Lab 2: OpenCL Libraries

ENGINEERING A RIPPLE CARRY ADDER

February 26, 2018

Richard Carter
Clemson University
Department of Electrical and Computer Engineering
rcarte4@clemson.edu

1 Abstract

OpenCL contains libraries that enable the compilation of VHDL modules that are executable in an OpenCL environment. The goal of this project was to use these libraries to compile, test, and run a 16-bit ripple carry adder in an OpenCL environment. The project was fully completed with a fully functional 16-bit ripple carry adder implemented.

2 Introduction

The approach to constructing our 16-bit ripple carry adder was to first construct a full adder module that we would later generate multiple copies of. Both this full adder module and the final 16-bit ripple carry adder were to be tested against multiple test cases to guarantee functionality. Once functionality was guaranteed, we were to instantiate our ripple adder inside a VHDL template that complied with the OpenCL libraries we were to use in our compilation. The compiled ripple adder was then to be moved to a Micro SD card containing a bootable Linux environment that would be booted by our DE1 board. The rest of the report details how we conducted all of the above steps.

3 DESIGN

3.1 Full Adder

For this project, we were provided with a full adder circuit diagram, shown in Figure 1.

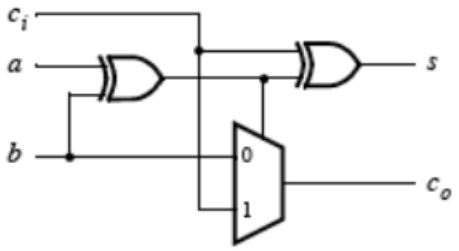

Figure 1: Full Adder Circuit Diagram

Using boolean logic matching the diagram, we were able to model a full adder.

```
s <= ci xor (a xor b);

co <= ci when (a xor b) = '1' else
          b when (a xor b) = '0';
```

3.2 16-bit Ripple Carry Adder

By generating 16 of the full adder modules, we were able to create a new entity capable of taking in two separate 16-bit inputs for addition, along with a carry-in bit. The output of our new entity would have a 16-bit result output as well, with an accompanying carry-out overflow bit. Carry-in and carry-out bits between the generated full adders were linked together in order to implement the ripple carry functionality. A small scale visualization of this is shown in Figure 2.

This attachment was accomplished by using an intermediary signal that would allow for newly generated full adders to attach their carry-in port to the previous carry-out

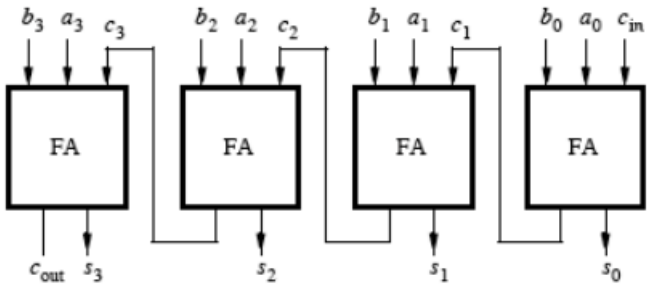

Figure 2: Example of Carry-out and Carry-in Attachment

port, and then subsequently attach their carry-out port to a future carry-in port. A portion of the code for this is shown below.

```
architecture behav of ripple_carry_adder is
        signal carry_through : std_logic_vector(N downto 0) ;
begin
        .....

        fadd_group : for i in 0 to N-1 generate
                fadd : entity work.adder
                        port map (
                                ci => carry_through(i),
                                co => carry_through(i+1)
        .....
```

It should be noted that the initial carry-in bit and the final carry-out bit must be treated as special cases, as these are used in the main ports of the ripple adder entity.

4 TESTING

4.1 Full Adder

The testing of the full adder was simple, as there were only a few possible inputs and outputs to test against. These were given to us in the form of a truth table, which is shown in Figure 3.

Using this truth table, we created a test bench that tested all cases. The resulting waveform of our testing is shown in Figure 4. From analysis of the waveform, we were able to identify that our full adder was working completely as intended.

4.2 16-bit Ripple Carry Adder

We could not test all cases of our ripple adder, as this would be a very large number of test cases. Therefore, we tested all fringe cases (such as overflow cases and "0" input

b	a	c_i	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 3: Full Adder Truth Table

cases), as well as some basic functionality cases. The testbench code for generating fringe cases is shown below.

```
edge_cases : for i in 0 to 1 loop
        if i = 1 then
                ci <= '1';
        else
                ci <= '0';
        end if ;

        A <= (others => '0');
        B <= (others => '0');
        wait for 10 ns;

        A <= (others => '1');
        wait for 10 ns;

        B <= (others => '1');
        wait for 10 ns;

        B <= (others => '0');
        wait for 10 ns;
end loop ; -- edge_cases
```

5 Conclusion

All-in-all, our project turned out exactly as we had hoped and planned. One lesson that we did learn was to take precaution when generating test benches. One small syntax slip-up could cause you to think that your overall top-level entity isn't functioning correctly.

References

6 Appendix

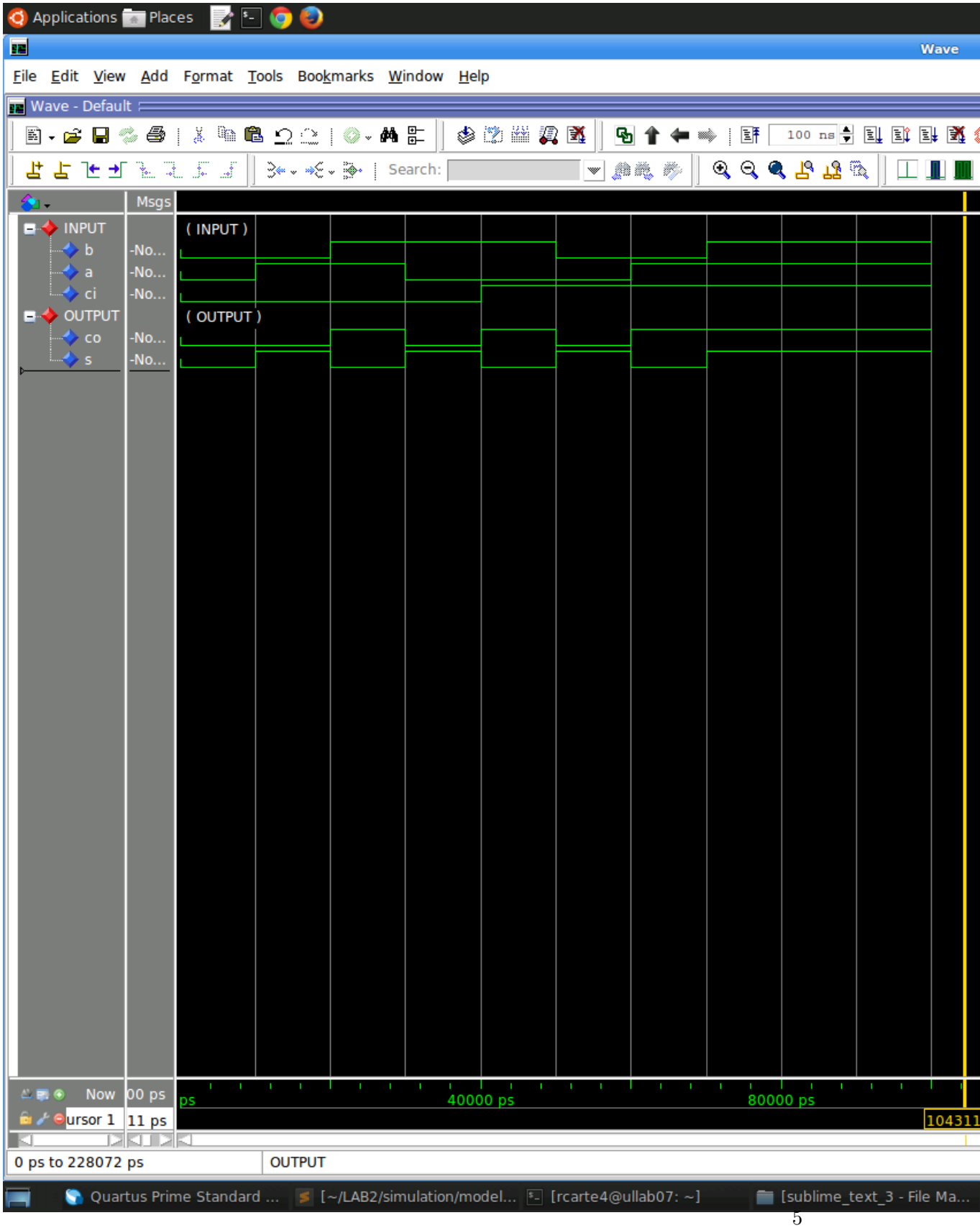

Figure 4: Waveform Result of Full Adder Testbench

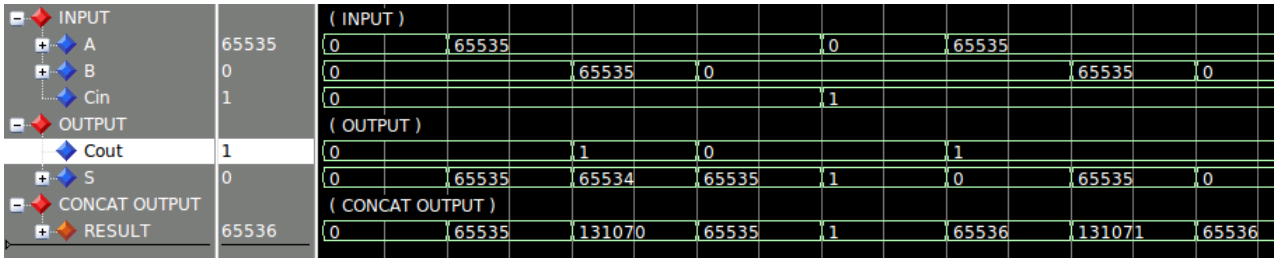

Figure 5: Testbench Waveform for RCA Part 1

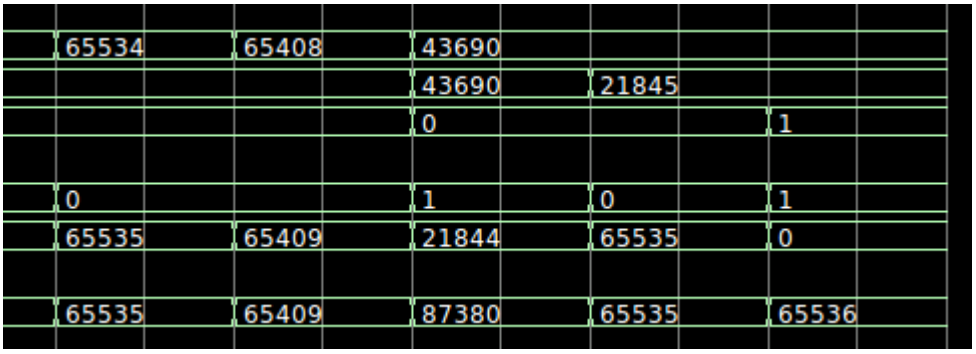

Figure 6: Testbench Waveform for RCA Part 2