CSSE4010 A3

David Gaul

s4671313

Lab Session: Wednesday 10-12

Submission Date: 5th-ish September

1 Introduction

There were two separate tasks that were to be fulfilled for prac 3. These tasks were:

- To create a 16-bit two's comp adder. This adder has to take two 16-digit inputs, as well as a carry-in. Depending on whether saturation was enabled or not, the output would either wrap around or set the two's comp ZNCV flags.
- To create a two-digit BCD adder. This process had to be complete by first creating a 1-digit BCD adder, before cascading it into the second.

Ready for another fun filled report???

LET'S GOOOOOOOO

2 Block Diagram

When discussing the project architecture, it's important to talk about the basics. To begin, this is an 8-bit register:

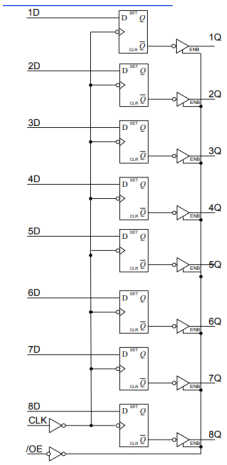


Figure 1: Pictorial depiction of an 8-bit register

And this is the complete prac 3A architecture:

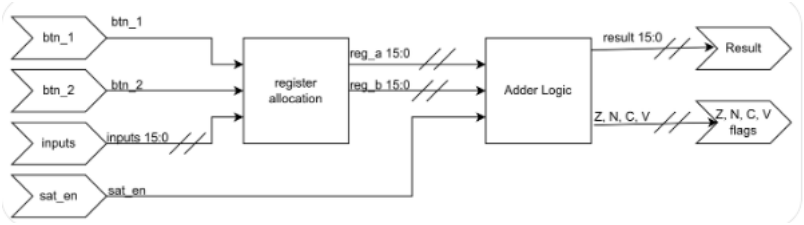


Figure 2: Completed block diagram for part A

Confused? So am I.

The entire design was implemented in such a way that it could be tested on the Nexys 4 boards (And it eventually was, successfully, but the full integration is outside the scope of this assignment, and details are not included). The register allocation system received a 16 bit input line, as well as two buttons. Depending on which of these buttons was pressed, the 16-bit input buffer was saved into either register A or B. These registers are then added together within the adder logic, which returns the result of the addition along with any flag values. The rules of the flag values are as follows:

If saturation is enabled:

- If a positive overflow occurs, 0x7FFF is returned, and the overflow flag is set high
- If a negative overflow occurs, 0x8000 is returned. The overflow flag is once again set high
- If the result is negative, the negative flag is set high
- If there is a carry out bit, the carry flag is set high
- If the result is 0, the zero flag is set high

If saturation is not enabled:

- The carry out flag will be set high when there is a carry out bit

The architecture of part 3B was slightly more involved. It required the use of clock dividers, two cascaded BCD adders and a finite state machine which finally led into the seven-seg decoder. The fully architecture of this can be found within figure 3

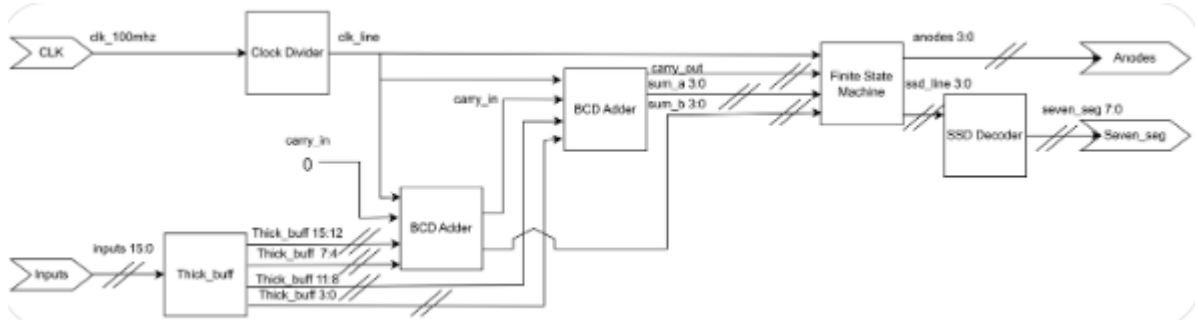


Figure 3: Completed block diagram for part B

The clock divider has been included to help with synchronicity of the design. This was integral for use in the BCD adders, as well as the finite state machine, which controlled the outputs of the seven-seg display.

Two cascaded BCD adders have been included. The input into each of the registers of the BCD adders is capped at nine. That is to say, each of the registers for the BCD adder have the following logic:

```
if input > 9:
    output <= 0;
else:
    output <= input;
```

And the logic for a single BCD adder is:

```
if (carry_in = '1') then
    if (a_line + b_line + 1 < 10) then
        sum <= a_line + b_line + 1;
        carry_out <= '0';
    else
        sum <= a_line + b_line - 9;
        carry_out <= '1';
    end if;
else
    if (a_line + b_line < 10) then
        sum <= a_line + b_line;
        carry_out <= '0';
    else
        sum <= a_line + b_line - 10;
        carry_out <= '1';
    end if;
end if;
```

Essentially meaning that the output of the BCD adder will be the sum of the input registers plus the carry-in bit. If an overflow occurs, the carry-out bit will be set high and the result will be wrapped around by 10 (Which will display the correct output on the seven seg. This is essentially the equivalent of adding 6 to the design, as is recommended within the task sheet).

The carry-in to the first BCD adder is set to 0, as it is not required. The carry-out for the first adder connects to the carry-in for the second adder. In creating the adders in this fashion, it was possible to cascade the same design multiple times.

Finally, the finite state machine was used to output the result of the BCD addition onto the seven-seg display. Alternating on every rising clock edge, the FSM would output one of the three digits of the result onto the seven-seg display. The logic for this is as follows:

```
if rising_edge(clk_line) then

    if (alternator = 0) then
        anodes <= "11111011";
        if (carry_line = '1') then
            ssd_line <= "0001";
        else
            ssd_line <= "0000";
        end if;
        alternator <= alternator + 1;
    elsif (alternator = 1) then
        anodes <= "11111101";
        ssd_line <= output_line(7 downto 4);
        alternator <= alternator + 1;
    else
        anodes <= "11111110";
        ssd_line <= output_line(3 downto 0);
        alternator <= 0;

    end if;

end if;
```

Cool clarkies. Hope that makes sense.

3 Simulation Results

Once again, simulation results are kept intentionally brief. In this instance, there are 2^{17} possible combinations, which is far too many to write about in this document. For this

reason, only key results have been included within the simulation. Full functionality can be verified during testing time.

For task 3A, the following tests were covered:

- Initialising the simulated inputs to zero
- Inputting values of 1 and 0 (Which results in an output of 1, and verifies that summing is working)
- Inputting values of 1 and 8 (Which results in an output of 9 and further verifies summing functionality)
- Inputting two values of 0x7FFF to trigger a positive overflow (Overflow flag should trigger)
- Inputting two values of 0x8001 to trigger a negative overflow (Overflow, negative and carry-out flags should trigger)
- Inputting values of 0xFFFF and 0, to result in -1 and trigger the negative flag
- Inputting values of 0xFFFF and 0x0001 to result in 0. This will also trigger the carry-out flag

These tests were carried out both with saturation enabled, as seen in figure 4, and without saturation enabled 5.

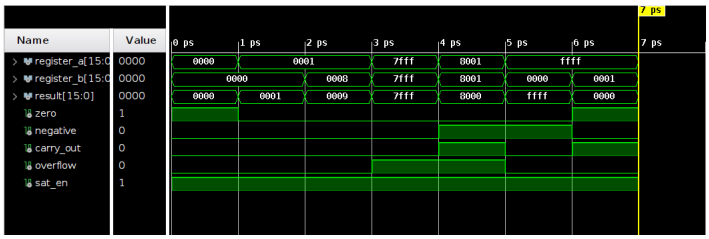


Figure 4: Simulation for part 3A with saturation enabled

As can be seen within figure 4, the test cases were able to be input and resolved correctly by the system.

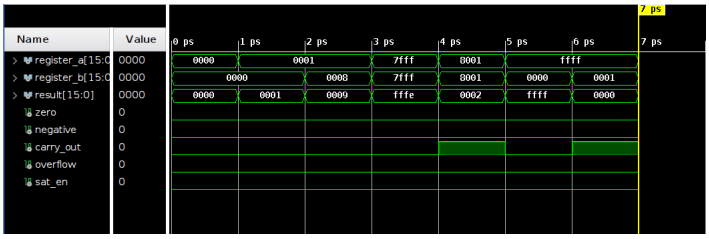


Figure 5: Simulation for part 3A with saturation disabled

As can be seen in figure 5, the product works exactly as it did in the first case, except zero, overflow and negative flags were disabled. Only the carry-out flag was enabled. Another note-worthy feature of this test case is that the system wraps around when overflow occurs. In test cases where saturation was implemented, the addition of two 0x7FFF numbers triggered saturation. Without saturation, however, the result was 0xFFFE, indicating that wrap-around took place and the result became negative. This was further reflected with negative wrap-around resulting in 0x0002.

In total, the system is able to receive inputs, and will output the correct results, depending upon whether the saturation pin has been triggered. But the important part is that full functionality has been achieved.

For part 3B, more testing was required. At first, it was necessary to test the functionality of a single BCD adder, to confirm that it functioned as expected. From here, both BCD adders were placed in cascade fashion, and its functionality was once again verified.

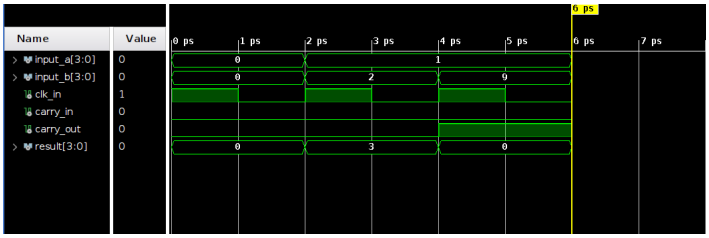


Figure 6: Simulation for a single BCD adder with no carry-in

Figure 6 shows the following test cases, with no carry-in enabled:

- Setting the inputs to 0, which results in an output of 0
- Setting the inputs to 1 and 2, which results in an output of 3
- Setting the inputs to 1 and 9, which triggers an overflow, setting the output to 0 and the carry-out bit to high

And as can be seen, the single adder performs as is expected. It's worth noting that neither of the inputs would exceed 9 during simulation. That is because a BCD adder does not take inputs of greater than 9. This test process was repeated with the carry-in bit enabled.

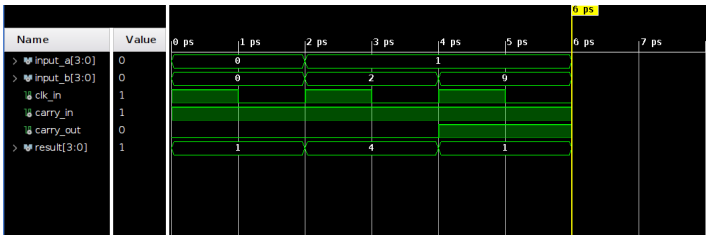


Figure 7: Simulation for a single BCD adder with carry-in enabled

As can be seen in figure 7, the outputs are identical to those in figure 6, save that they've been increased by 1 to fascilitate the carry-in bit.

So in total the single adder works perfectly. From here, two single-bit BCD adders were cascaded together to form the two-digit BCD adder. The simulation for this featured the following test cases:

- Inputting values of 0 into all inputs, which resets the ouptut to 0
- Inputting values of 2 and 1 into the lower bits, which results in an output of 0x03
- Inputting values of 9 and 1 into the lower bits. This triggers an overflow, and results in an ouput of 0x10
- Reseting the output to 0 by inputing 0x0000
- Inputting values of 2 and 1 into the higher bits, which results in an output of 0x30
- Inputting values of 9 and 1 into the lower bits. This triggers an overflow, and results in an ouput of 0x00, with the carry-out bit enabled

These test cases can be verified against the simulation shown in figure 8.

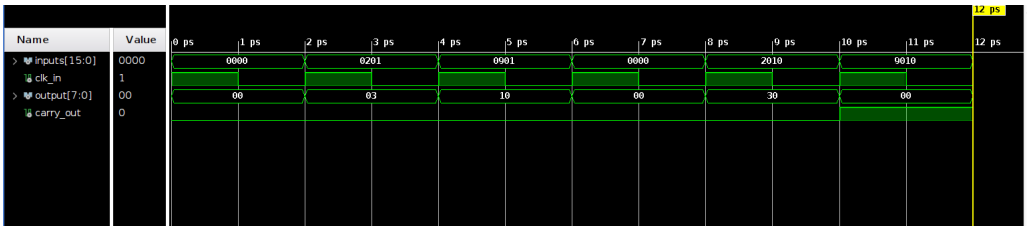


Figure 8: Simulation for the cascaded two-digit BCD

4 Results on the FPGA

To further verify the functionality of the design, it was uploaded onto the Nexys 4 FPGA. Several images were taken of the board operating as desired. Figure 9 shows the fpga with all switches disabled. Note that the output shown in this image is at 0x02. This is due to a broken pin on the board, not a design fault. Further information can be found on the Ed Discussion board about my broken FPGA.

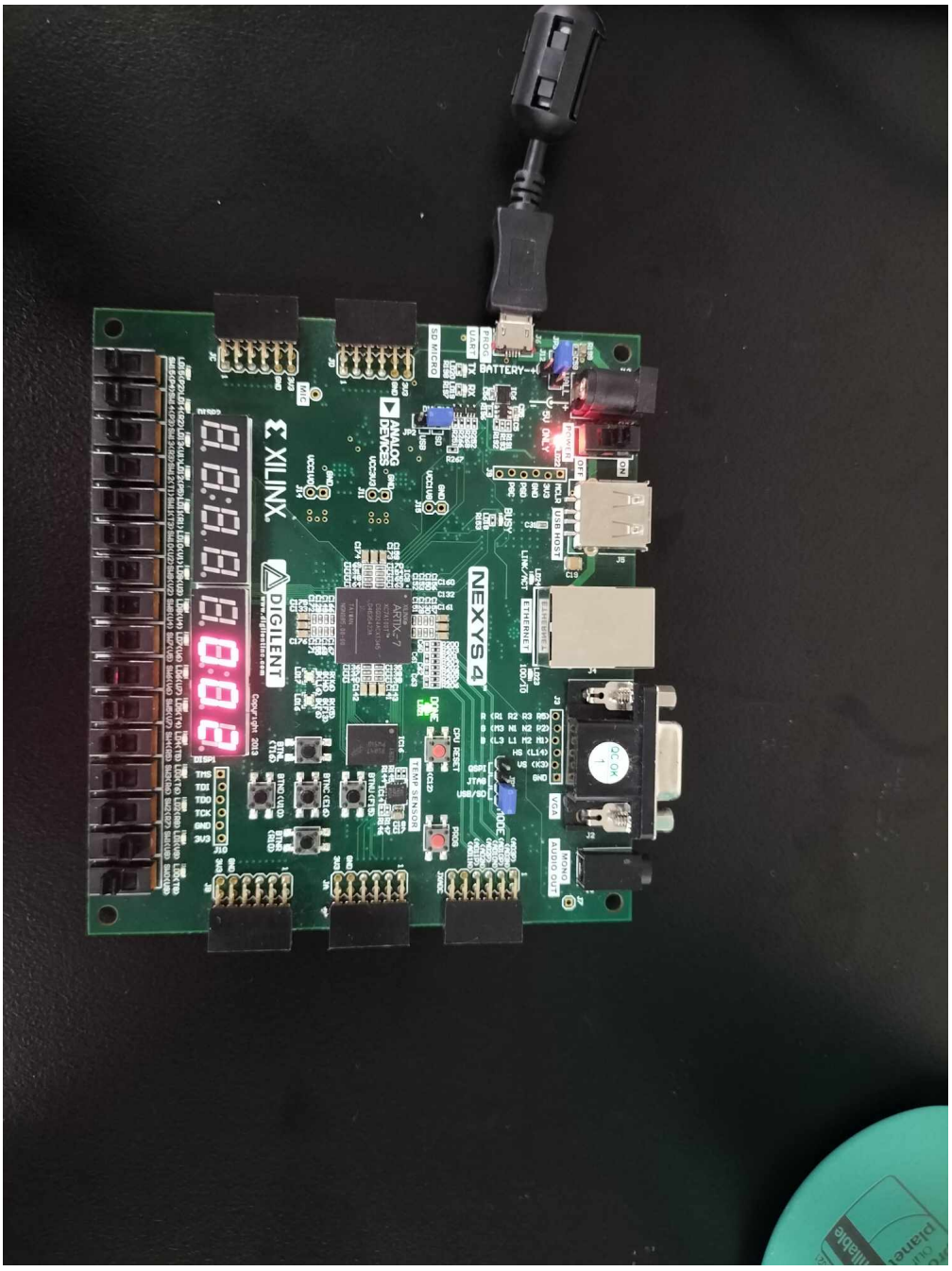


Figure 9: Base state for FPGA with no switches enabled. Note that an output of 2 is present due to a broken pin. This has been talked about on the Ed Discussion board

Secondly, by inputing values of 0x92 and 0x12, the board returned a result of 0x104, which is exactly as expected. This helped to verify the overflow functionality.

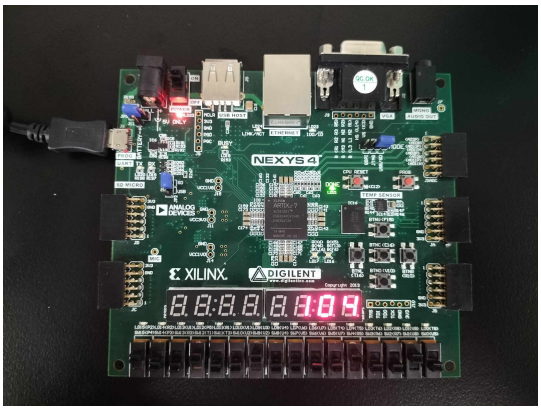


Figure 10: FPGA with an overflow on the higher digit, and a total of 4 on the lower

Figure 11 shows the final test case on the board, where 0x96 and 0x16 were input into the board, resulting in 0x112.

By this point, it can be safely verified that all major test cases have been satisfied and the board functions as expected.

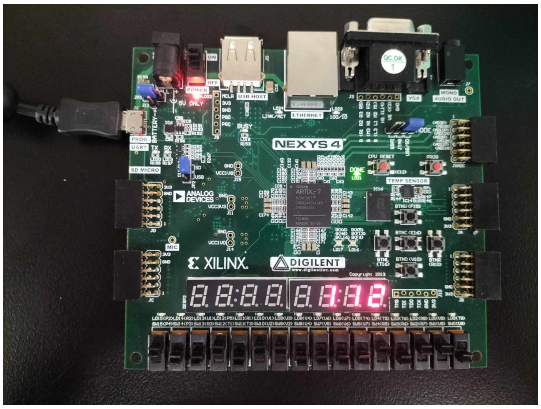


Figure 11: FPGA with an overflow on both the higher and lower digits

5 Register Transfer Level Schematic

Figure 12 shows the top level design of the RTL schematic. This is going to be explained in greater detail in this segment.

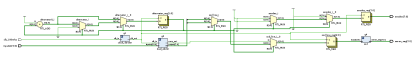


Figure 12: Register Transfer Level top level view

Figure 12 shows many similarities to the block diagram proposed at the beginning of this report. Asides from the finite state machine, which is represented with the alternator clock lines at the top of the RTL schematic, the rest of the diagram is near identical. The similarities become less apparent when taking a deep dive into the subsystems within the overall design. Figure 13 shows the RTL schematic for the clock divider subsystem.

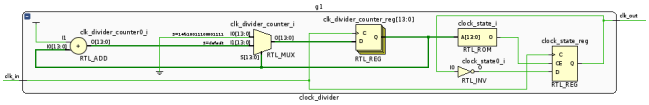


Figure 13: Register Transfer Level view of the clock divider subsystem

The clock divider circuit shows a hardware implementation of the logic that was used to define it. The logic for the clock divider is as follows:

```
process (clk_in)
    begin
        if rising_edge(clk_in) then
            if (clk_divider_counter = n-1) then
                clk_divider_counter <= 0;
                clock_state <= not clock_state;
            else
                clk_divider_counter <= clk_divider_counter + 1;
            end if;
        end if;
    end process;

clk_out <= clock_state;
```

As can be seen within this code logic, a counter is implemented, which is stored in a register which keeps on counting up. This is manifested within the left half of the RTL design, where the multiplexer is feeding into a register. The right hand side of the RTL design is concerned with the alternating of the clock state, which is manifested by a flip-flop connected to an inverter. This will flip the state of the clock every time the left-hand side has counted up to it's overflow value.

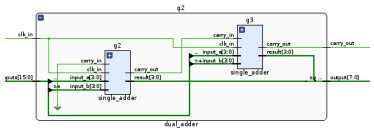


Figure 14: Register Transfer Level view of both BCD adder subsystems

Figure 14 shows the schematic of the two BCD adders put together. As can be seen, the first adder receives a carry-in input of 0 from ground, while receiving input lines from the thick buffer. The carry out from the first adder is transferred to the second, which also receives two inputs from the thick buffer. In terms of inputs and outputs, this system is identical to the proposed block diagram.

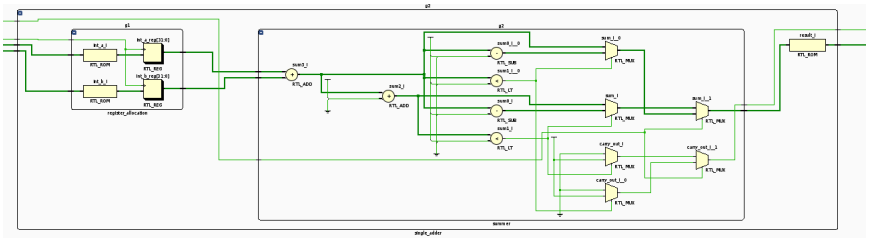


Figure 15: Register Transfer Level view of a single BCD adder

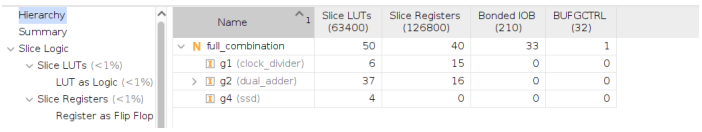
Figure 15 shows the RTL schematic of a single adder. Not really going to pretend to know what's going on with this one.

Figure 16: Register Transfer Level view of the seven-seg decoder

Figure 16 is the final component of the RTL schematic. This component represents a seven-segment decoder that takes a 4-bit input and outputs the 8-bits required for the seven-seg display. In the RTL schematic it is represented simply by a single block.

6 Synthesis Results

There were several key features worth considering in the synthesis of the model. The most important of these is the memory usage of the design. Figure 17 shows that less than 1% of the entire system's memory was used in terms of look-up tables and registers. Furthermore, 16% of the system's IO blocks were used in this design.



Hierarchy	Name	Slice LUTs (63400)	Slice Registers (126800)	Bonded IOB (210)	BUFGCTRL (32)
Summary					
Slice Logic					
Slice LUTs (<1%)					
LUT as Logic (<1%)					
Slice Registers (<1%)					
Register as Flip Flop					
	full_combination	50	40	33	1
	g1 (clock_divider)	6	15	0	0
	g2 (dual_adder)	37	16	0	0
	g4 (ssd)	4	0	0	0

Figure 17: Memory utilisation following synthesis

The other important component of the model's synthesis was to ensure that synchronicity is attained, and that the design is able to function without timing errors. This is shown in

figure 18, which shows that no slack exists in the design. It is worth mentioning that the previous practical featured a bug that introduced some degree of negative slack, as well as combinatorial logic loops that destabilised the clocking speed of the device. This error has since been addressed and is removed from practical 3.

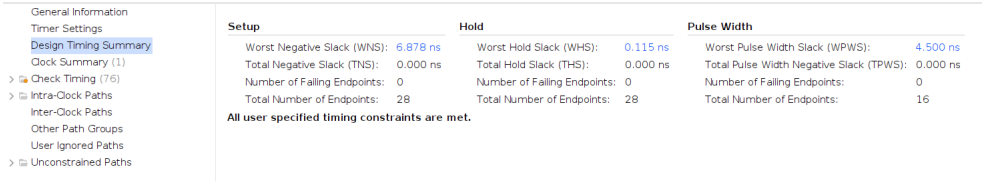


Figure 18: Timing of the synthesis showing that there are no failings with timing during implementation

7 Conclusion

In conclusion, both systems worked exactly as intended. The system for part 3A was able to create a 16-bit adders that provided all necessary functionality. Part 3B was a slightly more complex two-digit BCD adder, which was also created to satisfy the necessary design requirements. This time, there is no required places for improvement. It functions fully, while consuming less than 1% of the FPGA's resources.