Digital System Design Lab E

Ruairi Mullally – 22336002

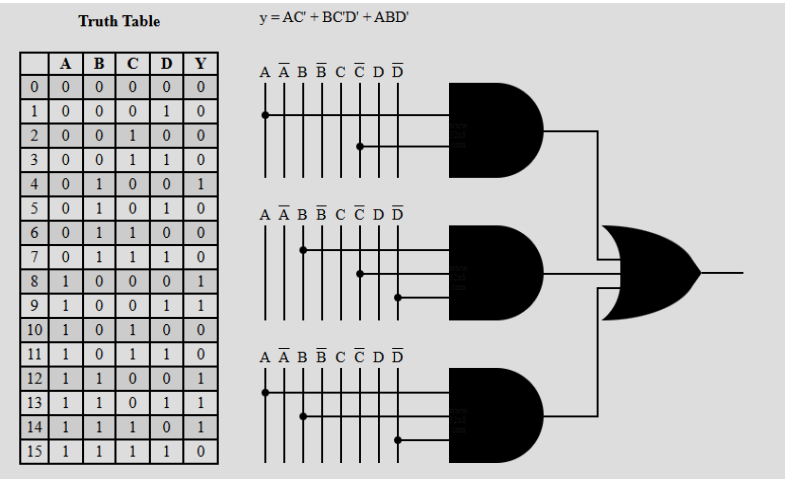
Introduction:
This report examines the design and implementation of multiple Verilog modules, including their architecture and testing, which is the culmination of the previous labs. Emphasizing a bottom-up approach, the concepts of constructing digital modules with structural Verilog, the implementation 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.



	A	B	C	D	Y
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	0
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	0

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

Figure 3. 6-bit ripple adder connections.

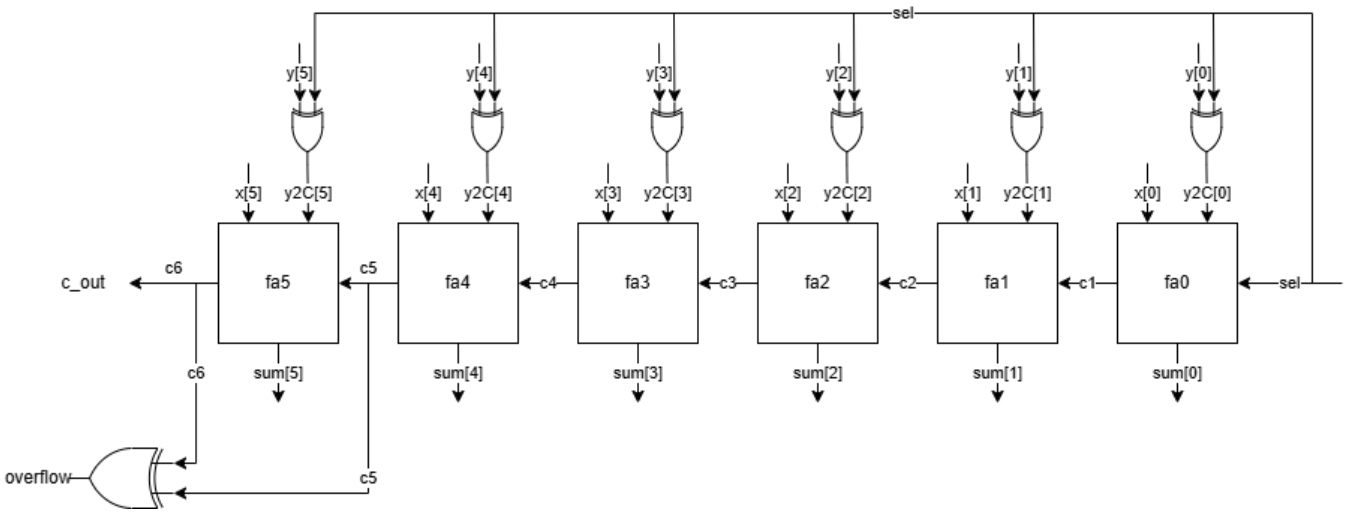


Figure 4. 6-bit ripple adder circuit diagram.

```
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	~~High/low~~	~~Sub/Add~~	~~Present/absent~~	~~High/low~~
Both negative	~~High/low~~	~~Sub/Add~~	~~Present/absent~~	~~High/low~~
One positive/one negative	~~High/low~~	~~Sub/Add~~	~~Present/absent???~~	~~High/low~~
All zero	~~High/low~~	~~Sub/Add~~	??	??

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.

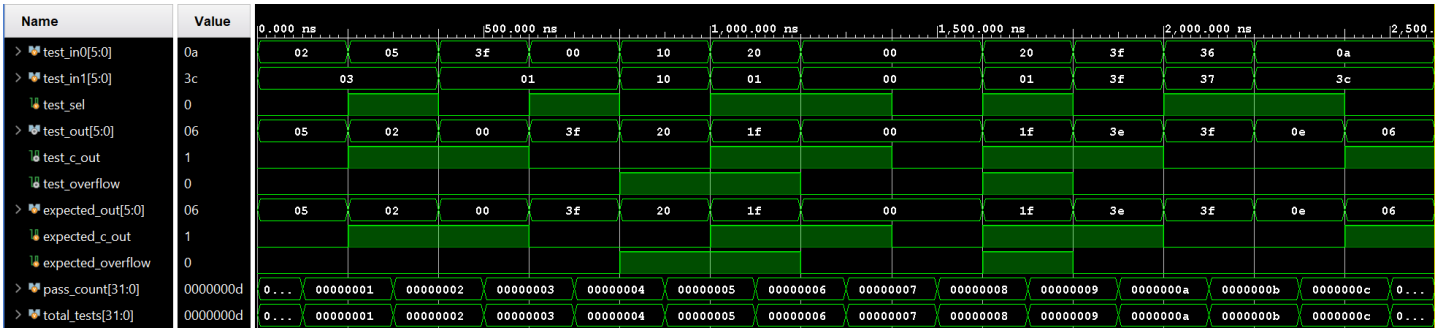


Figure 8. 6-bit ripple adder test vector waveform results.

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:

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

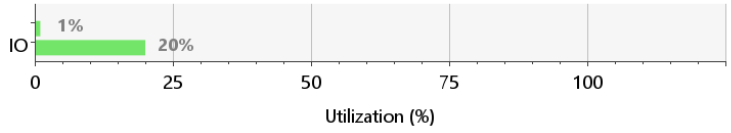


Figure 9. 6-bit ripple adder utilization stats.

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.