

# Digital System Design Assignment 1

Ruairi Mullally – 22336002

## Introduction:

This report examines the design, implementation, and testing of a miniature arithmetic logic unit (ALU) composed of multiple Verilog modules, which is the culmination of the previous labs. Building on the foundation of a ripple-adder and a 'greater than or equal to' comparator module, several functions were integrated. Each module was individually verified before integration ensuring robustness and simplifying debugging. A holistic testing strategy was also employed which verified the correct functionality of the ALU but noted possible improvements in robustness and error handling.

## Method:

fxn	X[5:0]
000	A
001	B
010	-A
011	-B
100	A<B (is A less than B)
101	(A <i>nxor</i> B) (Bitwise XNOR)
110	A+B
111	A-B

Figure 1. Mini-ALU connections.

To provide the ability to select a certain function of the mini-ALU, an 8x1 multiplexer was used. The return value (X) of the mini-ALU is selected by an 8x1 multiplexer, controlled by 'fxn'.

Table1. Mini-ALU module re-utilization.

fxn	Description of computation using existing modules.
000	Select A from MUX (MUX is the top-level module).
001	Select B from MUX.
010	Subtract A from 0 in the ripple adder.
011	Subtract B from 0 in the ripple adder.
100	Extend A and B to 8 bits. Compute $A \geq B$ with the 'greater than or equal' module. Invert the output with the 'full adder' module to get $A < B$ .
101	Write a <b>new</b> module to do a bitwise XOR and negate the output.
110	Add B to A with ripple adder.
111	Subtract B from A with ripple adder.

**FXN 000:** This function returns the original value of A; therefore, the A input is just connected as a multiplexer input and no additional processing is needed.

**FXN 001:** This function returns the original value of B; therefore, the B input is just connected as a multiplexer input and no additional processing is needed.

**FXN 010:** This function returns the negative of input A. To do this without creating a new module the 6-bit ripple adder from previous labs could be used. This was achieved by subtracting A from 0 to get the negative of A.

**FXN 011:** This function returns the negative of input B. To do this without creating a new module the 6-bit ripple adder from previous labs could be used. This was achieved by subtracting B from 0 to get the negative of B.

**FXN 100:** To implement the comparison ( $A < B$ ), the “greater than or equal to” module from previous labs could be used. ( $A < B$ ) is logically the opposite of ( $A \geq B$ ) so the output from the “greater than or equal to” module could be negated to achieve this functionality. To negate the output, I used a 1-bit full adder from the ripple adder module. By computing the original output + 1 with an initial carry-in of 0, the output is effectively negated.

Additionally, the width of the “greater than or equal to” module was designed for 8 bits while the ALU operates on 6 bits. To adapt the “greater than or equal to” module to work, two trailing zeros were appended to the 6-bit numbers. This ensures that the sign-checking, which operates on the most significant bit, for the twos complement numbers still operates correctly.

**FXN 101:** This was the only function of the ALU that predicated a completely new module. Using the built-in Verilog functions, this module computes the bitwise XNOR of A and B.

**FXN 110:** This function computes  $A + B$ . To do this without creating a new module the 6-bit ripple adder from previous labs could be used.

**FXN 111:** This function computes  $A + B$ . To do this without creating a new module the 6-bit ripple adder from previous labs could be used.

Leveraging the use of existing modules greatly simplifies the testing of the design, as there are few modules, which have already been extensively tested. The downside, however, is that there may be inefficient resource usage. For example, in the case of inverting A, a whole ripple adder is used for something that could be done with a purpose-built inverter using less resources. For the purposes of this ALU, which aimed to re-use as much hardware as possible, this is an acceptable trade-off.

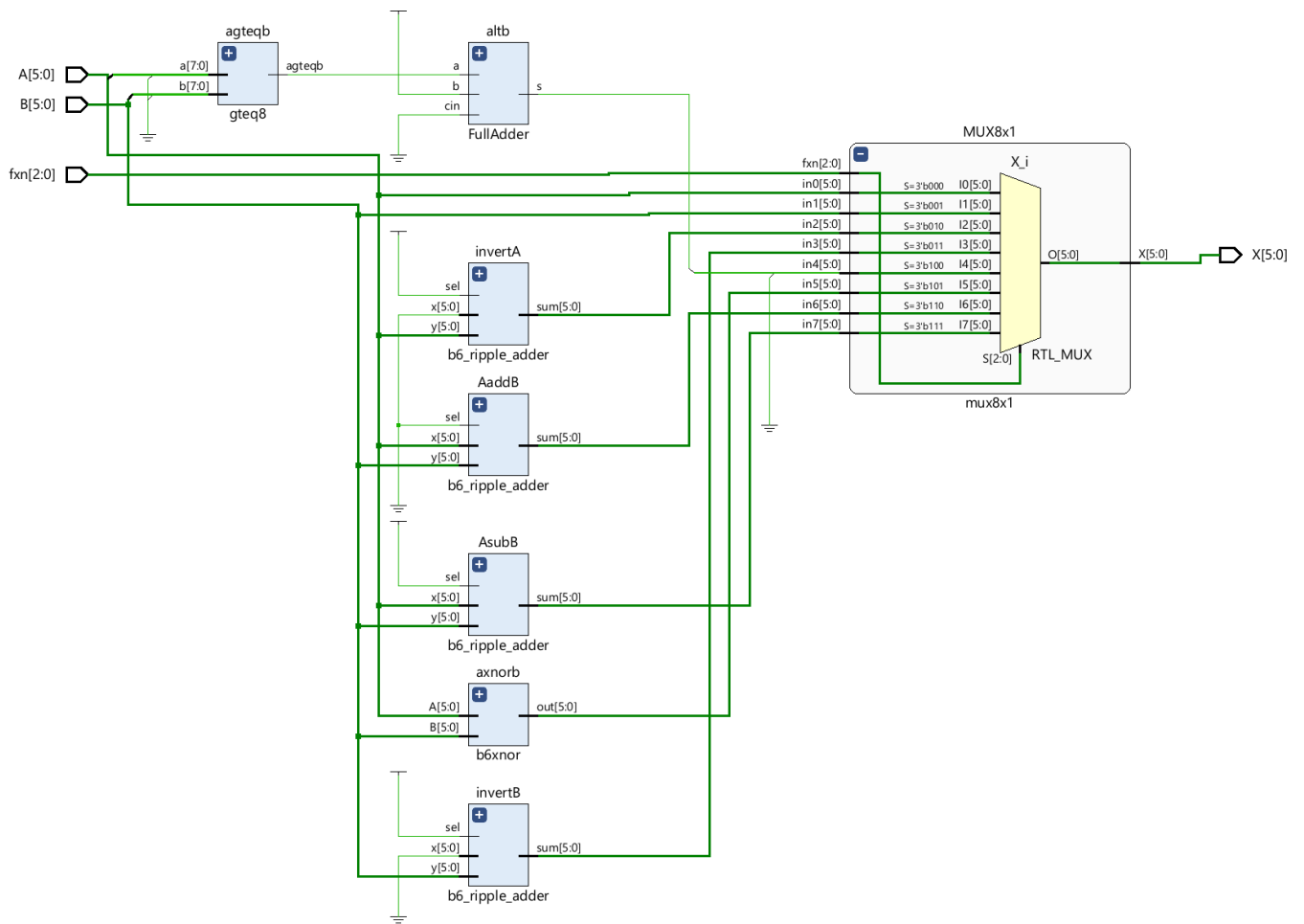


Figure 2. Schematic diagram of mini-ALU.

Inputs A and B are connected to every sub-module withing the ALU, enabling the parallel computation of all possible function simultaneously. The selection input, fxn, is connected to the multiplexer module. The output of each sub-module is also connected to the multiplexer, which using fxn outputs only the desired operation result. This parallel architecture allows for less delay as there are no sequential steps. Reusing some sub-modules such as the ripple-adder for inversion allows for a simpler but more wasteful (gates) design.

### Testing:

Individual sub-modules were extensively individually tested before their use in the implementation of the mini-ALU. They were each tested using comprehensive input permutations and edge-case tests. This modular approach greatly simplified debugging. For more details regarding the ripple-adder and 'greater than or equal to' modules, see appendix A.

Every function of the ALU was extensively tested for expected input. The chosen test vectors included my board number (56), 2 positive numbers, 2 negative numbers, combination of negative and positive number, all zeros and all 1s. These are all the possible permutations for expected input to the ALU.

It is important to recognise that these tests were all conducted for expected bitwise outcomes. Although a test might pass in truncated 6-bit terms, it may be arithmetically incorrect as the mini-ALU provides no error checking, such as overflow or carryout in the addition operations or any other of the operations. Arithmetic operations that exceed 6-bits are still regarded as correct if their truncated results match expectations. This can be seen in some of the addition subtraction test vectors, where the result of a subtraction falls outside of the 6-bit range.

Waveforms of the 7 chosen test vectors performed for each function are given below.

### FXN 000 (A):

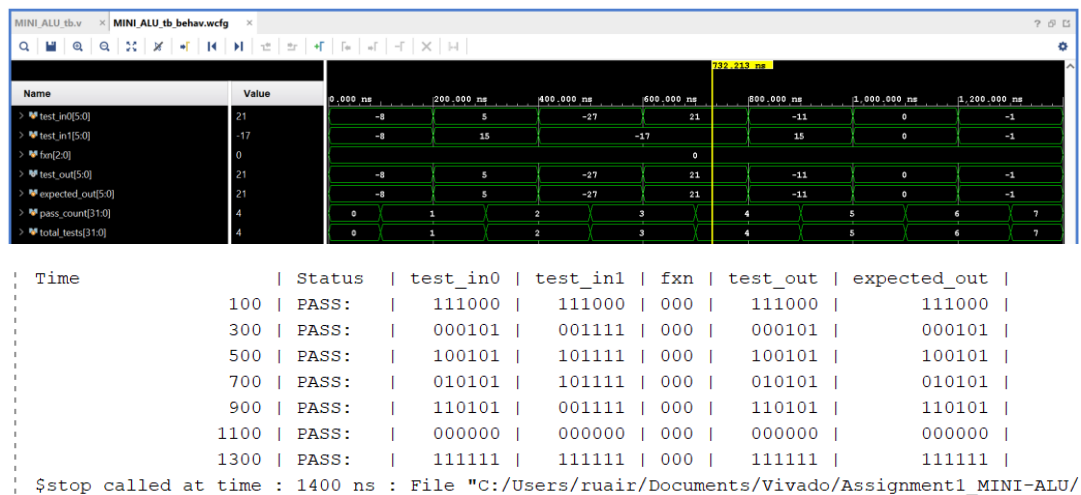


Figure 3. Testing results and waveform for ALU function 000.

### FXN 001 (B):

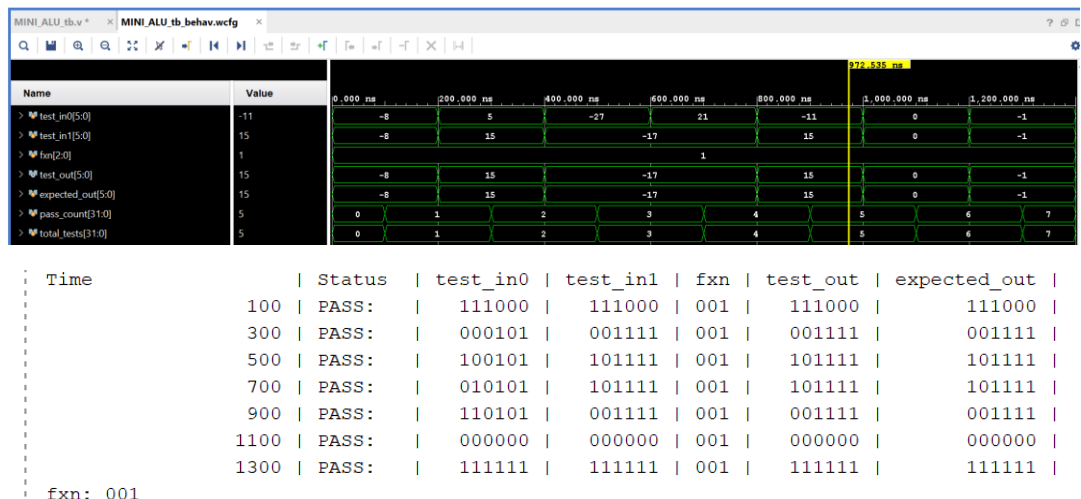
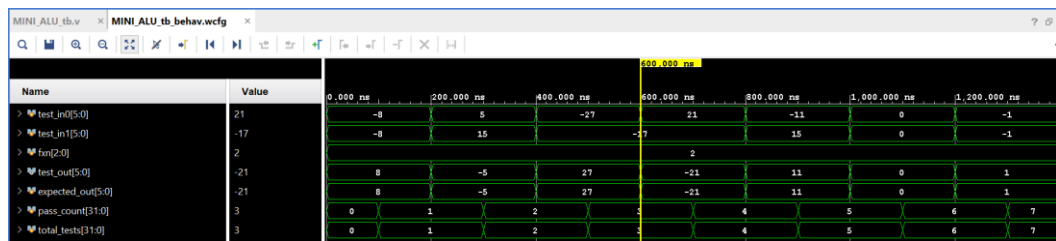


Figure 4. Testing results and waveform for ALU function 001.

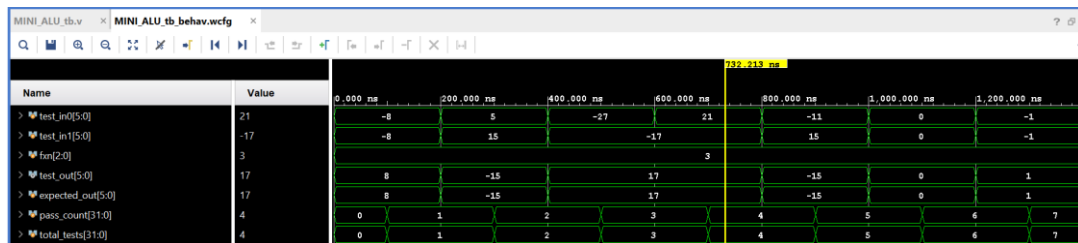
## FXN 010 (-A):



Time	Status	test_in0	test_in1	fxn	test_out	expected_out
100	PASS:	111000	111000	010	001000	001000
300	PASS:	000101	001111	010	111011	111011
500	PASS:	100101	101111	010	011011	011011
700	PASS:	010101	101111	010	101011	101011
900	PASS:	110101	001111	010	001011	001011
1100	PASS:	000000	000000	010	000000	000000
1300	PASS:	111111	111111	010	000001	000001

Figure 5. Testing results and waveform for ALU function 010.

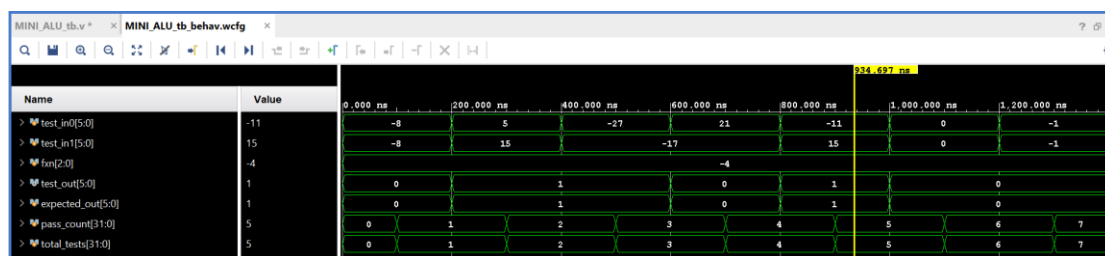
## FXN 011 (-B):



Time	Status	test_in0	test_in1	fxn	test_out	expected_out
100	PASS:	111000	111000	011	001000	001000
300	PASS:	000101	001111	011	110001	110001
500	PASS:	100101	101111	011	010001	010001
700	PASS:	010101	101111	011	010001	010001
900	PASS:	110101	001111	011	110001	110001
1100	PASS:	000000	000000	011	000000	000000
1300	PASS:	111111	111111	011	000001	000001

Figure 6. Testing results and waveform for ALU function 011.

## FXN 100 (A<B):



Time	Status	test_in0	test_in1	fxn	test_out	expected_out
100	PASS:	111000	111000	100	000000	000000
300	PASS:	000101	001111	100	000001	000001
500	PASS:	100101	101111	100	000001	000001
700	PASS:	010101	101111	100	000000	000000
900	PASS:	110101	001111	100	000001	000001
1100	PASS:	000000	000000	100	000000	000000
1300	PASS:	111111	111111	100	000000	000000

Figure 7. Testing results and waveform for ALU function 100.

## FXN 101 (A XNOR B):

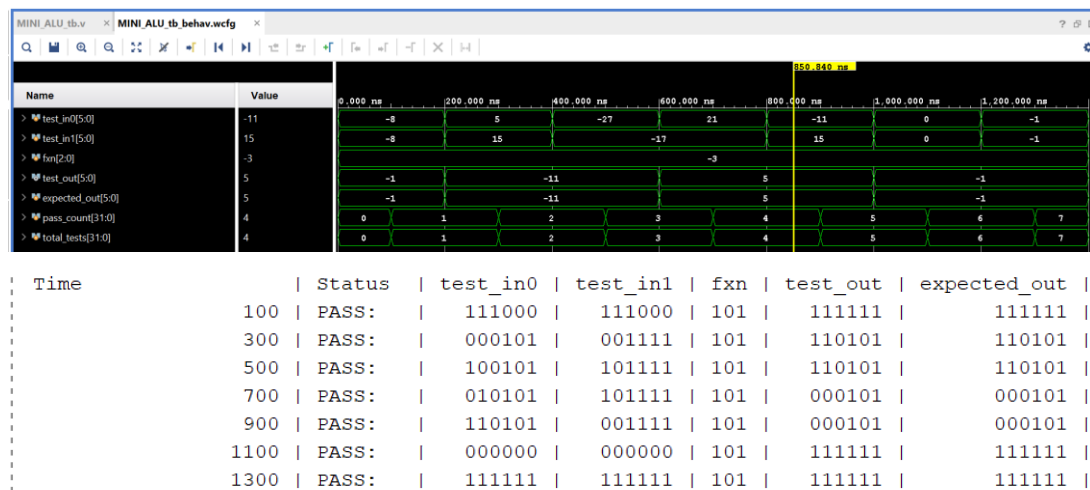


Figure 8. Testing results and waveform for ALU function 101.

## FXN 110 (A+B):

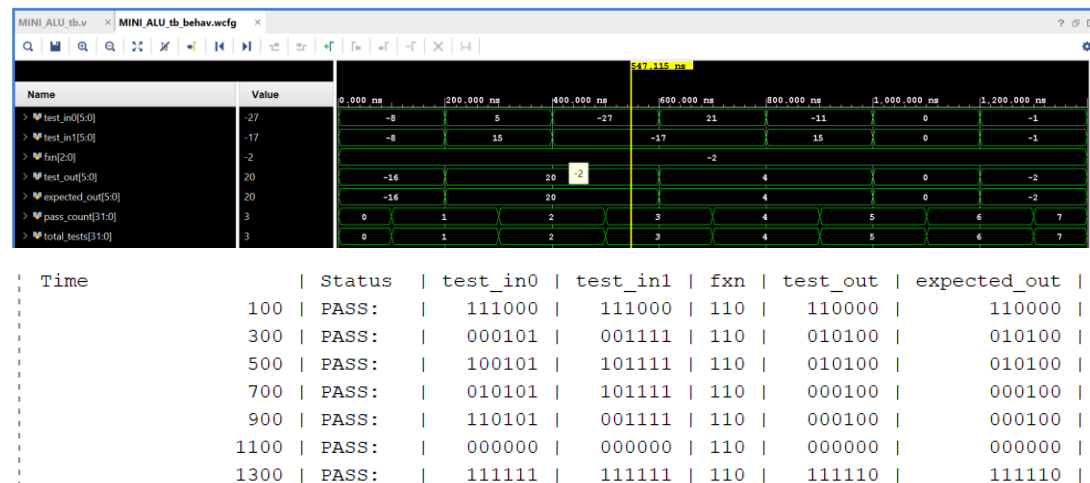


Figure 9. Testing results and waveform for ALU function 110.

## FXN 111 (A-B):

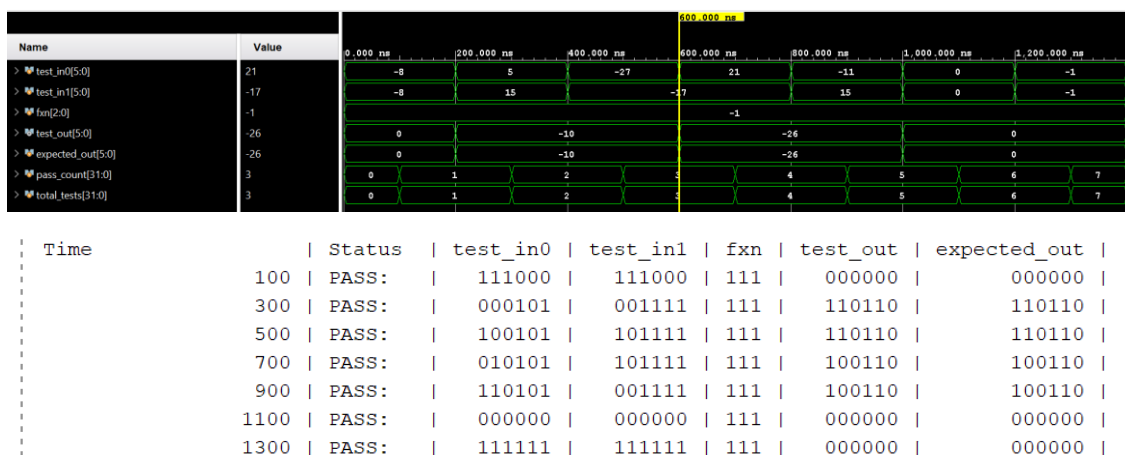


Figure 10. Testing results and waveform for ALU function 111.

## Robustness & Shortcomings:

Overall, the ALU is very well tested, with comprehensive testing with edge cases on a module-by-module level and an overall functional testing unit. The sub-modules are well tested for edge cases and can even handle overflows and unexpected input. The same cannot be said for the ALU. The key shortcoming of the ALU is that it does not avail of the ripple-adders built in overflow detection. This means it is not robust to unexpected or unintended input. Provided the ALU is used appropriately within the 6-bit range, it will function correctly, however outside this range it will behave predictably incorrectly, truncated to 6 bits.

## Results & Demo:

### Summary

Resource	Utilization	Available	Utilization %
LUT	29	20800	0.14
IO	21	106	19.81

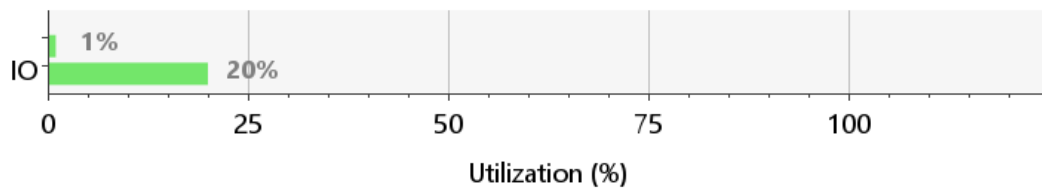


Figure 11. Utilization report of implemented design.

Look-up table utilization is very low for this design, only using 0.14% of the Basys3's available resources. 20% of IO is used for connecting inputs A, B, fxn on the boards switches and displaying the output with the LEDs.

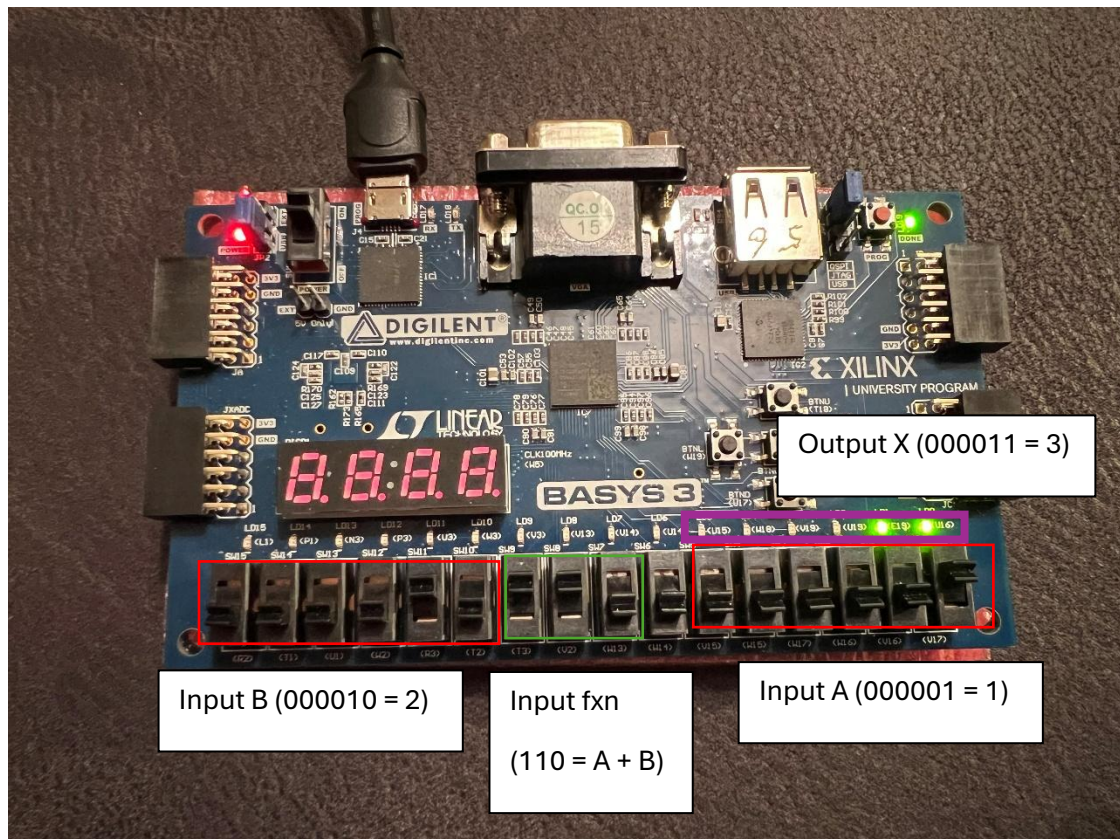


Figure 12. How to use the ALU.

In figure 12 the inputs and outputs are denoted. Input A is surrounded by the red box on the right, with the least significant bit (LSB) on switch V17, and the most significant bit (MSB) on switch V15. Input B is surrounded by the red box on the left with LSB at switch T2 and MSB at switch R2. The input fxn is surrounded by the green box with LSB at W13 and MSB at T3. The output is displayed in binary on the LEDs surrounded by the purple box, with LSB at V16, and MSB at V15. Move the switches into the appropriate binary form to perform the desired output, which will be displayed on the LEDs. In this example, function  $A + B$  is performed, with A set to 1 and B set to 2, the LEDs display the result of 3.



## File structure:

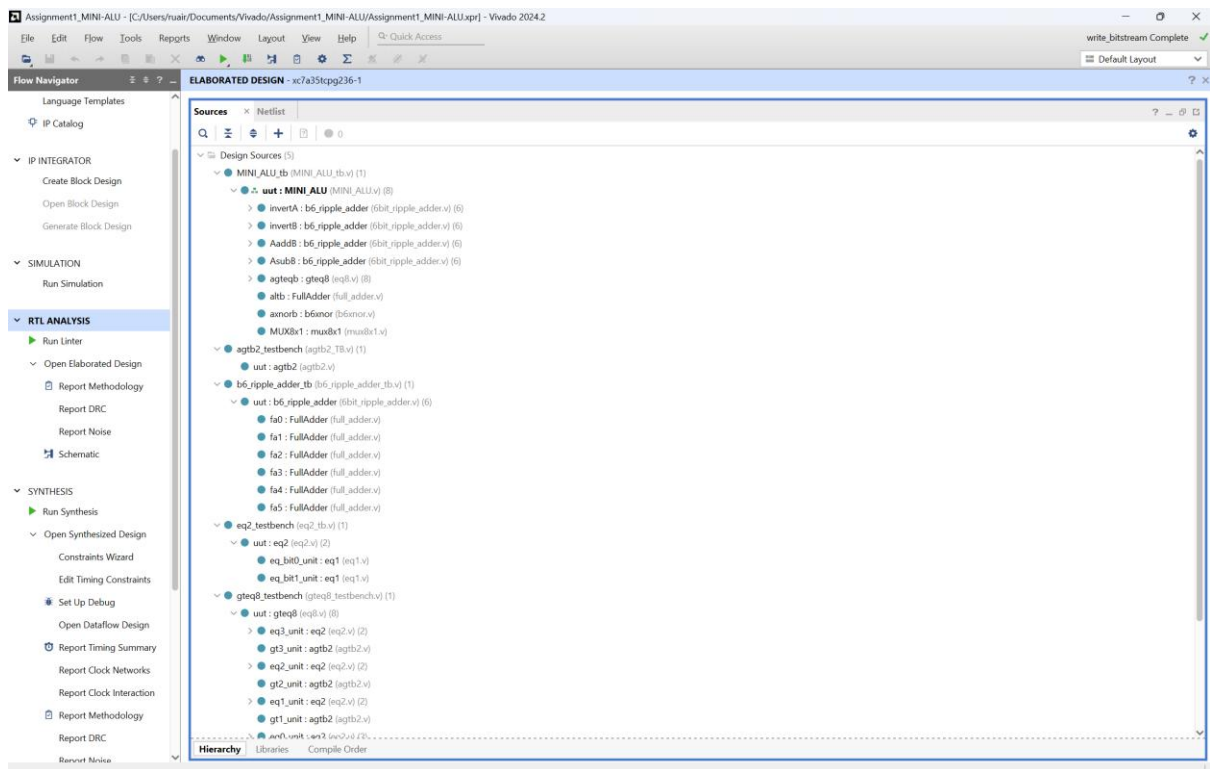


Figure 13. File structure in Vivado.

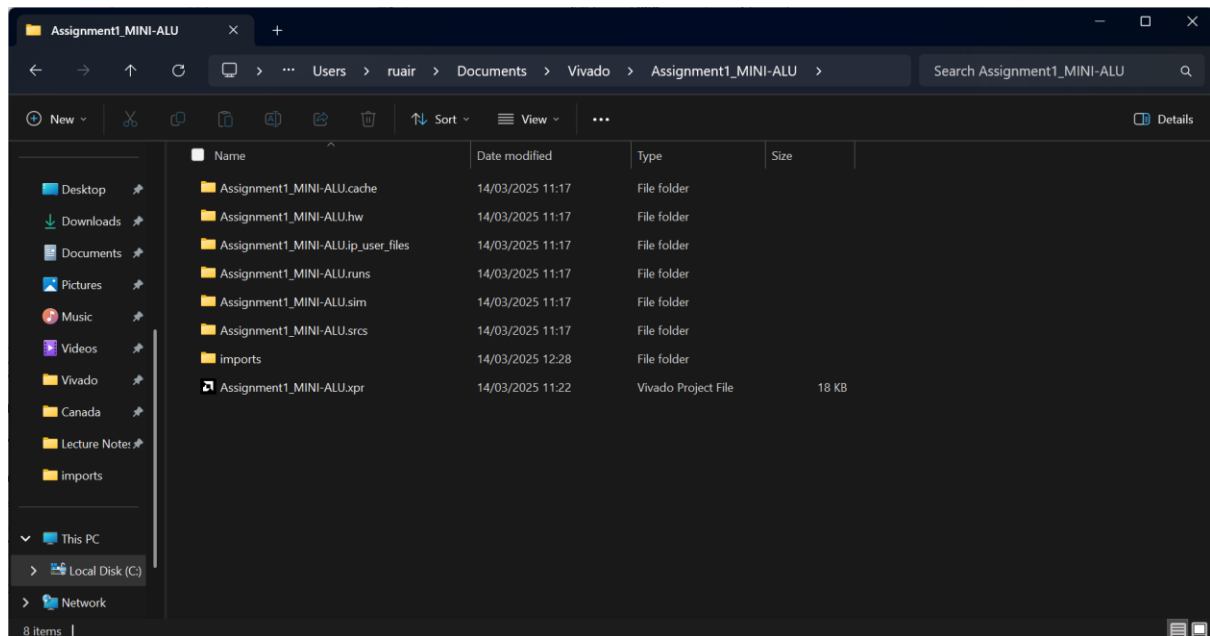


Figure 14. Project in file explorer.

## Appendix A:

# Digital System Design Lab E

Ruairi Mullally – 22336002

### Introduction:

This report examines the design and implementation of multiple Verilog modules, including their architecture and testing, which is the culmination of the previous labs. Emphasizing a bottom-up approach, the concepts of constructing digital modules with structural Verilog, the implementation flow, and the resource consumption of such modules will be discussed.

### Key takeaways on structural Verilog:

Structural Verilog pertains to the way that the hardware description is written. In writing structurally, one models the interconnections of physical pieces of hardware – logic gates and modules – rather than the behavioral constructs associated with other programming languages – if statements in C. This inherently promoted a bottom-up approach, as one starts out with simple modules, for example a 1-bit comparator that is written on a gate-by-gate level which is subsequently used to build a 2-bit comparator, etc. Structural Verilog denotes a clear map of the physical elements of the hardware, explicitly connecting elements, unlike a behavioral approach which infers the hardware through synthesis almost like a compiler.

For Lab B, I was given a 1-bit comparator, which was assumed to work. This simple module had 2 inputs: two 1-bit numbers, and 1 output: true or false. It output a 1 when the bits were equal and a 0 when the bits were not equal. This simple module could be used to build a 2-bit comparator, by using two instances of 1-bit comparators, and ANDing the result. This is a structural approach of writing the gate-level behavior of the circuit, while still enabling powerful encapsulation.

Then I derived the truth table for a 2-bit greater than circuit, got the sum of products form and used that to build the module structurally.

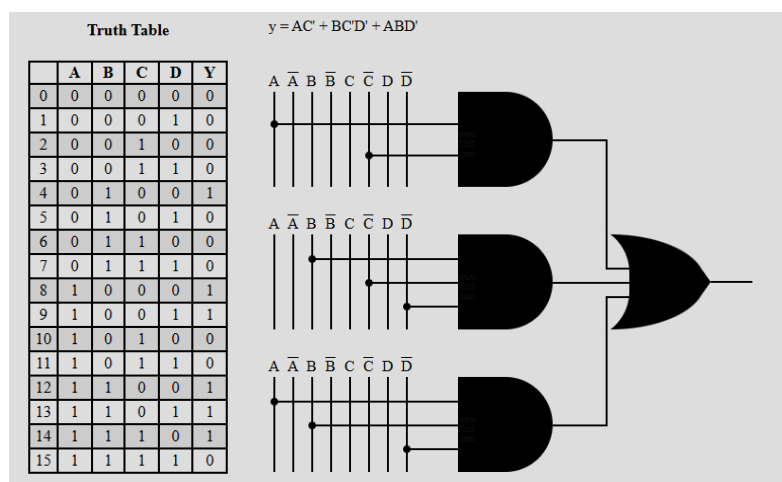


Figure 1. SOP derivation of 2-bit greater than comparator.

Using the SOP form, I could structurally write the gate level computation in Verilog:

```
module agtb2(  
    input wire[1:0] a, b,           // a and b are the two 2-bit numbers to c  
    output wire agtb                // single bit output. Should be high if a  
  
);  
    wire and1, and2, and3;  
    assign and1 = a[0] & ~b[0];  
    assign and2 = a[1] & ~b[0] & ~b[1];  
    assign and3 = a[0] & a[1] & ~b[1];  
  
    assign agtb = and1 + and2 + and3;  
endmodule
```

Figure 2. Structural 2-bit greater than comparator.

Using the 2-bit equal and 2-bit greater than modules, I could connect them to build an 8 bit greater or equal to module. The key takeaway is that structural Verilog allows for modular design of actual hardware which can be used to build more complex circuits, ensuring correctness at each step.

### Bottom-up design:

This bottom-up approach continued for lab 3, where I designed and built a 6-bit ripple adder/subtractor with a carry out and overflow detection. I was supplied with a 1-bit full adder which had three inputs: x, y, carry-in, and two outputs: carry-out and sum. First, I implemented a 6-bit adder by connecting six full-adders together. Each 1-bit adder's carry-out was connected to the carry-in of the next adder, enabling multi-bit addition. I built on this by implementing the carry-out and overflow detection by assigning the carry-out of the last adder to the carry-out wire and XORing the last and second last carry-outs, assigning the result to the overflow wire. Then, I implemented subtraction by flipping the sign of input y by XORing it with the select line and setting the carry-in of the adder to 1. By using the bottom-up approach, complex arithmetic could be achieved from fundamental building blocks, arriving at a full ripple adder/subtractor.

Port name	Type	Description
<b>x</b>	input	6-bit 2's complement number to add
<b>y</b>	input	6-bit 2's complement number to add
<b>sel</b>	input	Select functionality for add/subtract
<b>overflow</b>	output	Output flagging overflow in the sum output
<b>c_out</b>	output	MSB Carry out from the sum
<b>sum</b>	output	6-bit 2's complement sum of x and y

Figure 3. 6-bit ripple adder connections.

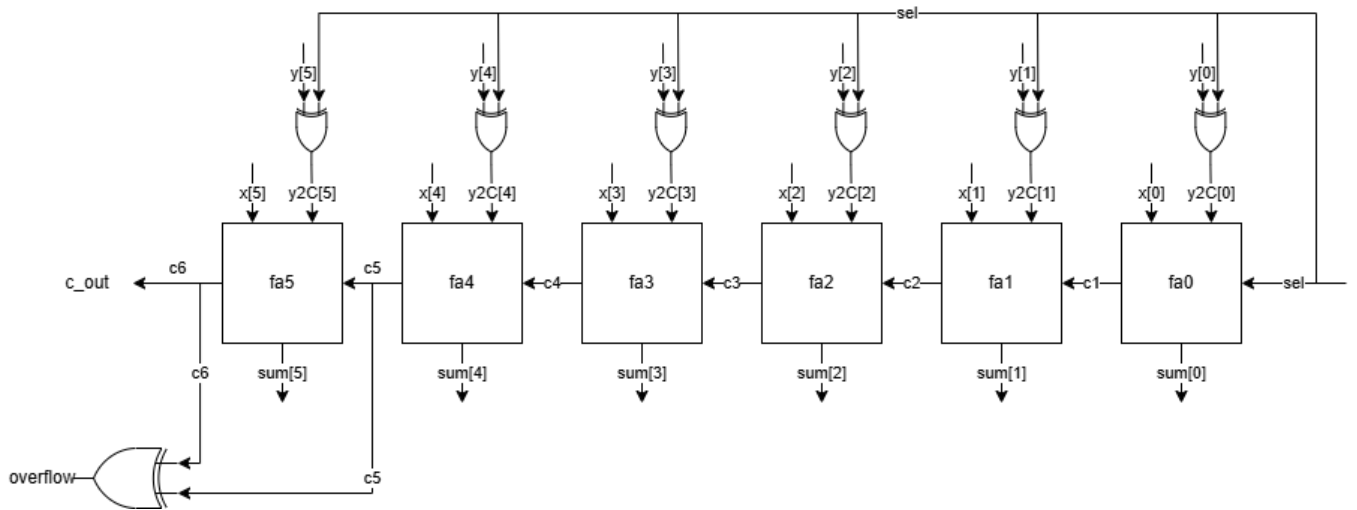


Figure 4. 6-bit ripple adder circuit diagram.

```

module b6_ripple_adder(
    input [5:0] x, y,
    input sel,
    output c_out, overflow,
    output [5:0] sum
);

    wire [5:0] y_2C;
    assign y_2C = y ^ {6{sel}};
    //carryout from each full adder
    wire c1, c2, c3, c4, c5, c6;

    FullAdder fa0(x[0], y_2C[0], sel, sum[0], c1);
    FullAdder fa1(x[1], y_2C[1], c1, sum[1], c2);
    FullAdder fa2(x[2], y_2C[2], c2, sum[2], c3);
    FullAdder fa3(x[3], y_2C[3], c3, sum[3], c4);
    FullAdder fa4(x[4], y_2C[4], c4, sum[4], c5);
    FullAdder fa5(x[5], y_2C[5], c5, sum[5], c6);

    assign c_out = c6;
    assign overflow = c6 ^ c5;

endmodule

```

Figure 5. 6-bit ripple adder Verilog code.

## Testing and Test Cases:

Table 1. 6-bit ripple adder test vectors.

Test Vector ID	sel	x	y	Expected c_out	Expected overflow	Expected sum
1:	0	6'b000010	6'b000011	0	0	6'b000101
2:	1	6'b000101	6'b000011	1	0	6'b000010
4:	1	6'b000000	6'b000001	0	0	6'b111111
5:	0	6'b010000	6'b010000	0	1	6'b100000
6:	1	6'b100000	6'b000001	1	1	6'b011111
7:	1	6'b000000	6'b000000	1	0	6'b000000
8:	0	6'b000000	6'b000000	0	0	6'b000000
9:	1	6'b100000	6'b000001	1	1	6'b011111
10:	0	6'b111111	6'b111111	1	0	6'b111110
11:	1	6'b110110	6'b101011	1	0	6'b001011
12:	1	6'b001010	6'b111100	0	0	6'b001110
13:	0	6'b001010	6'b111100	1	0	6'b000110

Inputs x and y	Sel	Function	Overflow	Carryout
Both positive	<del>High/Low</del>	<del>Sub/Add</del>	<del>Present/absent</del>	<del>High/Low</del>
Both negative	<del>High/Low</del>	<del>Sub/Add</del>	<del>Present/absent</del>	<del>High/Low</del>
One positive/one negative	<del>High/Low</del>	<del>Sub/Add</del>	<del>Present/absent???</del>	<del>High/Low</del>
All zero	<del>High/Low</del>	<del>Sub/Add</del>	<del>??</del>	<del>??</del>

Figure 6. 6-bit ripple adder test vector coverage.

While it was not feasible to exhaustively cover all possible inputs with test vectors, I aimed to cover as many combinations as possible: positive and negative numbers, overflow, carry-out and edge cases. I wanted a vector that covered every field of the table in figure 6. For example, test vectors 7 and 8 cover the cases of addition and subtraction when both numbers are zero, each generating a different carryout. I was able to achieve this with 13 test vectors. Below in figure 7 and figure 8 are the results of these tests. Ideally, with more time, more test vectors would be added such that only one parameter changes at a time, individually verifying each component. I would have liked to add all overflow cases (negative + negative, positive + positive, subtraction turning a negative into a positive, etc.). Nonetheless, the vectors I have are suitably exhaustive.

Time	Status	test_in0	test_in1	test_sel	test_out	test_c_out	test_overflow	expected_out	expected_c_out	expected_overflow
100	PASS:	000010	000011	0	000101	0	0	000101	0	0
300	PASS:	000101	000011	1	000010	1	0	000010	1	0
500	PASS:	111111	000001	0	000000	1	0	000000	1	0
700	PASS:	000000	000001	1	111111	0	0	111111	0	0
900	PASS:	010000	010000	0	100000	0	1	100000	0	1
1100	PASS:	100000	000001	1	011111	1	1	011111	1	1
1300	PASS:	000000	000000	1	000000	1	0	000000	1	0
1500	PASS:	000000	000000	0	000000	0	0	000000	0	0
1700	PASS:	100000	000001	1	011111	1	1	011111	1	1
1900	PASS:	111111	111111	0	111110	1	0	111110	1	0
2100	PASS:	110110	110111	1	111111	0	0	111111	0	0
2300	PASS:	001010	111100	1	001110	0	0	001110	0	0
2500	PASS:	001010	111100	0	000110	1	0	000110	1	0
Total Tests: 13, Passed: 13, Failed: 0										
Test Pass Percentage: 100.00%										

Figure 7. 6-bit ripple adder test vector results.

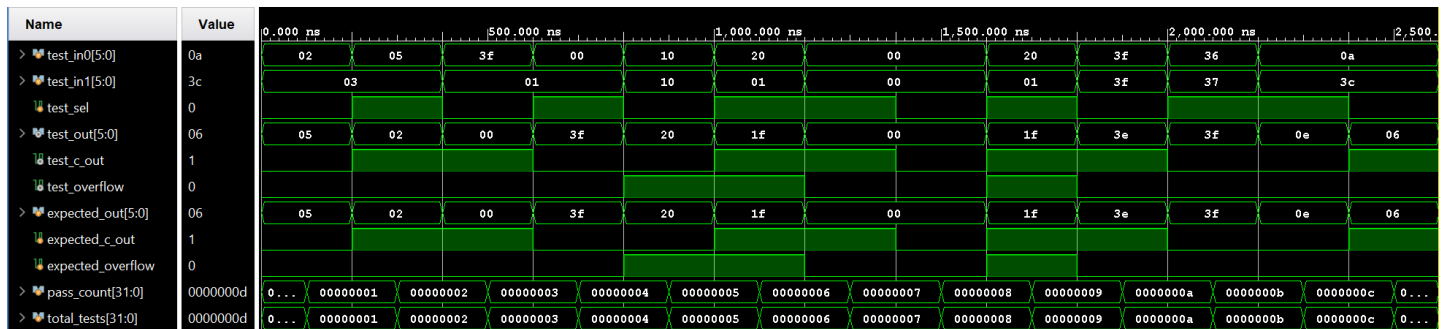


Figure 8. 6-bit ripple adder test vector waveform results.

## Implementation flow:

The implementation flow of a project begins on a conceptual level, where one imagines the desired functionality and deliverables of a circuit. The design specifications such as input requirements, functional requirements and constraints are considered. Next the high-level architecture is designed: what modules are needed and how will they interconnect. The next step in the implementation flow is programming, using a hardware description language to implement the desired design. Next the implementation is verified by writing a testbench to test the module and running a simulation on the testbench. Once the design is verified. It is synthesized into its gate-level representation. Next is the implementation phase, where inputs and outputs are mapped to physical connections on the target board (FPGA). A bitstream of the implementation is generated and uploaded to the board.

## Basys-3 Adder/Subtractor implementation consumption:

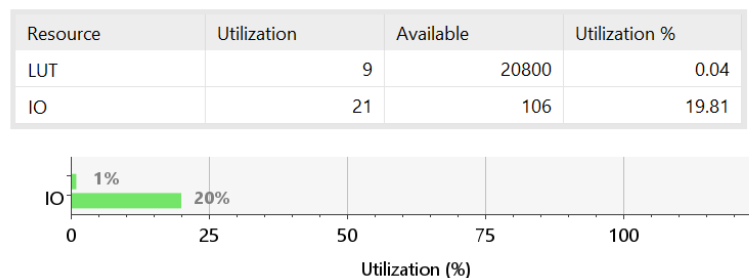


Figure 9. 6-bit ripple adder utilization stats.

The computational overhead of this design is at low utilization of 0.04% with only 9 look up tables in use. In comparison, input/output utilization is quite high with 20% of IO ports in use. This represents the two 6-bit numbers, the sum output, the select and carry-out and the overflow. There are no registers or flip-flops reported as it is a purely combinatorial design.