ELEC6233 FPGA Implementation of a Complex Number Multiplier

Joshua Tyler
jlt1e16
MSc Embedded Systems
Dr Jeff Reeve

Abstract

1 Introduction

The aim of this coursework assignment is to design, and implement using SystemVerilog, a multiplier of two complex numbers.

The objectives are:

1. Design of a module capable of multiplying two complex numbers.
2. Implementation of the designed module using SystemVerilog.
3. Verification of the SystemVerilog model, by simulation.
4. Validation of the synthesised design

In order to effectively implement and demonstrate the multiplier, a state machine is also required which reads the desired inputs switches, and displays the result on light emitting diodes (LEDs) on a field programmable gate array (FPGA) development board.

In addition to the base specification, an additional SystemVerilog module was written which decodes the value displayed on the switches and LEDs and displays it in decimal using 7-segment displays. It should be noted that this module was previously created by the author for a different piece of coursework, and so has been adapted for this project, but not written from scratch [1].

It has been decided to display block diagrams, rather than register transfer level (RTL) schematics in this report, as they convey the same information, whilst being far more comprehensible.

This report contains timing diagrams which have been exported from ModelSim, and converted to a Ti*k*Z timing waveform using modelsim2latex [2]. Small modifications were made to the program to ensure compatibility with the exported waveforms. In each case, the waveform rendered by Ti*k*Z has been checked against the displayed result in ModelSim, to ensure correct operation of the program.

2 Design

2.1 Overview

Figure 1 shows the overall block diagram of the implemented design. The central block is the main state machine of the design. This state machine is responsible for reading the desired input values from switches, storing them in registers, and displaying the calculated result on LEDs.

It should be noted that the most significant two switches are debounced before being presented to the state machine. The reason for this is that these two switches are responsible for signalling when to load each input word, and when to display each output word (referred to as the 'handshake' switch within this report), and the asynchronous reset of the system. The other switches are only clocked on transition of the handshake switch, and so it does not matter if they bounce, provided they have settled when the handshake switch is transitioned.

The multiplier is purely combinational and calculates the result of multiplication of the input words presented to it by the state machine. The state machine then multiplexes the relevant output word to the output.

It should be noted that both the input and output words are truncated from what the multiplier is capable of. This is discussed further in section 2.2.

The two display decoders take an 8-bit signed input word each, and display the result on seven segment displays.

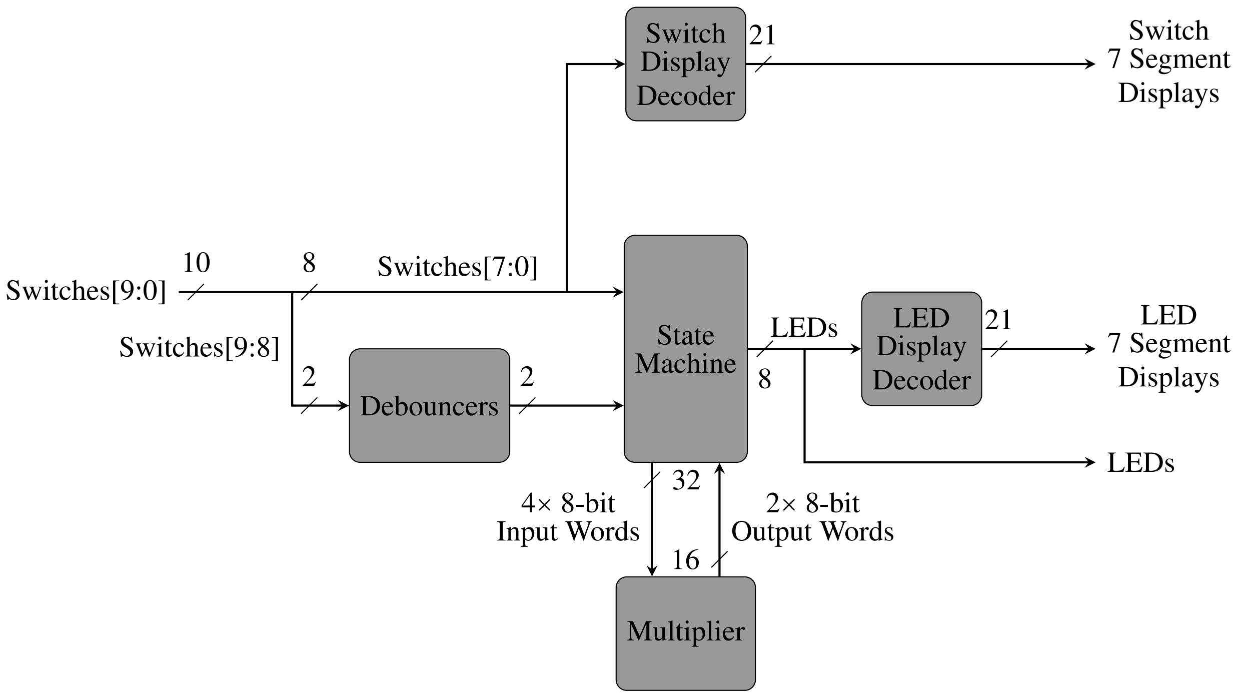

Figure 1: Overall Block Diagram

2.2 Complex Multiplier

A block diagram of the complex Multiplier is shown in Figure 2. This is the traditional architecture for multiplication of complex numbers, and contains four n-bit multipliers, a $2n$-bit adder, and a $2n$-bit subtracter. This architecture comes from expanding the multiplication of two complex numbers $x = a + jb$, and $y = c + jd$, as shown by Equation 1.

$$\begin{aligned} z &= (a+jb)(c+jd) \\ &= ac + jad + jbc - bd \\ &= (ac - bd) + j(ad + bc) \end{aligned} \tag{1}$$

An alternative architecture is possible, which requires, two n-bit adders, an n-bit subtracter, three n-bit multipliers, a $2n$-bit adder, and a $2n$-bit subtracter. This architecture is often preferred if implementing a multiplier from logic elements, as a multiplier would use many more logic elements than an adder and subtracter. The Cyclone IV FPGA that is used for this implementation has many hardware multipliers, which cannot be reconfigured to another purpose if not used [3, p.4-1]. Therefore it is advantageous to make use of them, and save the logic elements which would be required for the adder and subtracter.

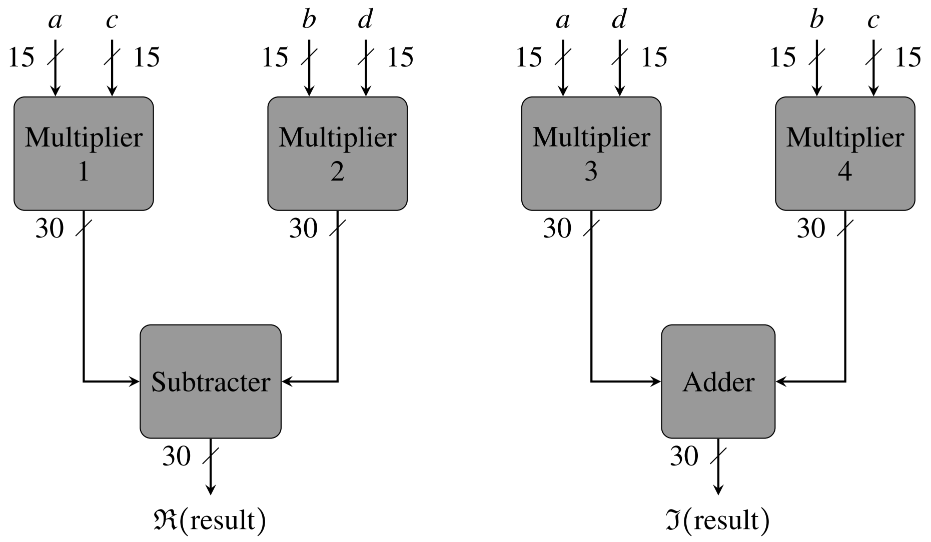

Figure 2: Multiplier architecture

It should be noted that the inputs to the multiplier are 15-bits wide, which is far greater than the 8-bit words used by the rest of the system. This is necessary because the binary values of the two input words defined in the specification α and q, have different bit significances to each other [4]. The real and imaginary parts of q use the standard 8-bit signed integer representation, giving input range -128 – 127, and having bit significances $-2^7, 2^6, ..., 2^0$. α, however, uses a fixed point representation with bit significances $-2^0, 2^{-1}, ..., 2^{-7}$, giving an input range -1 – $[1-2^{-7}]$.

The multiplier input words are thus designed to be wide enough to accept both these representations. Since the 2^0 bit is present in both, 15 bits are necessary in total. The q numbers are right padded with 7 0s, and the α numbers are sign extended.

2.3 State Machine

A diagram showing the operation of the main state machine is shown in Figure 3. The four states in the top row of the diagram read the four input numbers. Only one state is required for each of these because, whilst reading the inputs is a more complex procudure, a second state machine handles this, and signals to the main state machine when to read the input number and when to transition to the next state. `read_done` is the signal which tells the state machine to transition to the next state whilst reading.

The other two states display the real and imaginary parts of the result. The real part is displayed first then, once the handshake switch goes high, the imaginary part is displayed. This is governed by a multiplexer whose select input is the state of the multiplexer.

Figure 4 shows the read state machine. In the `HALT` state the state machine waits for a the main state machine to be in a read state.

On the transition from `WAIT_1` to `WAIT_0`, the state machine asserts a `read` signal for a single clock cycle. This tells the main state machine to sample the data on the switches into an input register. When the read state machine is in the `DONE` state, the `done` signal is asserted which triggers the transition of the main state machine.

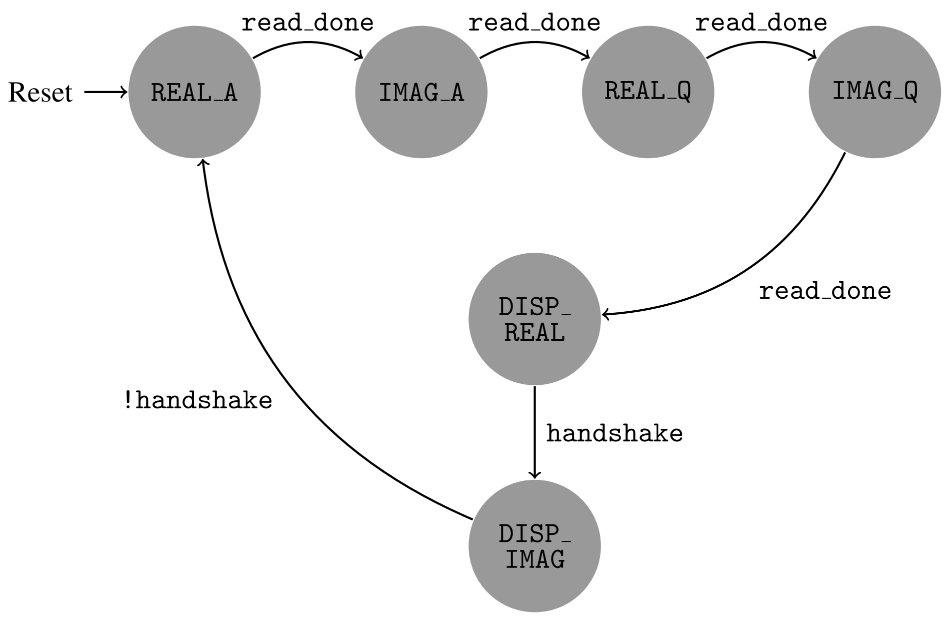

Figure 3: Main state machine state transition diagram

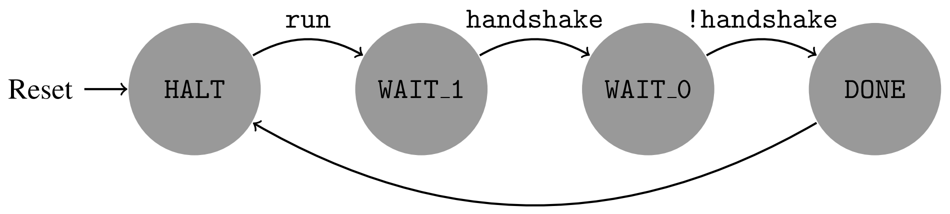

Figure 4: Read state machine state transition diagram

2.4 Debouncer

The design of the debouncer is relatively simple. It consists of a free-running counter (that is initialised to zero by the bitstream). When the counter reaches it's maximum count (which is determined at compile time from the desired debounce time and clock period, passed to the module as parameters), the input signal is propagated to the output, and the counter is reset. However the module stores the previous value of the input on each clock edge, and if this changes between cycles, the counter is reset.

The effect of this is that the input is only allowed to pass to the output once it has been stable for the time taken for a full run of the counter. This has the effect of suppressing all fast transitions called by switch bounce.

A settling time of 20 ms is used in the synthesis of the debounce module.

3 Verification

This section, briefly, presents the verification results of each module, as well as from system level testing. It should be noted that for some of the waveforms, the `clk` signal is displayed as ■. This is because it has too many transitions to accurately display.

3.1 Multiplier

In order to test the multiplier, a set of test vectors were generated using the fixed-point library in matrix laboratory (MATLAB), and exported to a text file. This is discussed in Appendix A.

Listing 1 shows the stimulus used, which loads all of the vectors from the text file. The output waveform, Figure 5 shows the response of the multiplier to each set of inputs. This is verified using assertions in the `testNums` task.

Listing 1: `test_mult.sv` Stimulus

```
initial
begin

    file = $fopen(FILENAME,"r");
    while($fscanf(file,"%b, %b, %b, %b, %b, %b, %s\n", re_x_in, im_x_in,
       ↳ re_y_in, im_y_in, re_z_in, im_z_in, description) == 7)
    begin
        testNums(re_x_in, im_x_in, re_y_in, im_y_in, re_z_in, im_z_in);
    end
    $stop;
end
```

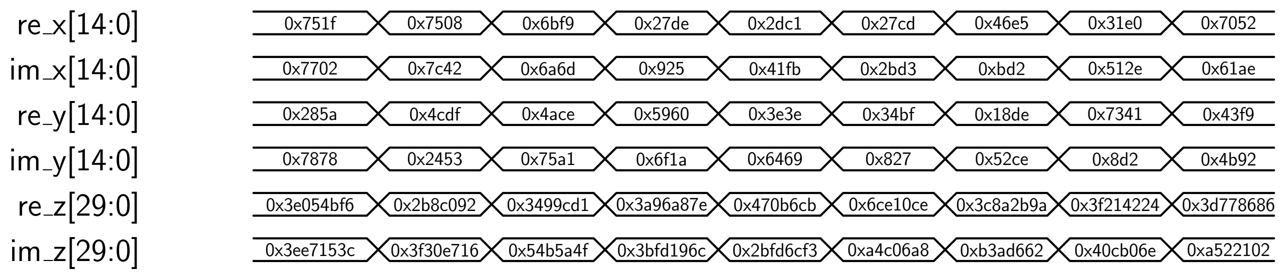

Figure 5: `test_mult.sv` Output

3.2 Main state Machine

In order to test the main state machine, dummy values of each input word, and the result from the multiplier are presented to the state machine at the relevant time, and the result is checked in each instance by assertion. For the simulation, the state machine was presented with the values $1,\ldots,6$. The stimulus for simulation is presented in Listing 2, and the resultant waveform is shown in Figure 6.

Listing 2: `test_sm.sv` Stimulus

```
initial
begin
    //Reset
    handshake = 0;
    data_in = 0;
    #32ns;

    testCycle(1,2,3,4,5,6);
    $stop;
end
```

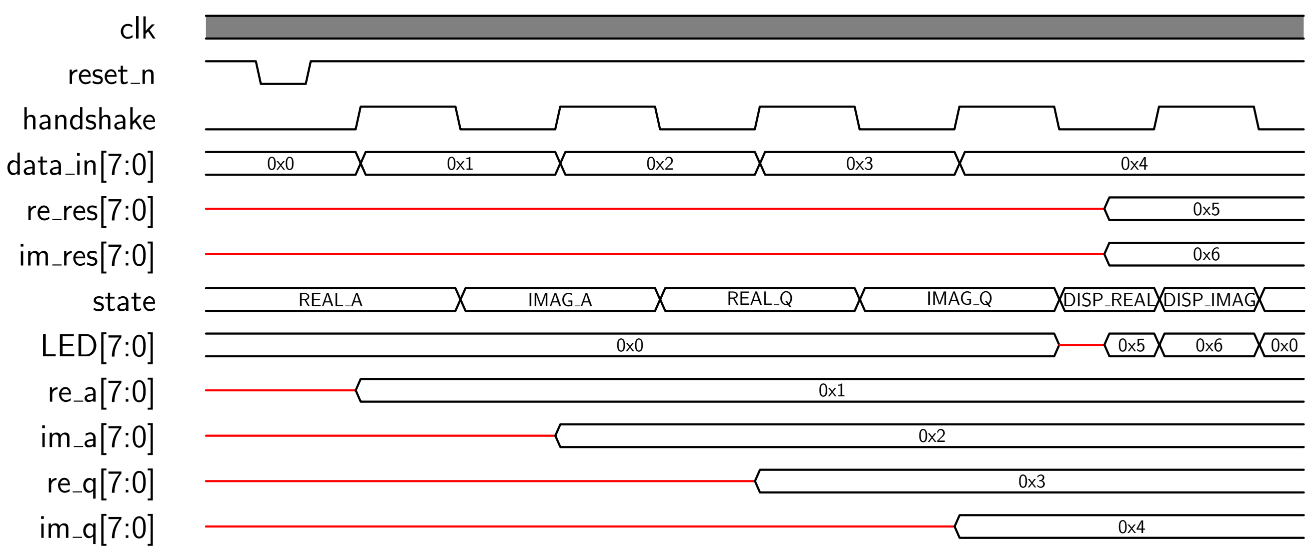

Figure 6: `test_sm.sv` Output

3.3 Read State Machine

Testing of the read state machine was performed by providing generating `run` and `handshake` signals, then testing by assertion that the `read` and `done` outputs performed as expected. The stimulus used, and resultant waveform is presented in Listing 3 and Figure 7 respectively.

Listing 3: `test_read_sm.sv` Stimulus

```
initial
begin
    handshake = 0;
    run = 0;
    #32ns;

    @(posedge clk);
    run = 1;
    @(posedge clk);
    run = 0;
    #30ns handshake = 1;
    @(posedge clk) assert(read);
    #50ns handshake = 0;
    @(posedge clk) assert(done);
    #50ns;
    $stop;
end
```

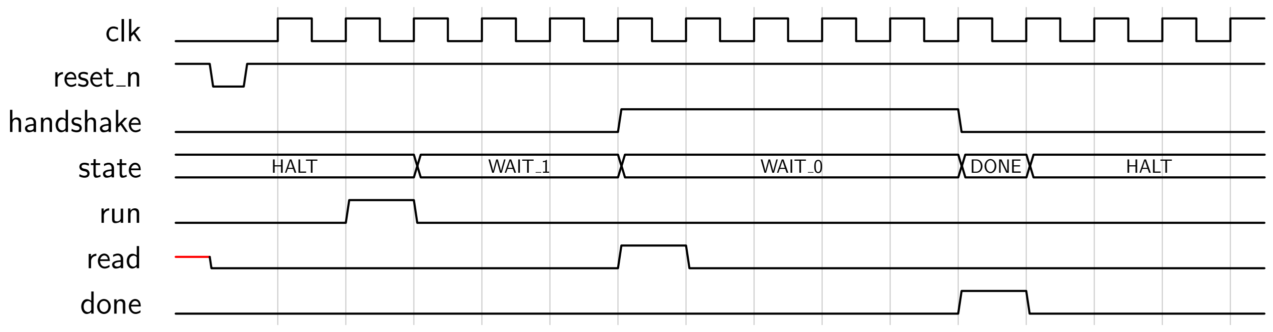

Figure 7: `test_read_sm.sv` Output

3.4 Debouncer

To test the debouncer, the module was parametrised to wait for 100 ns, then an oscillating input stimulus was presented, and the output was verified by assertion. The stimulus used, and resultant waveform is presented in Listing 4 and Figure 8 respectively.

Listing 4: `test_debounce.sv` Stimulus

```
initial
begin
    signal_in = 0;
    #110ns assert(signal_out == signal_in);

    signal_in = 1;
    #20ns assert(signal_out != signal_in);
    signal_in = 0;
    #10ns signal_in = 1;
    #110ns assert(signal_out == signal_in);
    #50ns;
    $stop;
end
```

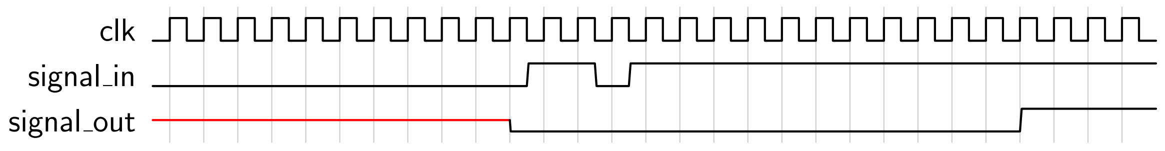

Figure 8: `test_debounce.sv` Output

3.5 BCD Decoder

Testing of the BCD decoder consists of looping through all possible input values and observing the outputs, this is conducted with the stimulus of Listing 5, and the output waveform, Figure 9, for the first two numbers −128 and −127 shows the numbers correctly decoded.

Listing 5: `test_bin_to_bcd.sv` Stimulus

```
initial
begin
    for(int i = -128; i < 128; i++)
    begin
        in = i;
        # 10ns;
    end
    $stop;
end
```

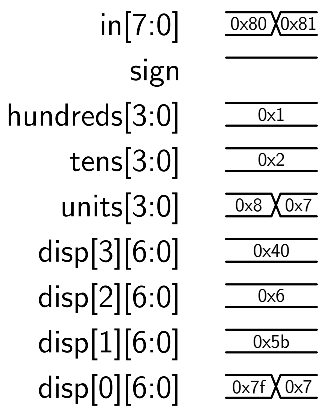

Figure 9: `test_bin_to_bcd.sv` Output

3.6 System Level

The system level testing is conducted in a similar manner to the testing of the multiplier described in Section 3.1. A set of test vectors were generated using the fixed-point library in MATLAB, and exported to a text file. This is discussed in Appendix A. The test vectors are asserted on the input bus, and the handshake switch is toggled as appropriate, the outputs are then verified as correct using assertions.

Listing 6 shows the stimulus used, and Figure 10 shows the output. It can be seen that the correct result is asserted on the outputs at the correct time. It should be noted that the small delay between the transition of handshake and the transition of the LEDs is due to the debouncing of the switches.

Listing 6: `test_cmplx_mult.sv` Stimulus

```
initial
begin
    //Reset
    handshake = 0;
    data_in = 0;
    #`SW_SIMULATION_DELAY;
    #`SW_SIMULATION_DELAY;

    file = $fopen(FILENAME,"r");
    while($fscanf(file,"%b, %b, %b, %b, %b, %b, %s\n", re_a, im_a, re_q,
       ↳ im_q, re_res, im_res, description) == 7)
    begin
        testCycle(re_a, im_a, re_q, im_q, re_res, im_res);
//      assert(0) else $fatal;
    end
    $stop;
end
```

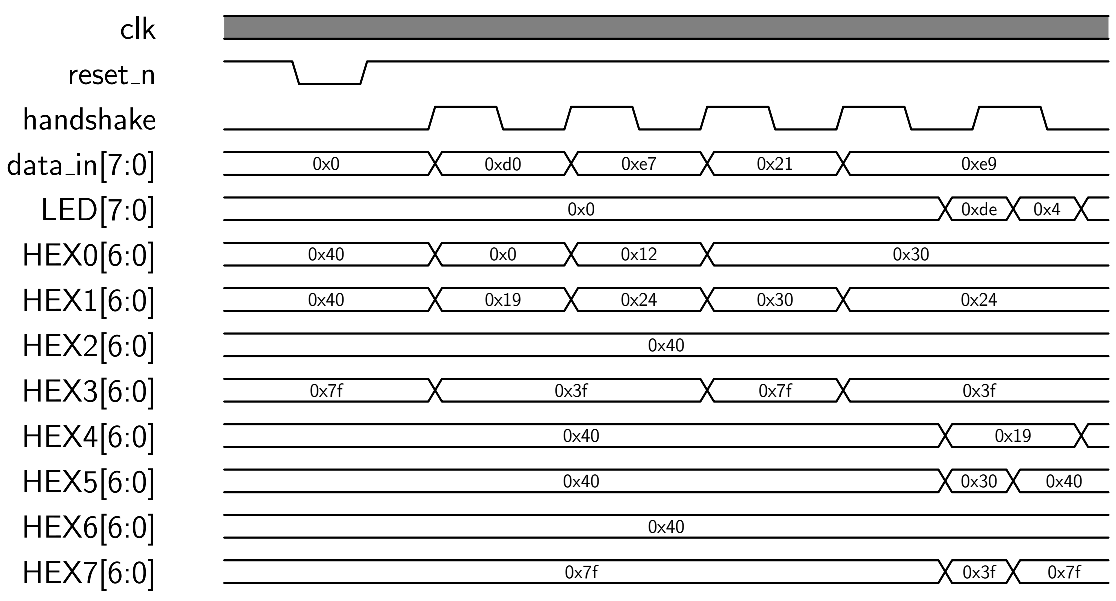

Figure 10: `test_cmplx_mult.sv` Output

4 Synthesis

The SystemVerilog module was successfully synthesised using Quartus, and the resource utilisation summary is listed in Listing 7. It should be noted however that much of this cost figure is consumed by the binary to decimal decoders, which do not form part of the core specification.

Listing 7: Synthesis Result

```
Total logic elements : 409 / 114,480 ( < 1 % )
    Total combinational functions : 393 / 114,480 ( < 1 % )
    Dedicated logic registers : 90 / 114,480 ( < 1 % )
Total registers : 90
Total pins : 75 / 529 ( 14 % )
Total virtual pins : 0
Total memory bits : 0 / 3,981,312 ( 0 % )
Embedded Multiplier 9-bit elements : 8 / 532 ( 2 % )
Total PLLs : 0 / 4 ( 0 % )
```

No significant problems were encountered when testing the implemented design using the FPGA developement board, and the validation strategy was to use test vectors generated by MATLAB and compare the result presented on the LEDs against the result calculated by MATLAB.

5 Conclusion

In conclusion, all the objectives outlined in Section 1 have been achieved. Design of the system has been discussed in Section 2, this was then verified and implemented in hardware successfully as discussed in Sections 3 and 4 respectively. In addition the project has been extended by adding a decimal readout of the switches and result.

The project has therefore been a success, and it is difficult to imagine how it could be improved without modification to the specification.

References

[1] J. Tyler, “ELEC6234 FPGA synthesis of a picoMIPS processor,” may 2017.

[2] sh-ow. (2016, jan) modelsim2latex. [Online]. Available: https://github.com/sh-ow/modelsim2latex/

[3] Altera, *Cyclone IV Device Handbook*. Altera, mar 2016, vol. 1.

[4] T. J. Kazmierski, “FPGA implementation of a complex number multiplier,” mar 2017.

Appendix A MATLAB Test Vector Generation

This Appendix briefly describes the use of MATLAB to generate the test vectors for the multiplier testing, and overall system testing.

Listing 8 shows the script used to generate the overall system tests. `generate_fp_num` (source listed in Listing 9) is a function which generates a random 15-bit fixed point number, either in the format of α, q, or using all the bits. The type of number generated depends on the input argument.

The generated numbers are then paired to from complex numbers, and the two resultant complex numbers are multiplied.

The relevant bits of the result are then extracted using `extract_result` (source listed in Listing 10), and the input and output numbers are logged to a text file. A sample output file is listed in Listing 11.

A very similar process is followed to generate tests for just the multiplier, except 9 is configured to generate a purely random 15-bit number, and the full significance of the result is used.

Listing 8: `generate_mult_tests.m`

```matlab
%Setup fixed point math settings
fimath('OverflowAction', 'Wrap', 'RoundingMethod','Zero');

%a has range -1 ... +1-2^(-7)
%q has range -128 ... 127

file = fopen('mult_tests.txt','w');

for i = 1:10
    [a_re, a_re_bits] = generate_fp_num('a');
    [a_im, a_im_bits] = generate_fp_num('a');

    [q_re, q_re_bits] = generate_fp_num('q');
    [q_im, q_im_bits] = generate_fp_num('q');

    a = complex(a_re, a_im);
    q =  complex(q_re, q_im);

    result = a*q;

    [res_re_bits, res_im_bits] = extract_result(result);

    %Format : a_real, a_imag, q_real, q_imag, res_real, res_imag
    fprintf(file, '%s, %s, %s, %s, %s, %s,%s\n', a_re_bits, a_im_bits,
        ↪ q_re_bits, q_im_bits, res_re_bits, res_im_bits, ['(', num2str(a)
        ↪ , ')*(',num2str(q),')=(',num2str(result),')']);
end

fclose(file);
```

Listing 9: `generate_fp_num.m`

```matlab
function [ num, num_bits ] = generate_fp_num( type )
%a has range -1 ... +1-2^(-7)
%q has range -128 ... 127

%Therefore for a
% [-2^0] [2^-1] [2^-2] [2^-3] [2^-4] [2^-5] [2^-6] [2^-7]
```

```

%For q
% [-2^7] [2^6] [2^5] [2^4] [2^3] [2^2] [2^1] [2^0]

%Therefore combined. 15 word bits, 7 fraction bits

%Setup generating function
length = 15;
frac_length = 7;
fixed = @(x) fi(x,true, length, frac_length);

assert(type == 'a' || type == 'q' || type == 'r');

if(type == 'r') %Purely random 15 bit number
    num = fixed(0);
    num_bits = dec2bin(randi(intmax('uint16'), 'uint16'),16);
    assert(size(num_bits,2) == 16);

    num_bits = num_bits(2:end); %Truncate to 15 bits

    num.bin = num_bits;

else %Random number confroming to either 'a' or 'q'

    %Generate a number and set the correct 8 bits to a random bit pattern
    num = fixed(0);
    rnd = dec2bin(randi(intmax('uint8'), 'uint8'),8);
    assert(size(rnd,2) == 8);

    num_bits = rnd;

    if(type == 'q')
        rnd = [rnd , '0000000']; %Append 7 blank bits to set range
    else
        if(rnd(1) == '0') %Append either 0s or 1s depending on sign
            rnd = ['0000000' , rnd];
        else
            rnd = ['1111111', rnd];
        end
    end
    assert(size(rnd,2) == 15);

    num.bin = rnd;

    if(type == 'q')
        assert(num <= 127);
        assert(num >= -128);
    else
        assert(num <= (1-2^(-7)));
        assert(num >= -1);
    end

end

```

```
end
```

Listing 10: extract_result.m

```
function [ real_num , imag_num ] = extract_result( num )
% Extract the bits we want to keep from the result
assert(num.WordLength == 31);
assert(num.FractionLength == 14);

real_num = real(num);
imag_num = imag(num);

real_num = real_num.bin;
imag_num = imag_num.bin;


%31 bits total. 14 fraction bits.
%Therefore bits 18-31 are fraction
%Therefore we want to keep 10-18
real_num = real_num(11:18);
imag_num = imag_num(11:18);
end
```

Listing 11: Sample generated tests

```
11010000, 11100111, 00100001, 11101001, 11011110, 00000100,(-0.375-0.19531
    ↪ i)*(33-23i)=(-16.8672+2.17969i)
10100010, 00011001, 01001000, 10001100, 11000011,
    ↪ 11000110,(-0.73438+0.19531i)*(72-116i)=(-30.21875+99.25i)
11110101, 11110111, 00101001, 11111000, 11110111,
    ↪ 11111011,(-0.085938-0.070312i)*(41-8i)=(-4.0859-2.1953i)
11110101, 01111100, 11001101, 00100101, 11000001,
    ↪ 10010110,(-0.085938+0.96875i)*(-51+37i)=(-31.4609-52.5859i)
01101100, 11101010, 11001011, 11110101, 10100010,
    ↪ 11111111,(0.84375-0.17188i)*(-53-11i)=(-46.6094-0.171875i)
10101000, 00001010, 11011001, 11101111, 00111000,
    ↪ 00010001,(-0.6875+0.078125i)*(-39-17i)=(28.1406+8.64062i)
10101110, 11000010, 10111110, 01100101, 10110110,
    ↪ 10111110,(-0.64062-0.48438i)*(-66+101i)=(91.2031-32.7344i)
10101000, 00101100, 10110101, 00001001, 01100000,
    ↪ 11000000,(-0.6875+0.34375i)*(-75+9i)=(48.4688-31.9688i)
01000111, 00001100, 00011001, 11010010, 00100100,
    ↪ 11010001,(0.55469+0.09375i)*(25-46i)=(18.1797-23.1719i)
10110010, 01010001, 11110011, 00001001, 00000100,
    ↪ 11100100,(-0.60938+0.63281i)*(-13+9i)=(2.22656-13.7109i)
```