Lab 5 EC413

Members: Pree Simphliphan, Rayan Syed

Note: for the error flags in modules MOV, NOR, NAND, AND, error flag = 1 means correct output.

In this lab, we created the ALU described in the lab document with all the listed components. The specifics of the modules will be listed below:

AND/OR GATES:

Nand and Nor gates had to be used as primitives for these and/or gates used throughout the lab. Using a truth table, we came up with the following implementations:

```
module and_s(out,a,b);

input a,b;
output out;

wire out1, out2;

nand nand1 (out1,a,b);
nand nand2 (out2,a,b);
nand nand3 (out,out1,out2);

endmodule
```

```
module or_s(out,a,b);

input a,b;
output out;

wire out1, out2;

nand nand1 (out1,a,a);
nand nand2 (out2,b,b);
nand nand3 (out,out1,out2);


endmodule
```

These both are quite self-explanatory.

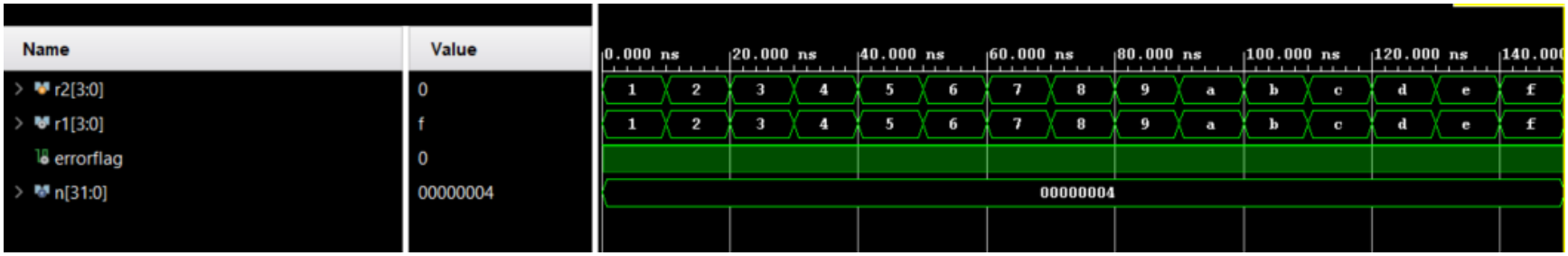
Now that the base of the whole project has been described, we will get into the specifics of the main ALU components:

MOV:

```
module mov #(parameter n = 8) (r2, r1);
input [n-1:0] r2;
output [n-1:0] r1;

genvar i;
generate
    for (i=0; i<n; i=i+1)
    begin
        and_s equal(r1[i],r2[i],1);
    end
endgenerate

endmodule
```

For this module, we very simply made a = b by making the output = a & a, using the and module we created earlier. We used generate to parametrize this function so it can work for as many bits as possible, as can clearly be seen here. We will not mention this again for the rest of the lab, as it should now be obvious that all out modules utilize this.

Testbench for MOV:



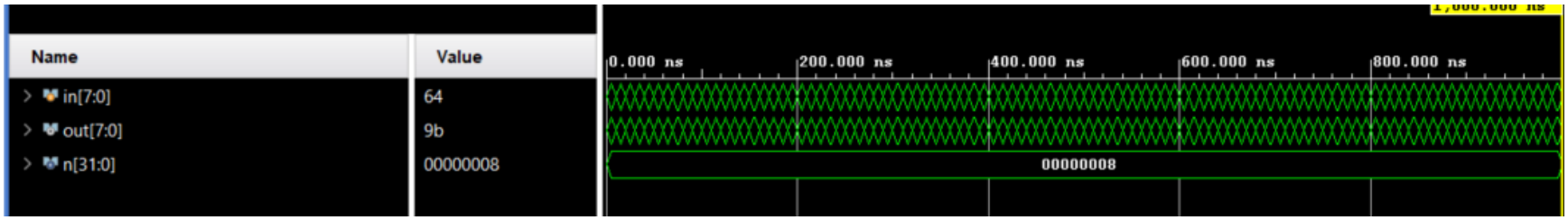
NOT:

```
module not_bitwise_par #(parameter n = 8)(in,out);

input [n-1:0] in;
output [n-1:0] out;

genvar i;
generate
    for (i=0; i<n; i=i+1)
    begin
        not_s not_s1(out[i],in[i]);
    end
endgenerate

endmodule
```

```
module not_s(out,a);

input a;
output out;

nand nand1 (out,a,a);

endmodule
```

This module (left) called a not module for each bit. The simple not module can be seen on the right. It is simply just an input getting *nand*ed with itself. Once again, nand is clearly a primitive here.

Testbench for NOT:



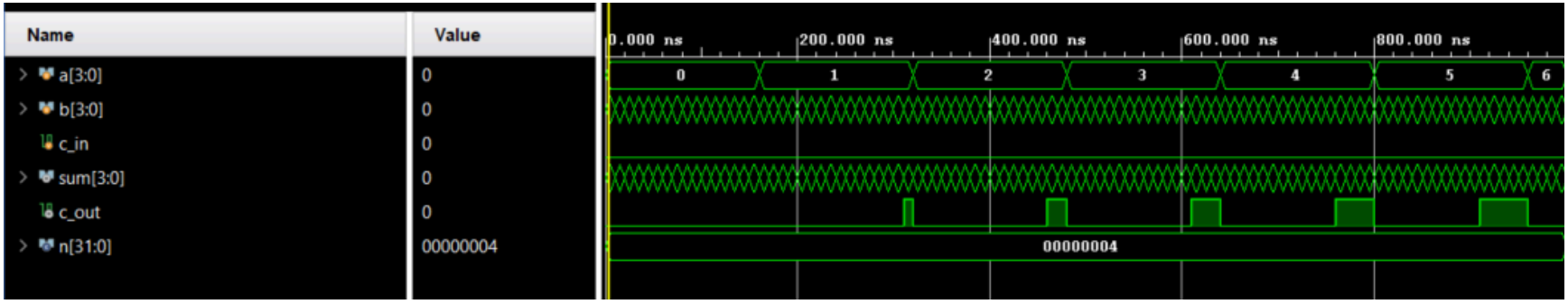
No error flag was needed here, as the desired output is obvious enough as is.

ADD:

For the adder, things were a little more complicated. We will let the code speak for itself. This was easy, as a similar assignment was done in EC311, so we knew exactly how to do this. The full adder module called on each loop was provided with the assignment.

```
module FA_par #(parameter n = 8) (c_out, a, b, sum, c_in);

input c_in;
input [n-1:0] a, b;
output  [n-1:0] sum;
output c_out;

wire [n:0]carry;
assign  carry[0] = c_in;
assign c_out = carry[n];
genvar i;
generate
    for (i=0; i<n; i=i+1)
    begin
        FA_str full(carry[i+1],sum[i], a[i], b[i], carry[i]);
    end
endgenerate


endmodule
```

Testbench for ADD:

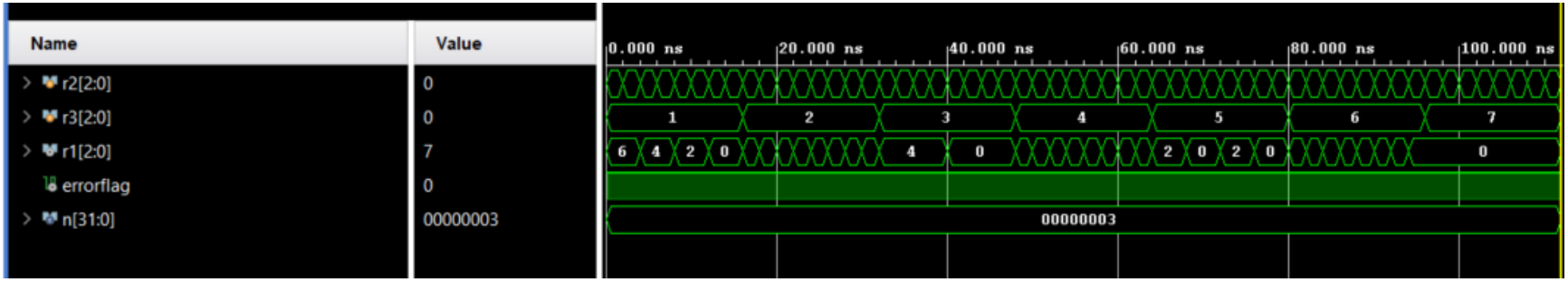


NOR:

The NOR module was also very easy, as NOR is a valid primitive in this assignment. So, we just called it n-times.

```
module nor_par #(parameter n = 4) (r1,r2, r3);

output [n-1:0] r1;
input [n-1:0] r2, r3;

genvar i;
generate
    for (i=0; i<n; i=i+1)
    begin
        nor no(r1[i],r2[i],r3[i]);
    end
endgenerate

endmodule
```

Testbench for NOR:



SUB:

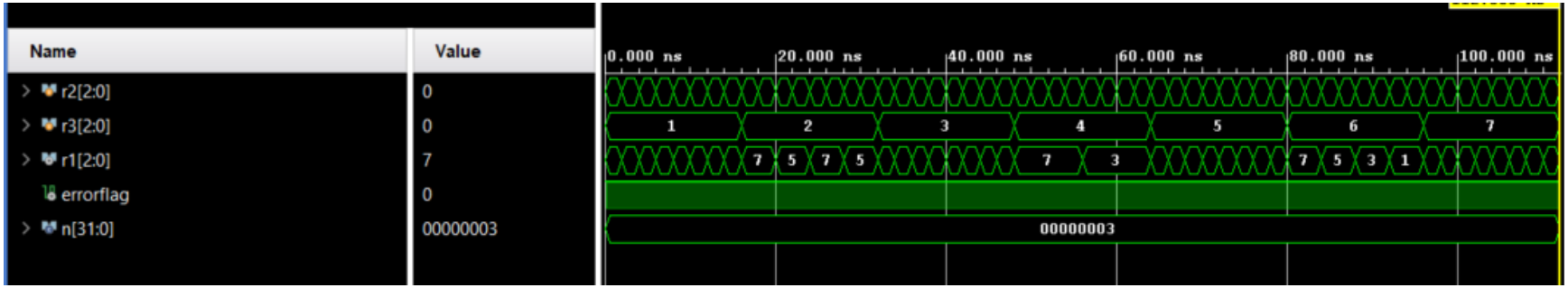
For SUB, we do not have a module nor testbench to display. We already have a strong foundation in 2's complement and understood that we can simply invert the input getting subtracted and set c_in to 1 for the adder module in order to perform subtraction. This is done in out ALU and we know it will work, as that is common sense.

NAND:

This module was similar to the NOR module, as we just ran NAND n-times.

```
module nand_par #(parameter n = 4) (r1,r2,r3);
input [n-1:0] r2,r3;
output[n-1:0] r1;

genvar i;
generate
    for (i=0; i<n; i=i+1)
    begin
        nand no(r1[i],r2[i],r3[i]);
    end
endgenerate

endmodule
```

Testbench for NAND:

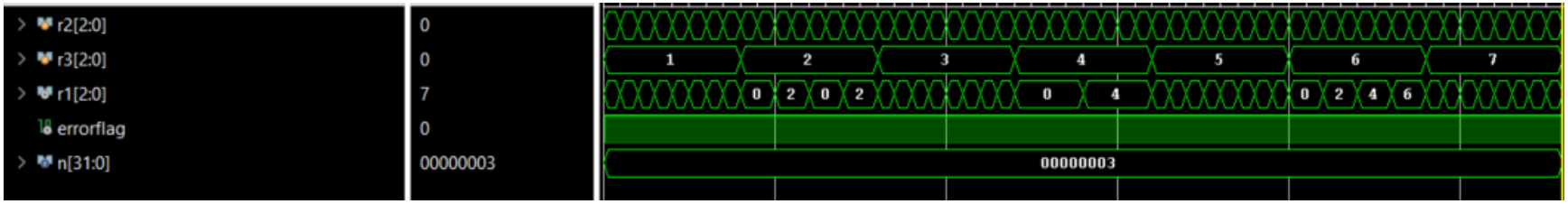


AND:

As we have already created an and module made from our primitives, this was easy, as we just called that module n-times.

```
module and_par #(parameter n = 4) (r1,r2,r3);

input [n-1:0] r2,r3;
output [n-1:0] r1;

genvar i;
generate
    for (i=0; i<n; i=i+1)
    begin
        and_s an(r1[i],r2[i],r3[i]);
    end
endgenerate

endmodule
```

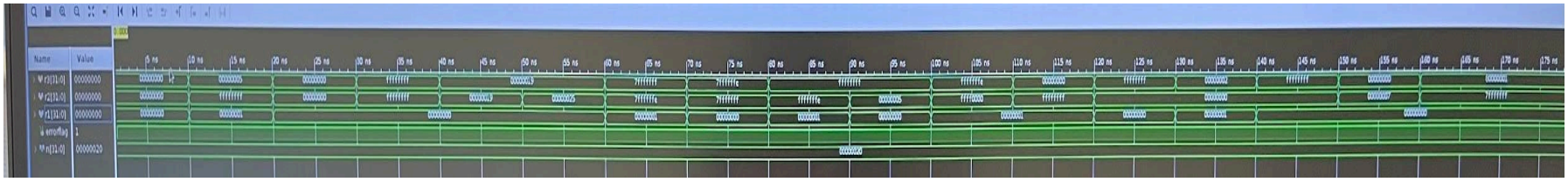
Testbench for AND:



SLT:

The SLT module first performs r2-r3 using structural Verilog. We were allowed to use behavioral for the second part, so we made a simple case statement to check the cases provided in the prelab discussion slides in order to assign the correct result.

```
module SLT_par #(parameter n = 4) (r1,r2,r3);
input  [n-1:0] r2,r3;
output [n-1:0] r1;

reg result;

//SUB
wire [n-1:0] r_3;
wire [n-1:0] subresult;
not_bitwise_par #(n) notr3 (r3,r_3);            //flip r3 since its being subtracted
FA_par #(n) SUB (c_out_2, r2, r_3, subresult, 1);      //c_in = 1 for subtraction, these two steps perform 2's compliment

always @ *
begin
    if(subresult == 0)
        result = 0;
    else
        case ({r2[n-1],r3[n-1]})
            00: result = subresult[n-1];
            01: result = r2[n-1];
            10: result = r2[n-1];
            11: result = subresult[n-1];
        endcase
end

assign r1 = result;

endmodule
```



ALU - PUTTING IT ALL TOGETHER:

Rather than attaching a screenshot, a description of the ALU will be provided instead. For the actual R1 outputs of the 8 possible functions, all 8 modules were called and 8 wires were created to hold their respective outputs in the ALU top module. A mux was then provided to determine the correct output to send to a register based on the op code. The carry out logic also uses a mux and has a similar system to the R1 output. The zero and overflow are calculated separately and assigned separately for ease of assignment.

The four results (R1, zero_flag, c_out, overflow) are then sent into four registers and the outputs of those registers are the outputs of the final ALU top module. This makes sure they are only updated at the clk's posedge. The given testbench was adapted to create our own testbench that reflects our approach on this lab.

Finally, we modified the verification module for comparing our ALU with behavioural verilog to check if c_out == c_outverify, zero = zero_verify, overflow == overflow_verify, and R1 == R1_verify, when one of the conditions failed, it will set errorflag as 1.

Testbench for the ALU - NOTE THE ERRORFLAG: