<u>EECE 2323 Digital Logic Design Lab</u>

Name: Daniel Gilligan **Lab** 1

Lab Partner: Eve Mooney **Date**: 1/23/2024

Background

The purpose of this lab is to create and design a module that is able to add two sixteen-bit signed numbers that contains 2 outputs. One of the outputs is a 16 bit signed number and the other is an overflow flag, indicating if the addition of the two numbers resulted in an overflow, which can happen if the sum of the 2 numbers exceeds the 16 bits of output that are available. For example, if we compute $FFF3_{16}$ + $FF98_{16}$, we get the output $1FF8B_{16}$, which overflows the 16 bits of output, so the program would output the first 16 bits of the sum as well as an output bit that is set to 1. Since our physical hardware did not have 32 input switches, we used a virtual I/O package to simulate entering inputs into the hardware, which will then be processed by our adder module. The purpose of this lab is to provide more practice with writing SystemVerilog modules, creating test cases, running test bench simulations, and also gaining more familiarity with the Vivado IDE.

PreLab

a	b	f	ovf
16'h25	16'h45	16'h6A	0
16'hA415	16'hA555	16'h496A	1
16'hF115	16'hF215	16'hE32A	0
16'h9D00	16'h9E00	16'h3B00	1
16'hED00	16'hEF03	16'hDC03	0
16'h8A10	16'h7110	16'hFB20	0
16'h21AA	16'h21BB	16'h4365	0
16'h9A00	16'h9F4F	16'h394F	1

Table 1: Prelab Test cases for 16 bit adder

2)
An equation to represent the overflow bit is shown below:

ovf = (a[15] & b[15] & ~f[15]) | (~a[15] & ~b[15] & f[15])

This is built based on the idea that, there is overflow if the sign (shown by the most significant bit(msb)) of the output is different from the sign's of the input numbers (which should have the same sign for there to be overflow).

3)

A	B	F
0000 0000 0010 0101	0000 0000 0100 0101	0000 0000 0110 1010
1010 0100 0001 0101	1010 0101 0101 0101	0100 1001 0110 1010
1111 0001 0001 0101	1111 0010 0001 0101	1110 0011 0010 1010
1001 1101 0000 0000	1001 1110 0000 0000	0011 1011 0000 0000
1110 1101 0000 0000	1110 1111 0000 0011	1101 1100 0000 0011
1000 1010 0001 0000	0111 0001 0001 0000	1111 1011 0010 0000
0010 0001 1010 1010	0010 0001 1011 1011	0100 0011 0110 0101
1001 1010 0000 0000	1001 1111 0100 1111	0011 1001 0100 1111
1111 1111 1111 1111	1111 1111 1111 1111	1111 1111 1111 1111
(missing a 1 at the red spot)	(missing nothing)	(missing a 1 at the red spots)

Table 2: Checking if every bit was tested in prelab test cases

In the prelab test cases, we found that the A input had a 0 and 1 for every bit, except the 7th bit which never is tested with a 1. In the B input, all bits are tested with a 0 and 1. We also checked the output in case, and found it was not tested with a 1 in the 5th and 8th bits.

To make sure everything was fully tested, we added 2 test cases:
16'hC0 + 16'h01 = 16'hC1
16'h90 + 16'h02 = 16'h92

Design & Implementation

Our SystemVerilog code was straight forward, as this was quite a simple operation to describe behaviorally. Looking at Appendix A, we can see that the module contains 2 16 bit signed inputs called *a* and *b*, one 16-bit signed output (*f*), and a one bit output flag representing overflow (*ovf*). Looking at the code, we can see that the output *f* is calculated by simply assigning it to the sum of *a* and *b* (*f* = *a* + *b*), since SystemVerilog is made to handle two's complement. The overflow bit was calculated by checking if the sign bit of the output is different from both of the sign bits in the inputs, and if it is different then this means that overflow occurred, since the sign bit was flipped. This works because if the input and output have 2 different sign bits, it will be impossible to overflow since we are adding 2 signed numbers within the 16 bit range. If the inputs are both positive/negative and the output is the opposite sign, then this means that overflow occurred because the addition resulted in the first 15 bits of the signed numbers overflowing into the sign bit.

For our test bench code, which is shown in Appendix B, most of the cases were provided to us in our prelab, but we were told to add more test cases so that we have cases for every input and output bit. The cases we added were 00C0 +0001, and 0090 + 0002. These needed to be tested because the output was missing the 5th and the 8th bits as 1's,, while input a was missing the 7th bit as a 1. The Xilinx board converted our *f* (output) values to hex, and displayed the output on the 7 segment display. The overflow was displayed on the first LED on the Xilinx board. The only errors that we experienced was when trying to run our simulation at first, we were not getting any output, which ended up being a result of us not naming our module instantiation correctly and forgetting to add the UUT in front of the initialization.

Evaluation

Testbench & Simulation

We first tested our 16 bit adder using behavioral simulation. We did this using testbench code, which can be found in Appendix B. This testbench runs 10 tests, which cover each of the 16 bits of both inputs being 0 and 1, along with all 16 output bits(all of the prelab test cases, including the additional tests needed to make sure all bits were covered). These tests produced the waveform shown in Figure 1, which has the correct expected output.

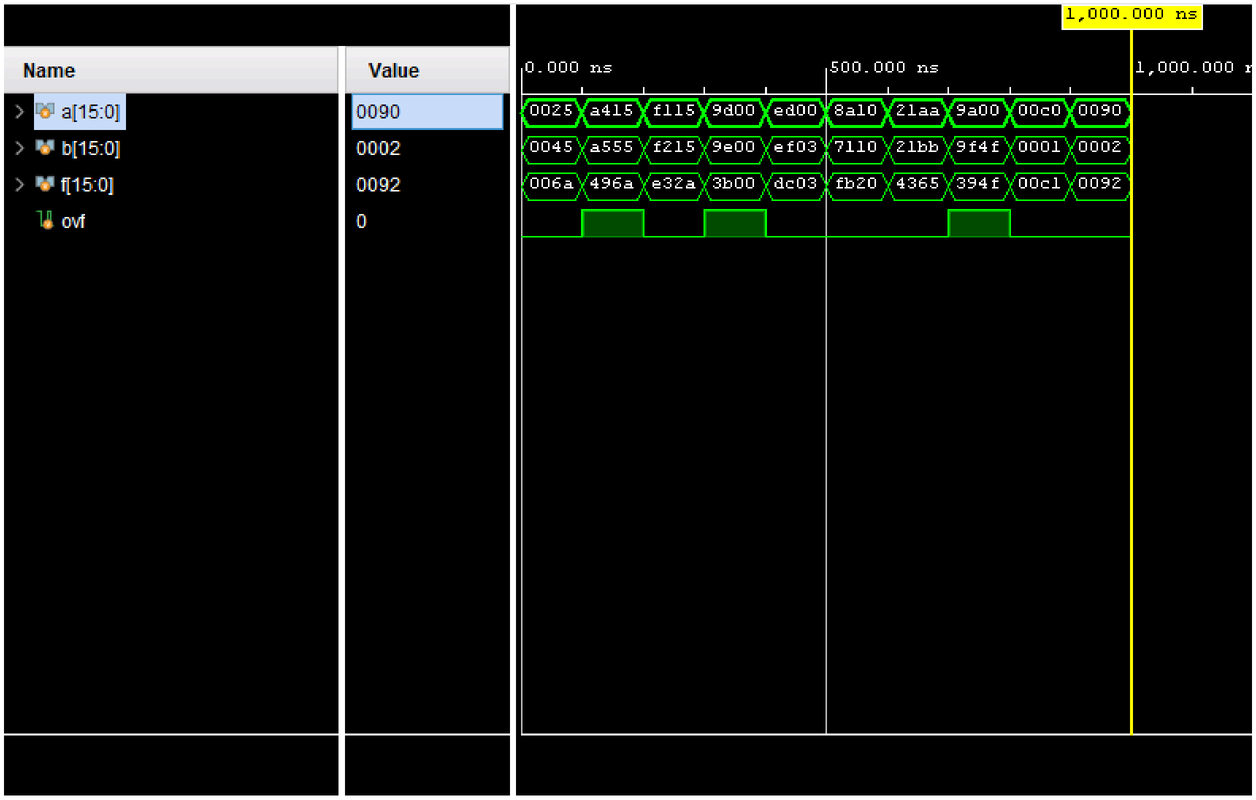
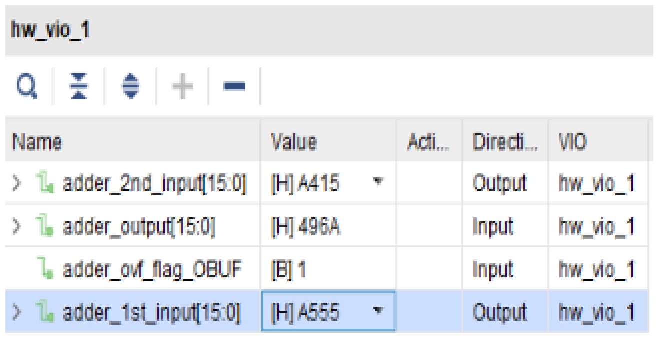

1,000.000 ns
Name
Value
0.000 ns
500.000 ns
1,000.000 ns
a[15:0]
0090
0025 a415 f115 9d00 ed00 8a10 21aa 9a00 00c0 0090
b[15:0]
0002
0045 a555 f215 9e00 ef03 7110 21bb 9f4f 0001 0002
f[15:0]
0092
006a 496a e32a 3b00 dc03 fb20 4365 394f 00c1 0092
ovf
0

Figure 1: Simulation Waveform based on the testbench in Appendix B

Hardware Testing

After we ran the behavioral simulation, we connected the board and moved on to testing the hardware. Because there are only eight switches on the PYNQ-Z2 and we need 2 16 bit inputs, a VIO needed to be used for the inputs. The way the Vio was set up, the outputs of the vio connected to the inputs of the hardware design, and the inputs of the vio connect to the design outputs. The ovf was mapped to an LED on the board, and the output bits were mapped to display on the add-on board(using the constraint file given). The hardware was tested by setting the VIO values representing the design inputs to different values, while connected to the board. An example output is shown in Figure 2, and more examples can be found in Appendix C.


hw_vio_1

Name	Value	Acti...	Directi...	VIO
adder_2nd_input[15:0]	[H] A415		Output	hw_vio_1
adder_output[15:0]	[H] 496A		Input	hw_vio_1
adder_ovf_flag_OBUF	[B] 1		Input	hw_vio_1
adder_1st_input[15:0]	[H] A555		Output	hw_vio_1

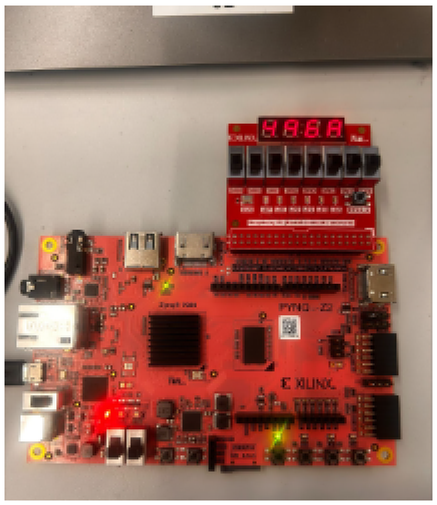


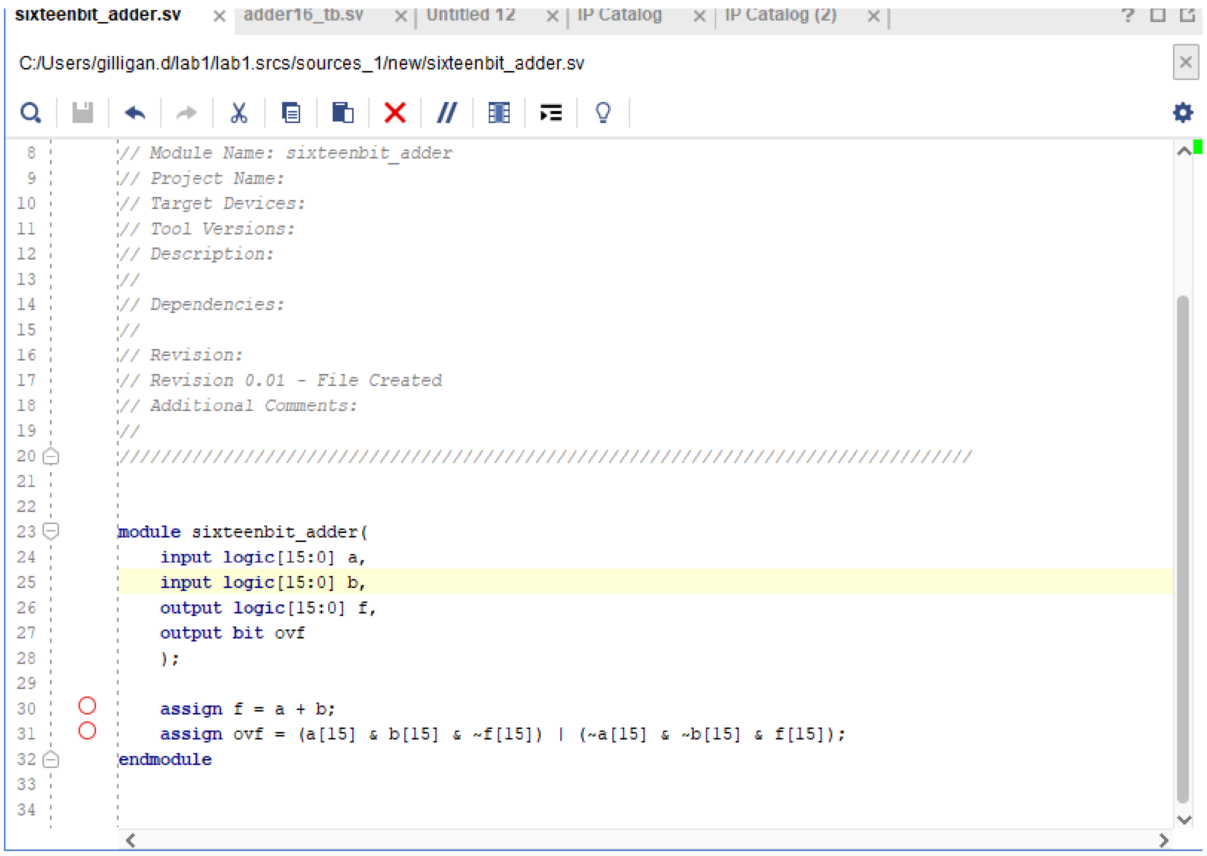
Figure 2: VIO input and output and hardware output, A415+A555=496A ovf=1

Our design operated successfully on the hardware as you can see. The green light at the bottom (not on the add-on board) displays the overflow bit, it is lit up when there is overflow and is not when there is not, while the display on the add-on board shows the other output.

Conclusion

In summary, this lab allowed us to practice using the software and hardware used in this course, create expansive test cases, and experience using a vio. In this lab, we were able to use the behavioral continuous model to describe our circuit in a quick and easy to understand way. We also learned how to make sure we were testing expansively to ensure the correct behavior was demonstrated, by checking every input bit to make sure it is tested with a 0 and 1 at some point in our test cases. This will be helpful later on so that we can ensure our designs are working before we have to build more complicated designs on top of them, and make sure those designs are also tested expansively. Additionally, we learned to use a VIO in this lab, which is important because the boards we are working with will not always have enough switches for the inputs we need to put in, so it is important to have the tools to simulate larger inputs when needed. In the future, it may have been interesting to set the ovf bit to a different position on the board, possibly on the add-on board, because it was somewhat unclear where the overflow was being displayed since it was so far from the inputs. This would have been an interesting challenge as well, since we have not edited the constraint files ourselves in the lab yet, we have just been provided with them.

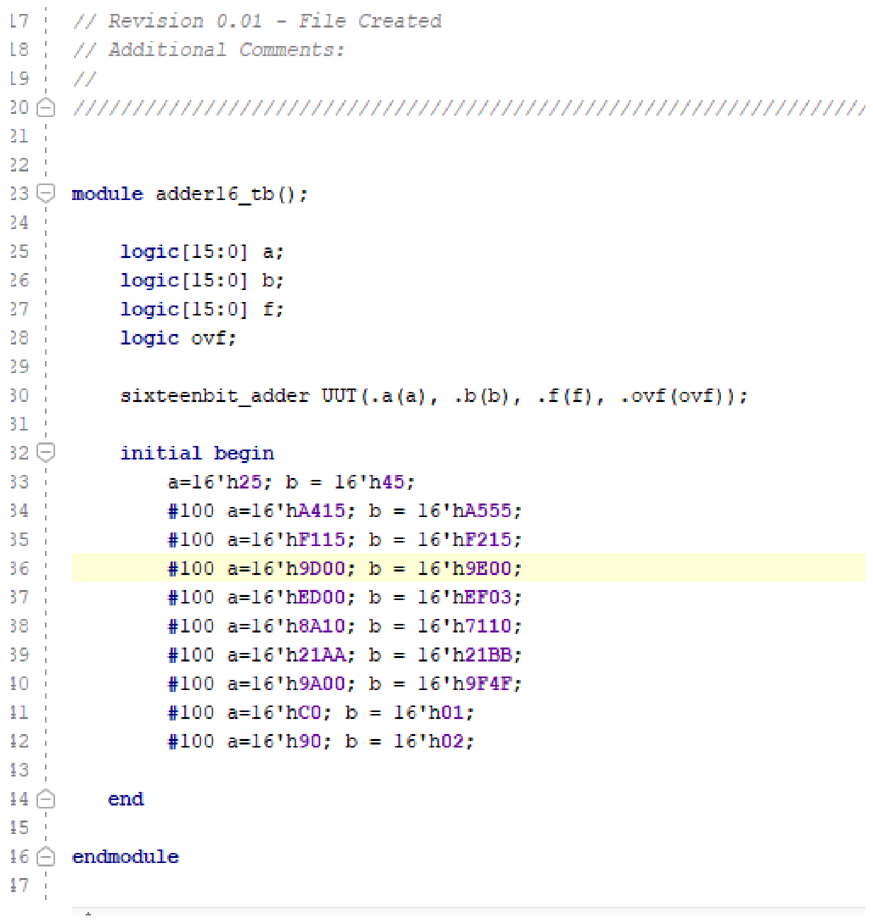
Appendix A - 16 bit adder SystemVerilog code


sixteenbit_adder.sv | adder16_tb.sv | Untitled 12 | IP Catalog | IP Catalog (2)

C:/Users/gilligan.d/lab1/lab1.srcs/sources_1/new/sixteenbit_adder.sv

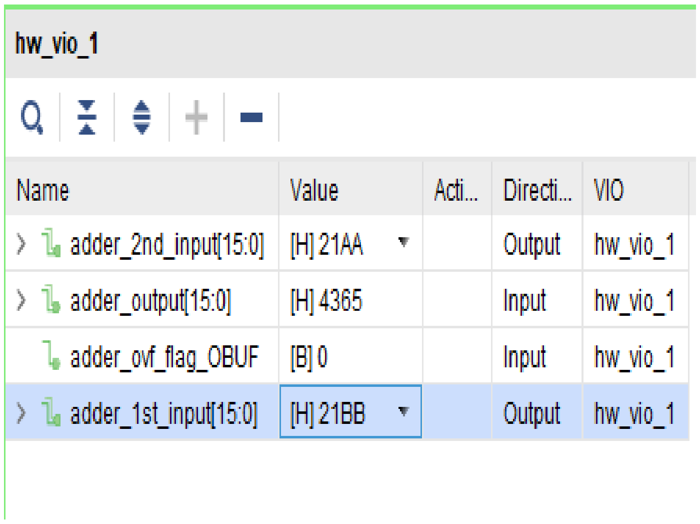
```
// Module Name: sixteenbit_adder
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module sixteenbit_adder(
    input logic[15:0] a,
    input logic[15:0] b,
    output logic[15:0] f,
    output bit ovf
    );

    assign f = a + b;
    assign ovf = (a[15] & b[15] & ~f[15]) | (~a[15] & ~b[15] & f[15]);
endmodule
```

Appendix B - TestBench File Code



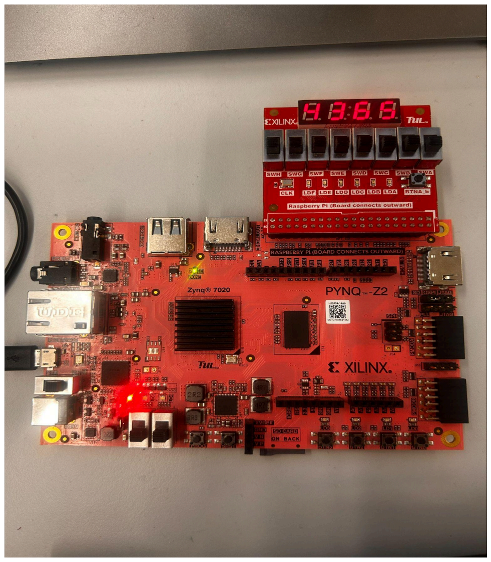
```
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////


module adder16_tb();

    logic[15:0] a;
    logic[15:0] b;
    logic[15:0] f;
    logic ovf;

    sixteenbit_adder UUT(.a(a), .b(b), .f(f), .ovf(ovf));

    initial begin
        a=16'h25; b = 16'h45;
        #100 a=16'hA415; b = 16'hA555;
        #100 a=16'hF115; b = 16'hF215;
        #100 a=16'h9D00; b = 16'h9E00;
        #100 a=16'hED00; b = 16'hEF03;
        #100 a=16'h8A10; b = 16'h7110;
        #100 a=16'h21AA; b = 16'h21BB;
        #100 a=16'h9A00; b = 16'h9F4F;
        #100 a=16'hC0; b = 16'h01;
        #100 a=16'h90; b = 16'h02;

   end

endmodule
```

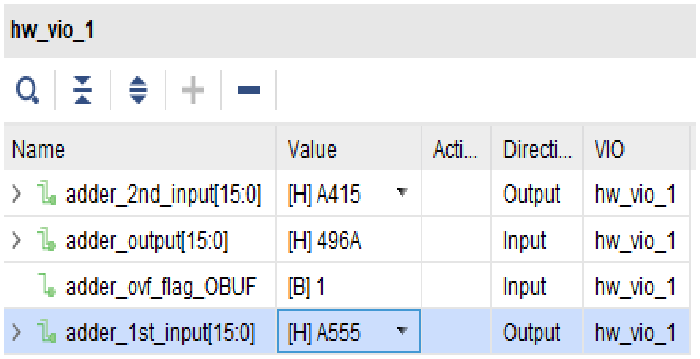
Appendix C - Hardware Testing Pictures



hw_vio_1

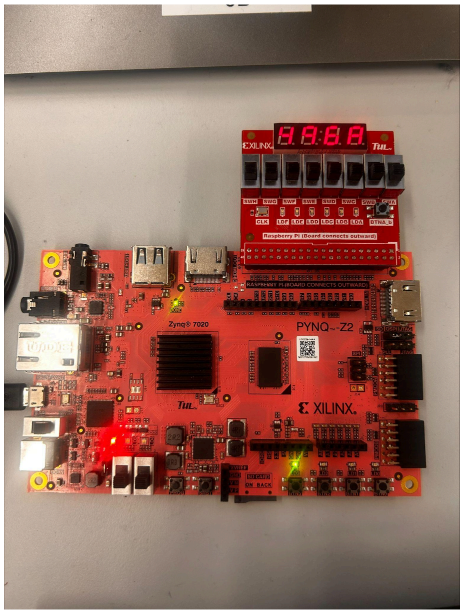
Name	Value	Acti...	Directi...	VIO
adder_2nd_input[15:0]	[H] 21AA		Output	hw_vio_1
adder_output[15:0]	[H] 4365		Input	hw_vio_1
adder_ovf_flag_OBUF	[B] 0		Input	hw_vio_1
adder_1st_input[15:0]	[H] 21BB		Output	hw_vio_1

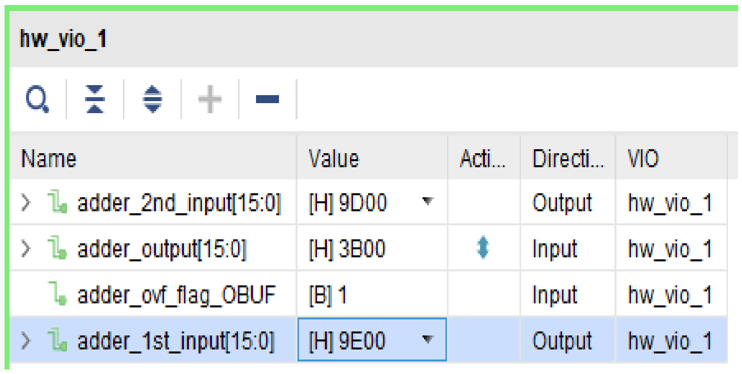




hw_vio_1

Name	Value	Acti...	Directi...	VIO
adder_2nd_input[15:0]	[H] A415		Output	hw_vio_1
adder_output[15:0]	[H] 496A		Input	hw_vio_1
adder_ovf_flag_OBUF	[B] 1		Input	hw_vio_1
adder_1st_input[15:0]	[H] A555		Output	hw_vio_1





hw_vio_1

Name	Value	Acti...	Directi...	VIO
adder_2nd_input[15:0]	[H] 9D00		Output	hw_vio_1
adder_output[15:0]	[H] 3B00		Input	hw_vio_1
adder_ovf_flag_OBUF	[B] 1		Input	hw_vio_1
adder_1st_input[15:0]	[H] 9E00		Output	hw_vio_1