EC311 - Logic Design Lab

Lab 1: Design 4-bit binar Adder-Subtractor and ALU

Student Name: Giacomo Cappelletto
Student ID: U91023753
Date: October 10, 2025
Section: A2

---

1. Objective

The primary objective of this lab is to design and implement a 4-bit ALU, which accepts two 4-bit inputs and performs addition, multiplication, concatenation and left shift. This ALU must also accept a 2-bit control signal to select the operation to be performed. The output of the ALU must be and 8-bit result signal.

The secondary objective of this lab is to design and implement a 4-bit adder-subtractor, which accepts two 4-bit inputs and performs addition and subtraction. This adder-subtractor must also accept a 1-bit control signal to select the operation to be performed and output a 1-bit overflow signal other than the 4-bit result and carry signal. This adder-subtractor must be implemented using an instantiation of multiple 4-bit full adders, which are to be implemented with structural verilog. Futhermore, these full adders must be implemented using an instantiation of multiple half adders, also to be implemented using structural verilog.

Furthermore, each submodule of the ALU and the adder-subtractor must be extensive-lytested using a testbench, which verifies the functionality of each submodule by sweeping through all possible input combinations and checking the output.

2. Methodology

2.1. Design Approach

2.1.1. Adder and Subtractor Implementation

The adder and subtractor implementation was deveoped with the end goal of a ripple-carry adder constituted of multiple full adders, each of which is instantiated with two half adders. The half adders are implemented using structural verilog with gate-level primitives. The full adders are implemented using structural verilog with an instantiation of two half adders.

The adder-subtractor is implemented using structural verilog with an instantiation of four full adders, and additional logic to handle the subtraction operation and overflow detection. We XOR the each bit of the second operand with the subtraction control signal to conditionally invert the second operand for the subtraction operation, and feed the carry-in of the first full adder with the subtraction control signal to complete the 2′s complement arithmetic process. We also detect overflow by checking if the carry-in of the most significant full adder is different from the carry-out of the most significant full adder, which is accomplished by exposing the third carry-out of the full adders in the adder module (to avoid recalculating it).

2.1.2. ALU Implementation

Similarly to the adder-subtractor, the ALU is implemented with a hierarchy of modules, but using behavioral verilog instead of structural verilog. The 4 main submodules are the addition, multiplication, concatenation and left shift. Each of these submodules are implemented using behavioral verilog. To combine them, a 4:1 8-bit multiplexer is used to select the output of the desired submodule based on the 2-bit control signal.

2.2. Verilog Implementation

2.2.1. Half Adder Module

The half adder is the fundamental building block, implemented using structural Verilog with gate-level primitives.

```
module half_adder(
    input wire a,
    input wire b,
    output wire sum,
    output wire carry
    );
    xor G1(sum, a, b);
    and G2(carry, a, b);
endmodule
```

2.2.2. Full Adder Module

The full adder is constructed by instantiating two half adders and combining their outputs, and is the building block of the adder-subtractor.

```
module full_adder(
  input  wire a,
  input  wire b,
  input  wire cin,
  output wire sum,
  output wire cout
);
  wire s1, c1, c2;
  half_adder ha0(.a(a),  .b(b),   .sum(s1),  .carry(c1));
  half_adder ha1(.a(s1), .b(cin), .sum(sum), .carry(c2));
  or G3(cout, c1, c2);
endmodule
```

2.2.3. 4-bit Adder Module

The 4-bit adder is implemented using an instantiation of four full adders in a ripple-carry configuration. Notice that we expose the third carry-out of the full adders to detect overflow further on in the adder-subtractor.

```
module adder4(
  input  wire [3:0] A,
  input  wire [3:0] B,
  input  wire       cin,
  output wire [3:0] S,
  // For overflow detection
  output wire        C3,
  output wire C4
);
  wire c1, c2, c3;
  full_adder fa0(.a(A[0]), .b(B[0]), .cin(cin), .sum(S[0]), .cout(c1));
  full_adder fa1(.a(A[1]), .b(B[1]), .cin(c1),  .sum(S[1]), .cout(c2));
  full_adder fa2(.a(A[2]), .b(B[2]), .cin(c2),  .sum(S[2]), .cout(c3));
  // Expose C4
  full_adder fa3(.a(A[3]), .b(B[3]), .cin(c3),  .sum(S[3]), .cout(C4));
  // Expose C3
  assign C3 = c3;
endmodule
```

2.2.4. 4-bit Adder-Subtractor Module

The adder-subtractor instantiates the previously defined 4-bit adder module, and adds additional logic to handle the subtraction operation and overflow detection. We XOR the each bit of the second operand with the subtraction control signal to conditionally invert the second operand for the subtraction operation, and feed the carry-in of the first full adder with the subtraction control signal to complete the 2′s complement arithmetic. We also detect overflow by checking if the carry-in of the most significant full adder is different from the carry-out of the most significant full adder, which was previously exposed in the adder module.

```
module addsub4(
    input wire  [3:0] A,
    input wire  [3:0] B,
    input wire        m,
    output wire [3:0] S,
    output wire       cout,
    output wire       vout
    );
  wire [3:0] Bx;
  xor x0(Bx[0], B[0], m);
  xor x1(Bx[1], B[1], m);
  xor x2(Bx[2], B[2], m);
  xor x3(Bx[3], B[3], m);
  wire C3, C4;
  // cin = m accounts for 2s comp +1
  adder4 add4(.A(A), .B(Bx), .cin(m), .S(S), .C3(C3), .C4(C4));
  assign cout = C4;
  xor overf(vout, C3, C4);
endmodule
```

2.2.5. ArithmeticSubmodules of the ALU Module

The 4 arithmetic submodules are the addition, multiplication, concatenation and left shift. Each of these submodules are implemented using behavioral verilog, which is a higher level of abstraction than structural verilog, and therfore allows for better readability and lower possibilities of errors.

```
module add(input wire [3:0] A, B, output wire [7:0] Y);

    assign Y = {4'b0000, A} + {4'b0000, B};

endmodule
module mult(input wire [3:0] A, B, output wire [7:0] Y);

    assign Y = A * B;

endmodule
module concat(input wire [3:0] A, B, output wire [7:0] Y);

    assign Y = {A,B};

endmodule
module shift(input wire [3:0] A, B, output wire [7:0] Y);

    assign Y = (B > 7) ? 8'd0 : ({4'b0000, A} << B);

endmodule
```

2.2.6. Multiplexer Module

The multiplexer selects the output of the desired arithmetic submodule based on the 2-bit control signal, as per the lab instructions. The 4 input signals as well as the output are 8-bit signals, making this a 4:1 8-bit multiplexer.

```
module mux(input wire [7:0] ADD, MULT, CONCAT, SHIFT, input wire [1:0] S, output
reg [7:0] Y);
    always @* begin
        case (S)
            2'b00: Y = CONCAT;
            2'b01: Y = ADD;
            2'b10: Y = SHIFT;
            2'b11: Y = MULT;
            default Y = 8'd0;
        endcase
    end
endmodule
```

2.2.7. ALU Module

Since most of the modules are already implemented, all that is left is to wire the arithmetic submodules to the multiplexer and the multiplexer to the output.

```
module alu(
    input wire  [3:0] A,
    input wire  [3:0] B,
    input wire  [1:0] S,
    output wire [7:0] Y
    );

    wire [7:0] ADD, MULT, CONCAT, SHIFT;

    concat c0(.A(A), .B(B), .Y(CONCAT));
    add    a0(.A(A), .B(B), .Y(ADD)    );
    mult   m0(.A(A), .B(B), .Y(MULT)   );
    shift  s0(.A(A), .B(B), .Y(SHIFT) );

    mux
muxY(.ADD(ADD), .MULT(MULT), .CONCAT(CONCAT), .SHIFT(SHIFT), .S(S), .Y(Y));
endmodule
```

2.3. Simulation and Testing

Comprehensive testbenches were developed for each module to ensure functionality before integration. Each testbench was developed to test all possible input combinations for the module, and to verify the functionality of the module matcheds the expected behavior. For conciseness purposes, only the testbenches for the adder-subtractor and the ALU are included here.

2.3.1. Adder-Subtractor Testbench

The testbench for the adder-subtractor checks functionality for all 512 possible combinations of the 3 input signals (A, B, and m) against outputs S, cout and vout on an instantiation of the addsub4 module. We compute the expected outputs using behavioral verilog, using ripple carry logic in order to be able to check C3 and C4, and consequently check the overflow signal. After the loop the passed and failed tests are reported.

(See Appendix A for the complete testbench)

2.3.2. ALU Testbench

The testbench for the ALU checks functionality for all 1024 possible combinations of the 3 input signals (S, A, and B) against the 8-bit output Y on an instantiation of the alu module. We compute the expected output using behavioral verilog reference functions for each operation—concatenation, zero-extended addition, bounded left shift, and 4x4 multiply—selected via a case on S. The testbench includes a few directed smoke tests, then exhaustively iterates inputs, guards against X/Z on inputs and output, and compares Y to the reference using case-inequality to catch X/Z mismatches. After the loop the passed and failed tests are reported.

(See Appendix A for the complete testbench)

3. Observation

Present your results with supporting evidence. Include screenshots, timing diagrams, waveforms, and schematics.

3.1. Simulation Results

Document waveform observations from simulation.

To include a figure:

```
#figure(
  image("simulation_waveform.png", width: 90%),
  caption: [Simulation waveform showing counter operation with reset and enable
signals]
)
```

Describe what the waveforms show:

- Signal transitions
- Timing relationships
- Functional verification points

3.2. Synthesis Results

Report resource utilization and synthesis statistics.

Example table:

Resource	Utilization	Percentage
LUTs	25	0.12%
FFs	16	0.03%
Block RAM	0	0.00%
DSPs	0	0.00%

Table 1: Resource utilization on Artix-7 XC7A35T

Note any warnings or optimization results.

3.3. Timing Analysis

Present timing analysis results.

Include:

- Maximum achievable frequency
- Setup/hold time analysis
- Critical path identification
- Any timing violations and resolutions

Example: *Timing analysis shows the design meets timing at 100 MHz with a slack of 3.2 ns. No setup or hold violations were detected.*

3.4. RTL Schematic

Show the RTL schematic generated by Vivado.

```
#figure(
  image("rtl_schematic.png", width: 95%),
  caption: [RTL schematic showing the synthesized design hierarchy]
)
```

3.5. Hardware Testing (if applicable)

Document physical FPGA testing results.

Include:

- Board setup and configuration
- Test procedure
- Observed behavior
- Photos/screenshots if helpful

Example: *The design was successfully programmed onto the Basys3 board. The counter increments on each clock pulse when the enable switch is high, and resets to 0 when the reset button is pressed.*

```
#figure(
  image("hardware_demo.jpg", width: 70%),
  caption: [Basys3 FPGA board running the counter implementation]
)
```

4. Conclusion

Summarize findings and reflect on the laboratory experience.

4.1. Summary of Results

Briefly restate accomplishments and whether objectives were met.

Example: *This laboratory successfully demonstrated the design, simulation, synthesis, and implementation of a 4-bit synchronous counter using Verilog HDL and Vivado. All functional requirements were verified through simulation and hardware testing.*

4.2. Challenges and Solutions

Discuss difficulties encountered and their resolutions.

Example issues:

- Initial synthesis warnings due to incomplete sensitivity lists → resolved by using `always @(posedge clk)`
- Timing violations at 150 MHz → added pipeline stage to reduce critical path
- Simulation mismatch → fixed improper reset logic

4.3. Learning Outcomes

Reflect on what was learned from this lab.

Consider:

- Conceptual understanding gained
- Tool proficiency developed
- Design techniques learned
- Debugging skills improved

Example: *This lab reinforced understanding of synchronous sequential logic design and provided hands-on experience with the complete FPGA development workflow from RTL to bitstream.*

4.4. Future Improvements

Suggest potential enhancements or alternative approaches.

Example:

- Add a configurable counting direction (up/down counter)
- Implement variable count step size
- Add seven-segment display output
- Explore power optimization techniques

Tips for a Successful Lab Report

1. Observation Section - Including Images:

To include images in your report, save them in the same directory as your report.typ file and use:

```
#figure(
  image("your_image.png", width: 90%),
  caption: [Your descriptive caption here]
)
```

2. Common Image Types to Include:

- Simulation waveforms (from Vivado Simulator)
- RTL schematics (from Vivado)
- Timing reports (screenshot from Vivado)
- Utilization reports (screenshot from Vivado)
- Photos of hardware setup (from your phone/camera)

3. Recommended Workflow:

1. Write your Verilog code in Vivado
2. Run simulations and take waveform screenshots
3. Synthesize and capture resource utilization
4. Run implementation and capture timing results
5. Export RTL schematic as PNG/PDF
6. Test on hardware and take photos/videos
7. Fill in this template with your findings
8. Add all images to the report
9. Compile to PDF: `typst compile report.typ`
10. Review PDF to ensure it's under 5 pages

4. Page Management:

If your report exceeds 5 pages:
- Make images slightly smaller (adjust width percentages)
- Be more concise in descriptions
- Remove redundant information
- Use tables efficiently
- Consider combining related figures

5. Compiling to PDF:

From terminal in the lab1 directory:

```
typst compile report.typ
```

This creates `report.pdf` ready for BlackBoard submission.

Delete all instruction pages and keep report under 5 pages

5. Appendix A: Testbench Listings

5.1. Adder-Subtractor Testbench (`tb_addsub4.v`)

```
`timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/03/2025 01:27:34 PM
// Design Name:
// Module Name: tb_addsub4
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module tb_addsub4;
  reg  [3:0] A, B;
  reg        m;
  wire [3:0] S;
  wire       cout, vout;
  reg  [3:0] S_exp;
  reg        C3_exp, C4_exp, V_exp;

  addsub4 add4(.A(A), .B(B), .m(m), .S(S), .cout(cout), .vout(vout));

  task calc_expected;
    reg [3:0] Bx;
    reg c1, c2;
    begin
      Bx = B ^ {4{m}};
      {c1, S_exp[0]} = A[0] + Bx[0] + m;
      {c2, S_exp[1]} = A[1] + Bx[1] + c1;
      {C3_exp, S_exp[2]} = A[2] + Bx[2] + c2;
      {C4_exp, S_exp[3]} = A[3] + Bx[3] + C3_exp;
      V_exp = C3_exp ^ C4_exp; //overflow
    end
  endtask

  integer k, errors;
  initial begin
    errors = 0;
    {m, A, B} = 9'b0;

    for (k = 0; k < 512; k = k + 1) begin
      {m, A, B} = k[8:0];
      #1;
      calc_expected();

      if (S !== S_exp || cout !== C4_exp || vout !== V_exp) begin
        errors = errors + 1;
        $display("MISMATCH k=%0d  m=%b A=%b B=%b | DUT S=%b cout=%b vout=%b  EXP
S=%b C4=%b V=%b",
                 k, m, A, B, S, cout, vout, S_exp, C4_exp, V_exp);
        //Stop on error
        $stop;
      end
      #9;
    end

    if (errors == 0) $display("ADDSUB4: all 512 passed.");
    else             $display("ADDSUB4 FAIL: %0d errors.", errors);
    $finish;
  end
endmodule
```

5.2. ALU Testbench (`tb_alu.v`)

```
`timescale 1ns/1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/03/2025 02:44:51 PM
// Design Name:
// Module Name: tb_alu
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
```

```
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module tb_alu;
  reg  [3:0] A, B;
  reg  [1:0] S;
  wire [7:0] Y;

  alu dut(.A(A), .B(B), .S(S), .Y(Y));

  function [7:0] f_add(input [3:0] a, b); f_add = {4'b0,a} + {4'b0,b};
endfunction
  function [7:0] f_mul(input [3:0] a, b); f_mul = a * b;
endfunction
  function [7:0] f_cat(input [3:0] a, b); f_cat = {a,b};
endfunction
  function [7:0] f_shf(input [3:0] a, b); f_shf = (b>7) ? 8'd0 : ({4'b0,a} <<
b); endfunction

  integer k, errors;
  reg [7:0] Y_exp;

  function has_xz8(input [7:0] v); has_xz8 = (^v === 1'bx); endfunction
  function has_xz4(input [3:0] v); has_xz4 = (^v === 1'bx); endfunction
  function has_xz2(input [1:0] v); has_xz2 = (^v === 1'bx); endfunction

  initial begin
    errors = 0;

    A=4'hF; B=4'hF; S=2'b01; #1 $display("ADD FF+FF -> Y=%h (expect 1E)", Y);
    A=4'h8; B=4'h8; S=2'b10; #1 $display("SHF A=8,B=8 -> Y=%h (expect 00)", Y);
    A=4'hF; B=4'hF; S=2'b11; #1 $display("MUL F*F -> Y=%h (expect E1)", Y);

    for (k = 0; k < 1024; k = k + 1) begin
      {S, A, B} = k[9:0];

      #1;
      if (has_xz2(S) || has_xz4(A) || has_xz4(B)) begin
        $fatal(1, "X/Z on inputs at k=%0d S=%b A=%b B=%b", k, S, A, B);
      end

      case (S)
```

```
        2'b00: Y_exp = f_cat(A,B);
        2'b01: Y_exp = f_add(A,B);
        2'b10: Y_exp = f_shf(A,B);
        2'b11: Y_exp = f_mul(A,B);
      endcase

      if (Y !== Y_exp) begin
        errors = errors + 1;
        $display("MISMATCH k=%0d  S=%b A=%h B=%h | DUT Y=%h  EXP Y=%h",
                 k, S, A, B, Y, Y_exp);
      end

      if (has_xz8(Y)) begin
        errors = errors + 1;
        $fatal(1, "Output X/Z at k=%0d  S=%b A=%h B=%h  Y=%h", k, S, A, B, Y);
      end

      #9;
    end

    if (errors==0) $display("PASS: all 1024 ALU vectors matched.");
    else           $display("FAIL: %0d mismatches.", errors);
    $finish;
  end
endmodule
```