a comprehensive and detailed overview of the sequential multiplier's various design and implementation phases. The report covers three main areas, the design of the multiplier, implementation issues and challenges, alongside the validation techniques and activities followed to ensure accurate results. The two main implementation stages are addressed, starting with the initial translation of the multiplier logic into hardware on logisim, followed by the Verilog implementation on the FPGA. Corresponding to each of the two main stages, the design methodology is justified along with the obstacles encountered and techniques employed to ensure optimal implementation and the desired result. Consequently, providing a detailed step-by-step description of the development process.

Contribution: We all worked together on each and every step in this project.

Circuit Design:



We divide our whole circuit into many circuits (blocks) connected to each other, which are:

1- Left Side circuit:-

Description of inputs: The left side circuit will take the 8-bit multiplier (A) as an input, and it will take signals such as load A (LA), enable A (EA), enable P (EP), the least significant bit of the multiplicand that comes from the right side (b0), clock (CLK), and reset (rst). LA signal is responsible for either loading new data into shift-left registers (if it is equal to 1) or not loading it (if it is zero). EA signal is responsible for either shifting the data inside shift-left registers to the left (if it is equal to 1) or not shifting them (if it is zero). EP signal is responsible for either loading the partial products into a register which will contain the final output of the multiplication at the end (if it is equal to 1) or not (if it is zero).

Description of outputs: The output of the left side is 16-bit, which will be the result of multiplying the multiplier A by the multiplicand B.

The whole function of this circuit: This circuit will take the multiplier A and check if its most significant bit is zero or one through a multiplexer. If it is one, it will take its two's complement; otherwise, it will take it as it is. Then the output of this multiplexer will be the input of shift-left registers. At the beginning of multiplication, these registers will be loaded with the output of the multiplexer through the LA signal. However, they will shift this data through the EA signal until the multiplication ends. The output of this circuit will be stored inside a 16-bit register. If b0 is zero, this register will not be loaded with new data. However, if b0 is 1, then this register will be loaded with the summation of its output and the output of the shift-left register through the EP signal.

2- Right Side circuit:-

Description of inputs: The right side circuit will take the 8-bit multiplicand (B) as an input, and it will take signals such as load B (LB), enable B (EB), and Clock (CLK). LB signal is responsible for either loading new data into the shift-right registers (if it is one) or not (if it is zero). EB signal is responsible for either shifting the data inside the shift-right register to the right (if it is one) or not (if it is zero).

Description of outputs: The outputs of this circuit are zero flag (z), b0, and b7. Zero flag represents if the multiplicand is zero or it reaches after shifting. b0 represents the least significant bit of the multiplicand after shifting it to the right. b7 represents the most significant bit of the multiplicand (B).

The whole function of this circuit: This circuit shifts the multiplicand to the right. It takes the multiplicand and checks if its most significant bit is zero or one through a multiplexer. If it is one, the output of the multiplexer will be the two's complement of the multiplicand: otherwise, it will be the same. The output of this multiplexer will be input to an 8-bit shift-right register that will shift the multiplicand to the right to be able to get the b0 after each shifting and will be input to the left side circuit.

3- Control Unit circuit:-

Description of inputs: Q7A is the most significant bit of the multiplier (A). Q7B is the most significant bit of the multiplicand (B). b0 is the least significant bit of the multiplicand (that will be the output of the right side circuit), and it will change after each shifting right of the multiplicand in the right side circuit. rst is the reset which will be our start button to start the multiplication. It resets the counter to start counting from zero again to start a new multiplication.

Description of outputs: LA, EA, and EP are the same signals that were described in the left side circuit. LB and EB are the same signals that were described in the right-side circuit. psel is the same as the b0 signal. The SF signal represents the sign of multiplying A by B (if it is one, the sign is negative). The flg signal represents that the counter (which is in the control unit) reaches 6 to reset the registers to zero in the bcd_all circuit.

The whole function of this circuit: LA and LB are connected to the reset because when reset is 1, we want to load new data (multiplier A and multiplicand B or their two's complement) in the shift registers in the right-side and left-side circuits. EA and EB signals will always be one until the counter reaches 7 (they will be zero). This is because we want the shift registers in the right-side and left-side circuits to shift the data until the multiplication ends (the counter reaches seven). EP signal will be one when the b0 is one to be able to load the data (the summation that happens in the left-side circuit) in the register in the left-side circuit. However, when b0 is zero, we do not want to load new data, so EP is connected to the b0 signal. It is also connected to the reset to reset the data to zero in it at the beginning of multiplication. Thus, This circuit controls the whole signals of the whole circuit to determine which signals should be one and which signals should be zero.

4- The double dabble circuit (DD_C):-

Description of inputs: This circuit will take the output of the left-side circuit (which is the final product of multiplying A by B) as an input. This input will be only 15-bit because we will only need the magnitude of the final product without its sign in this circuit.
Description of outputs: The output of this circuit is 20 bits which is divided into five groups of four bits.
The whole function of this circuit: This circuit takes the final product and converts it into groups of four bits where every four bits represent a number from zero to nine in decimal. They are five groups of four bits the maximum number of multiplying 8 bits by 8 bits will be five decimal numbers, each one consists of four bits.

5- The bcd_all circuit:-

Description of inputs: This circuit will take the five groups of four bits (that are the output of the DD_C circuit) as an input. The flag indicates that the counter in the control unit circuit reaches 6 to be able to reset the registers in the bcd_all circuit to zero. The sig_left and sig_right enable shifting the number displayed in the seven segments display to the left or to the right.
Description of outputs: The output will be only three groups of four bits which represents the first three numbers in the equivalent decimal number.
The whole function of this circuit: This circuit enables displaying the first three numbers in the equivalent decimal number. Then by using the sig_left and the sig_right, it enables shifting the number to the left or to the right by using registers to store the number after shifting it.

6- The seg7_dec and the sign_dec:-

The seg7_dec takes 4 bits and converts it to 7 bits that will be input to the display to enable showing the number in decimal.

The sign_dec takes one bit (the SF signal from the control unit) and converts it to 7 bits that will be input to the display to enable showing the negative sign. If the SF is zero nothing will be displayed.

Software Design:

module **Digital_Multiplier()**: this is the top module, which contains all the submodules necessary for calculating the product and displaying the result on the FGPA board.

Functionality:

Inputs : clk, rst, btnl, btnr, [7:0]A, [7:0] B

outputs : [0:6] seg, output [0:3] anode, output led

This function operates as follows:

- First, it has a clock divider instantiation to change the clk of the FPGA from 100 MHz to 16 Hz , which will be used in most of the circuit components.
- It contains a push button detector to properly deal with the button input, which is responsible for shifting the product of the FPGA screen.
- Then, it goes to the multiplier module which is responsible for producing the output.
- Then, the product goes into the Double_Dabble module which will be furtherly illustrated.
- Finally, the Display module to handle all necessary operation to print the output in a proper way on the FPGA screen.

module **clockDivider()**: this module is to reduce the clock frequency from 100 MHz to a suitable frequency for each component in our design. The default is to reduce the frequency to 16 Hz.

module **push_button()**: this is a combinational module that handles the inputs coming from the buttons. These signals are responsible for the shifting that happens to product on the FPGA screen.

Functionality:

Inputs : clk, rst, x

outputs : z

This function operates as follows:

- First it has a clock divider module which takes the 100MHz down to 20Hz, which is the most suitable after experimenting different frequencies.
- Then, it goes to a debouncer instance. The purpose of a debouncing circuit is to filter out the glitches associated with switch transitions.
- The output of the debouncer goes directly to a synchronizer module to ensure the metastable state of the flip flops and registers has passed by introducing a delayed output signal.
- After that, it goes to a rising edge detector module. The rising edge detector is used to detect the rising edge of a signal, which indicates a transition from a low (0) to a high (1) state. To ensure we only have one click no matter how many clock cycles pass while pressing on the buttons.

module **multiplier()**: this module performs signed multiplication of two 8-bit numbers using a shift-and-add algorithm and two's complement arithmetic. It provides the multiplication result, along with flags indicating the sign and zero conditions.
Functionality:
Inputs : [7:0] A, [7:0] B, clk, rst
outputs : [15:0] temp, z_flag, sf
This function operates as follows:

- A two's complement module is instantiated to compute the two's complement of the input A. Then, An assignment statement assigns the value of "A_comp" to the wire "outmuxL" based on the MSB of A. If A[7] is 1, then A_comp is selected; otherwise, A is selected. And, the same steps happened to the input B.
- The "always" block is triggered on the positive edge of the clock (posedge clk) or positive edge of the reset (posedge rst).
- If rst is high:
    The variables outregL, outregR, and temp are reset to their initial values.
    And the sf (sign flag) is set to 0.
- If rst is low:
    The code checks if outregR is not equal to 8'b0 (checks if it is not zero). If outregR is not zero:
        The sf (sign flag) is set by XOR'ing the MSBs of A and B.
        The z_flag (zero flag) is set to 1.
        The contents of outregL are shifted left by 1.
        The contents of outregR are shifted right by 1.
        If the LSB (outregR[0]) is 1:
            the value of temp is updated by adding outregL to temp.
        If the LSB (outregR[0]) is 0:
            the value of temp remains unchanged.
    If outregR is zero:
        The z_flag is set to 0.

module **Double_Dabble()**: the "Double_Dabble" module implements the Double Dabble algorithm to convert a 15-bit binary number into a 20-bit BCD representation. It performs BCD conversion by shifting and adding bits, and adjusts BCD digits if necessary.
Functionality:
Inputs : [14:0] prod
outputs : [19:0] bcd
This function operates as follows:

- First, a for loop iterates 15 times, once for each bit in the input number.

- Within each iteration, the module checks if each 4-bit BCD digit (from the least significant digit to the most significant digit) is greater than 4.
- If any of the BCD digits are greater than or equal to 5, it adds 3 to that digit. This step is necessary for BCD conversion.
- Then, the module shifts the existing "bcd" value one bit to the left (discarding the most significant bit) and adds the next bit from the "prod" input.

module **seven_seg_display()**: display the decimal number generated from multiplication
Functionality:
Inputs : clk, flag, sf, reset, btnl, btnr, [19:0] d
outputs : [0:6] seg, [0:3] ana
This function operates as follows:

The counter_to_decoder module generates a 2-bit count value a based on the clk_out clock. This value determines the active anode.
The bcd_circuit module converts the input data to BCD and provides the converted BCD value through the “Q” output.
The btnl and btnr inputs control scrolling through the digits.
The sign selection logic determines the sign based on the sf input.
The fourdigit_seven_seg module controls the display of the digits and segments based on the selected digit, sign, and active anode.
The resulting segment and anode values are provided as outputs “seg” and “ana” from the seven_seg_display module.

Implementation issues:

This section aims to address execution challenges that were faced while translating the sequential multiplier into physical hardware through Logisim, alongside other obstacles faced in implementing the final design of the Field Programmable Gate Array (FPGA).

A few issues arose during the initial process of developing the Verilog modules to simulate the sequential multiplier. One of the challenges is the inaccurate final results displayed on the FPGA board; this was seen during the testing phase, where the final product was displayed on LEDs. In this stage, the product would appear shifted two or more bits to the left. Thus, results were usually double what was expected. Consequently, the debugging stage took place in search of possible sources of error; two main areas were of concern the generation logic behind control signals and the shift registers responsible for shifting the multiplier. The former was tested by several test benches where test cases and corner cases were developed to view how the module behaved.

Furthermore, the logic generating each signal was reviewed and compared against the block diagram created alongside the initial design structure agreed upon. As for the latter, modules that included shift registers were reviewed to ensure the correct inputs were relayed alongside verifying the module's logic. As a result, the error was identified and patched to match the expected output.

Another obstacle faced was related to the display driver modules and displaying the multiplier result on the seven-segment display. In this implementation stage, test benches would simulate the desired output and behaviour expected from the modules. However, when applying this on the FPGA, the seven-segment display would not reflect the results seen on the test benches. In order to address this issue, the top module for the display driver was tested alongside the instantiations of other modules declared within. This would then identify the error's stage, allowing us to address it accordingly. While debugging each module, the error was pinpointed as a communication issue between the instantiated modules. Consequently, the required modifications were made, and the display driver was tested later on as a whole to ensure an accurate result.

Lastly, the double dabble algorithm responsible for converting the binary product to a BCD value presented a challenge in its implementation. Initially, the algorithm was designed as a sequential circuit alongside combinational logic. However, implementing this design proved to be challenging not only as a Verilog module but also as physical hardware on logisim. While translating the algorithm's behaviour on logisim, the circuit grew substantially large and complex, making the debugging process and pinpointing logical errors difficult. Furthermore, due to how the algorithm operates, syncing the double dabble circuit with the rest of the multiplier gave rise to several other problems. Consequently, a combinational approach was favoured due to the simpler circuit design and less challenging debugging phase. This was also reflected in the Verilog module and the logic on which it operates.

Validation activities:

This section highlights several verification and validation techniques applied throughout the development of the sequential multiplier to ensure that the circuit modelled the required behaviour and displayed accurate results.

While translating the initial block diagram and design ideology into hardware on logisim, various approaches were employed to guarantee satisfying the requirements.

Building blocks, such as customised registers and adders to be used throughout the circuit, were tested individually to avoid errors later in the development stage. Additionally, each major component of the whole circuit was tested separately to verify its functionality and outputs. This was applied when testing the control unit, display driver, and other significant circuit components. Furthermore, the logic behind each major component and output was reviewed to ensure functionality and optimal implementation, as this would then reduce the cost of implementing the multiplier practically while simultaneously allowing errors and issues to be identified and handled efficiently. Applying said approaches in the development process ensured a stable rate of progress and accurate final results.

On the other hand, in the second primary stage of development (translating the sequential multiplier to Verilog), similar approaches were used to validate the multiplier. This included having test benches to validate the output of each module and test for corner cases that could potentially arise. Additionally, every major milestone would be accompanied by an implementation on the FPGA board to ensure the desired output was observed and the module's logic was valid. Moreover, the code was formatted to ensure ease of readability and member collaboration, Preventing errors and speeding the debugging phase.