Lab E Report

Contains the following:

- Structural Verilog – a short section on your takeaways on designing digital modules using structural Verilog constructs (using B as the example)
- Bottom-up design – a short section that describes how you used building blocks from lab C (1-bit adder) to arrive at the wider adder subtractor module
- Test cases – a short section on your approach to testing the functionality of the adder / subtractor module (include the list of input/output case from bullet point 9 in lab C). Did your test cases cover all possible conditions and how/why?
- Implementation flow – a short section on your understanding of what happens in the implementation flow (don't worry, this is your understanding of the flow. We will cover this in detail in a later lecture).

Observations and resource consumption details captured for the Basys-3 implementation of your adder/subtractor module.

Structural Verilog

My takeaways on designing using structural Verilog constructs are that it is quite low level in the sense that every individual 'wire' connection must be explicitly defined and 'assigned'. Additionally, having to write out the truth table beforehand and have the correct Boolean logic function can be non-trivial especially for a large number of input parameter due to the exponential scale at which the number of output cases increase (2^N where N is the number of inputs).As a result, debugging can become quite non-trivial and arduous, which of course also leads to bottlenecks when it comes to scalability.

Bottom-up Design

The preexisting 1-bit adder from lab C was used as follows:

```
FullAdder unit1(.a(x[0]), .b(y_copy[0]), .cin(sel), .s(sum[0]), .cout(unit1_cout));
FullAdder unit2(.a(x[1]), .b(y_copy[1]), .cin(unit1_cout), .s(sum[1]), .cout(unit2_cout));
FullAdder unit3(.a(x[2]), .b(y_copy[2]), .cin(unit2_cout), .s(sum[2]), .cout(unit3_cout));
FullAdder unit4(.a(x[3]), .b(y_copy[3]), .cin(unit3_cout), .s(sum[3]), .cout(unit4_cout));
FullAdder unit5(.a(x[4]), .b(y_copy[4]), .cin(unit4_cout), .s(sum[4]), .cout(unit5_cout));
FullAdder unit6(.a(x[5]), .b(y_copy[5]), .cin(unit5_cout), .s(sum[5]), .cout(unit6_cout));
```

Akin to classes and in C++, I created instances of each 'FullAdder' unit and mapped my predefined wires that existed in the 6bit_ripple_adder module to the inputs of each unit, rather to constantly referring to the input sequence that existed in the 'FullAdder' module I made use of the '.', to ensure each wire and input was mapped correctly, removing any room for misassignment. The use of the building block made for significantly more readable and debugable Verilog code.

Test Cases

Implementation Flow

My present understanding of implementation flow is that, it follows an agile and iterative approach.

Initially, structural or behavioural constructs are used to design a digital logic circuit. Subsequently, a testbench is formulated and simulation is ran, in order to validate the performance of said design to ensure it adheres to all truth table values. Upon successfully completing the simulation phase, after making required alterations, the design is then synthesized in order to configure it for the target hardware, this involves connecting the module ports to the appropriate pins. Finally, the implementation is ran and the bitstream is generated and uploaded to the target device. During the synthesis and implementation phases one can observe and analyse the resource consumption details of their design to discern it's efficiency.

Resource Consumption

Resource	Utilization	Available	Utilization %
LUT	9	20800	0.04
IO	21	106	19.81