Assignment 1 - Arithmetic Logic Unit

Singularity

February 01, 2022

1 Introduction

We have constructed an ALU that can operate on 64-bit operands. It is written in Verilog (HDL), using the Icarus Verilog and Xilinx software.
It can perform:

1. Addition
2. Subtraction
3. AND operation
4. XOR operation, on any two 64 bit signed operands.

In case the output is a number greater than 64 bits, an overflow will be generated. We have created 5 modules that act as building blocks for ALU. The functionality of each module is verified by writing testbenches with a reasonable number of inputs.

2 Modules

To build an ALU, we have created the following modules:

1. Full Adder
2. Adder
3. Subtractor
4. AND
5. XOR

2.1 Full Adder

It is a standard 1-bit adder that adds two 1-bit inputs along with an input carry. The "Sum" and "Carry Out" are the two 1-bit outputs. It is the most basic module used for creating ALU. This module is used in the 64-bit adder as well as the 64-bit subtractor.

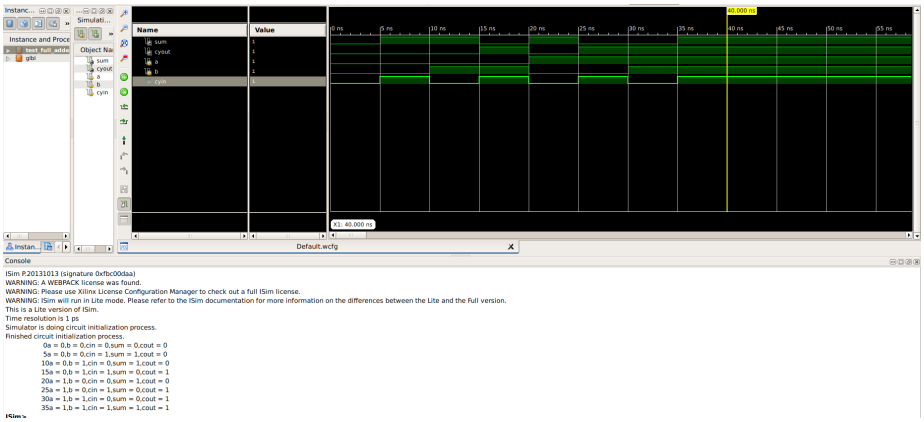

Figure 1: Truth table and waveform of Full Adder

2.2 Adder

It is a 64-bit adder that operates on two 64-bit signed operands resulting in addition of the two. We have instantiated the Full adder block 64 times using the generate block. The carry input of the very first full adder is set to zero. The input carry part is coded outside the generate block.
The inputs to the Adder are signed numbers 'a' and 'b', and the output is another signed number 'sum'. If the operation results in a number greater than 64-bits, overflow will be set to 1. Overflow might occur when addition of two same-sign bits takes place. In this scenario, the sign bit of result is not sensible concerning the inputs. This anomaly is captured by the overflow.
The module has been tested for a set of inputs and resulting waveform and value are shown in figure 2.

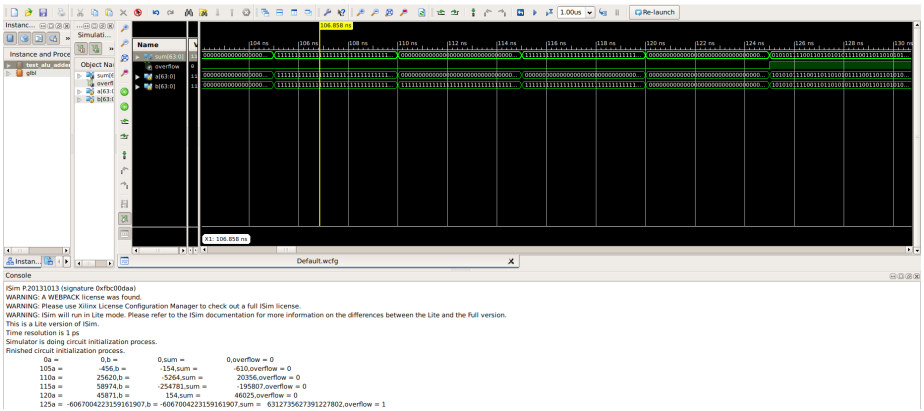

Figure 2: Test values and waveform of 64-bit Adder

2.3 Subtractor

It is a 64-bit subtractor that operates on two 64-bit signed operands resulting in subtraction of the two. We have used 2's complement method to execute this function. For this purpose, we invert every bit of subtrahend by using 64 NOT gates with the help of generate block and obtain 1's complement. After inversion, we add the minuend and subtrahend by instantiating the full adder using generate block. One point to be noted is that we set the initial carry of first full adder to 1, so that we obtain 2's complement of subtrahend.
The inputs to the subtractor are signed numbers 'a' and 'b', and output is another signed number 'difference'. If the operation results in a number greater than 64-bits, overflow will be set to 1. The module has been tested for a set of inputs and resulting waveform and value are shown in figure 3.

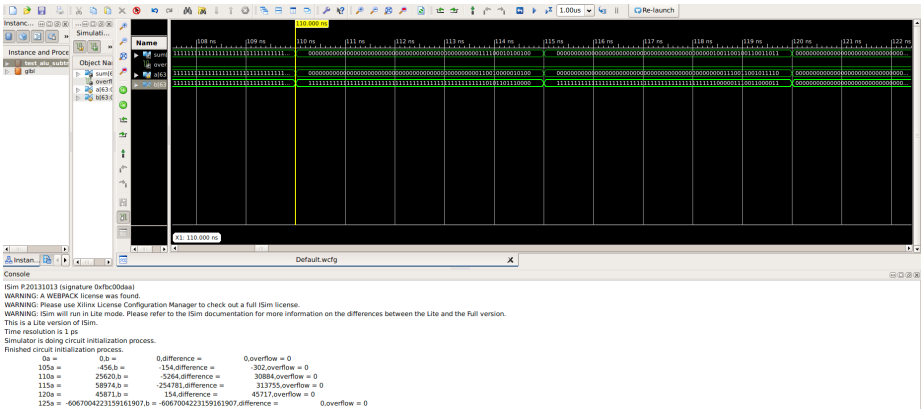

Figure 3: Test values and waveform of 64-bit Subtractor

2.4 AND operation

This module helps us perform AND operation between two corresponding bits of 'in1' and 'in2'. The bits in output register are 1 only when both the corresponding bits in inputs are 1. This module is build with the help of AND gate operation and generate block.
The module has been tested for a set of inputs and resulting waveform and value are shown in figure 4.

2.5 XOR operation

This module helps us perform XOR operation between two corresponding bits of 'in1' and 'in2'. The bits in output register are 1 only when both the corresponding bits in inputs are unequal. This module is build with the help of XOR gate operation and generate block.
The module has been tested for a set of inputs and resulting waveform and value are shown in figure 5.

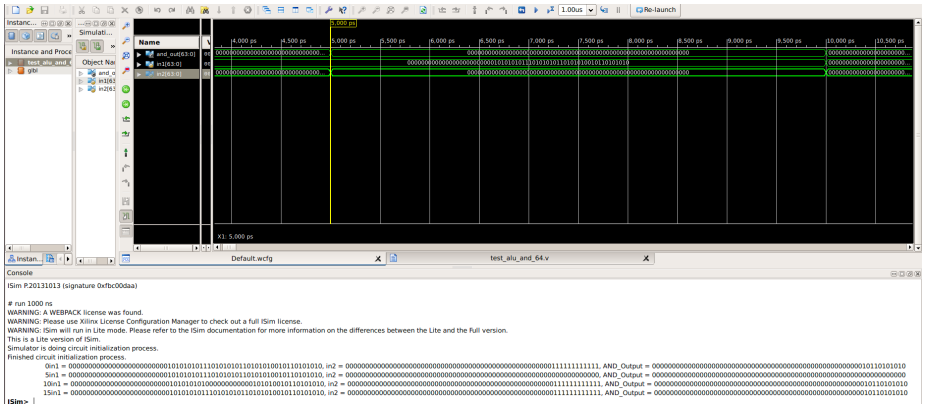

Figure 4: Test values and waveform of 64-bit AND

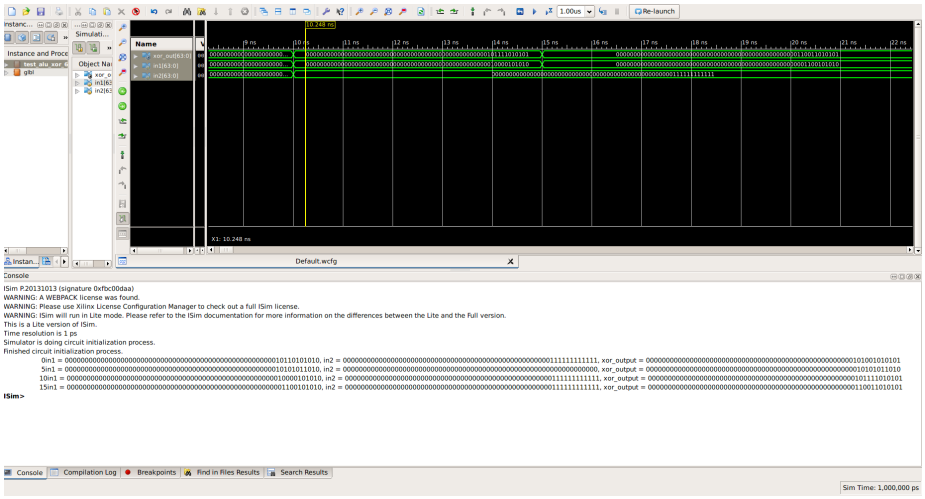

Figure 5: Test values and waveform of 64-bit XOR

3 ALU module

It is the final module in which all the above discussed sub-modules are instantiated. It takes two 64-bits signed inputs and performs specific operation depending upon Control Signal. Control signal is a 2 bit binary number which helps ALU to decide the output.

Control 00	ADD a and b
Control 01	Subtract b from a
Control 10	a AND b
Control 11	a XOR b

We have used the conditional arguments in order to assign desired values to output register.

$$assignout = sel[1]?(sel[0]?d : c) : (sel[0]?b : a);$$

The overflow value is also set in the same way. We also create temporary variables so that output of the instantiated modules are accessible.
We have successfully tested the ALU module by giving a set of inputs. The output is as shown in figure 6.

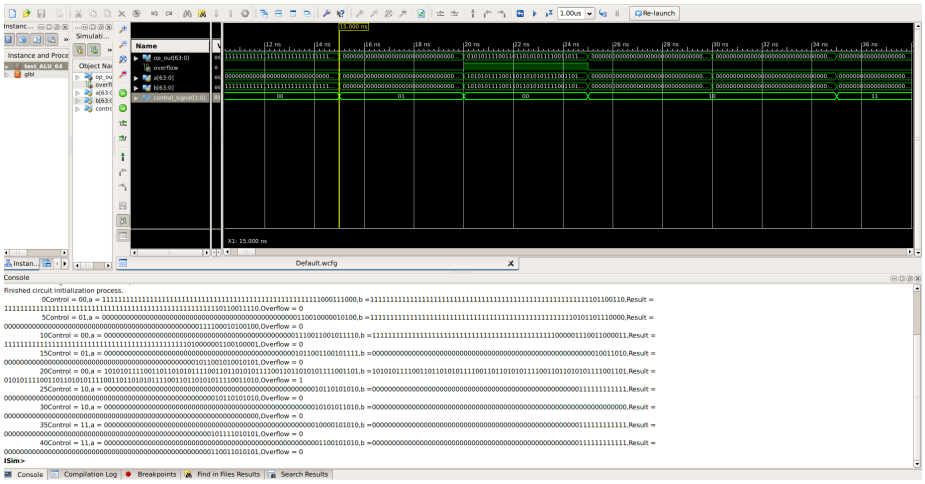

Figure 6: Test values and waveform of 64-bit ALU

4 Conclusion

We have successfully created a 64-bit ALU module using 5 sub-modules. Each one of the sub-modules was tested, and all of them were instantiated to obtain 64-bit ALU.