EC311 - Logic Design Lab

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

Student Name: Giacomo Cappelletto
Student ID: U91023753
Date: October 17, 2025

---

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 extensivelytested 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. (See Section 5.1.1 for code listing.)

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. (See Section 5.1.2 for code listing.)

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. (See Section 5.1.3 for code listing.)

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. (See Section 5.1.4 for code listing.)

2.2.5. Arithmetic submodules 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. (See Section 5.1.5 for code listing.)

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. (See Section 5.1.5 for code listing.)

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. (See Section 5.1.6 for code listing.)

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. For all testbenches, see Section 5.2.

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 Section 5.2.4, 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 Section 5.2.5, for the complete testbench.)

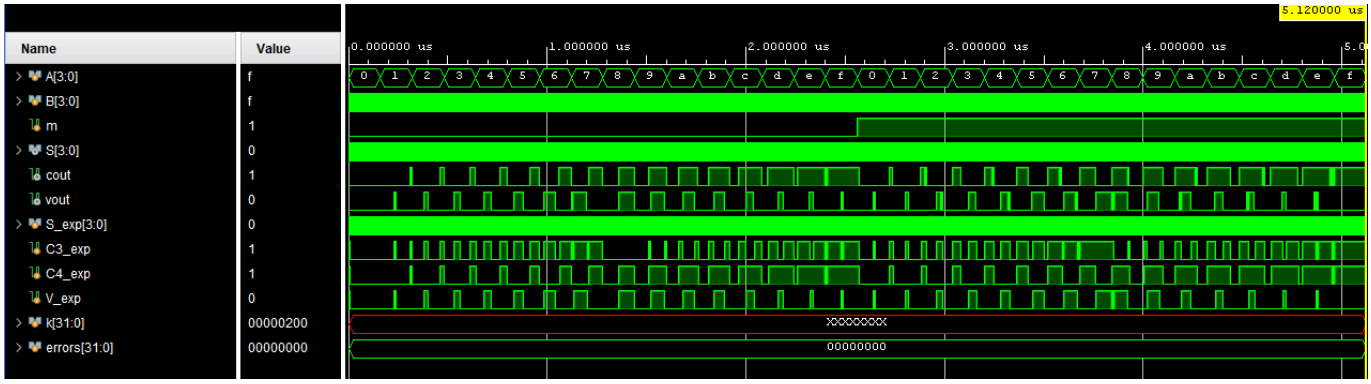
3. Observation

3.1. Simulation Results

See all waveforms in Section 5.3.

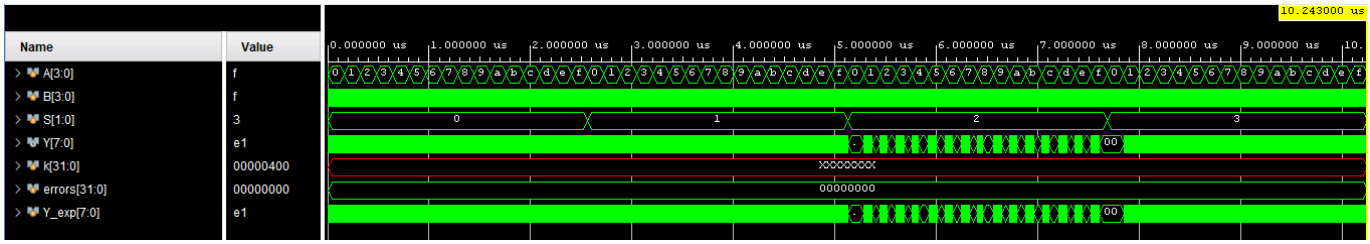
3.1.1. Adder-Subtractor Waveform

The adder-subtractor waveform shows the full 512 value sweep with a clear transition from addition to subtraction when `m` toggles. The ripple behavior is visible across the sum bits as carries propagate, and the overflow indicator `vout` asserts exactly when the carry into and out of the MSB differ, matching `C3_exp ^ C4_exp`. Throughout the sweep, the module outputs `S` and `cout` align with `S_exp` and `C4_exp`, and `errors` remains zero, showing passed results for both add and subtract.



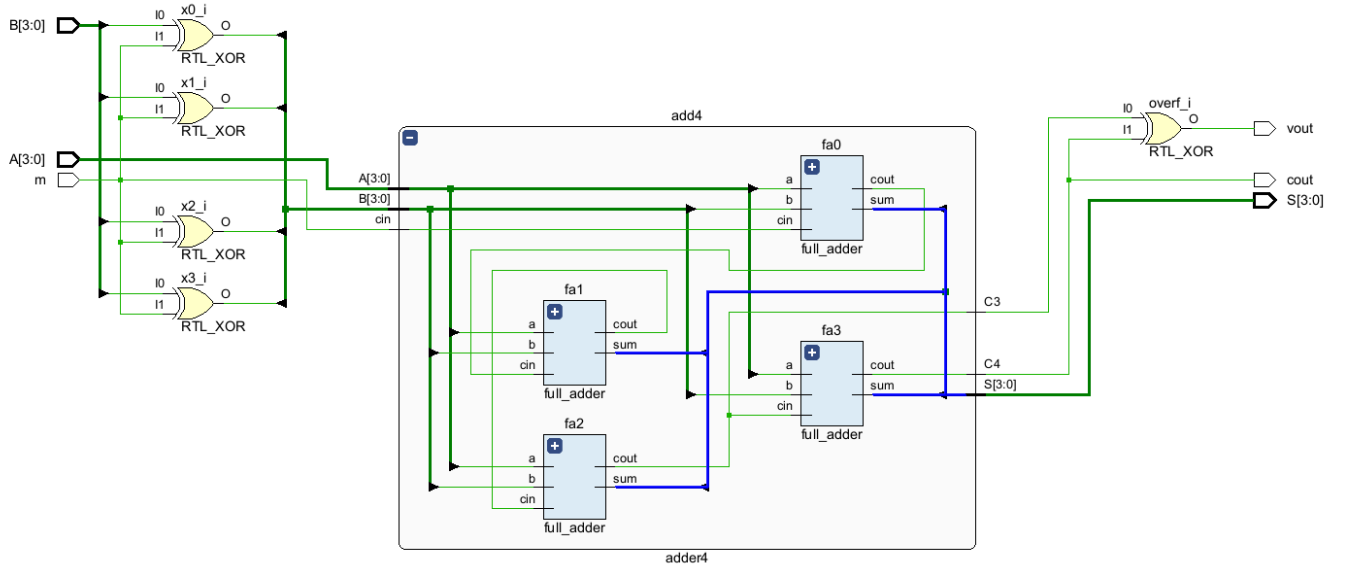
3.1.2. ALU Waveform

The ALU waveform breaks into clear bands by `S`: 00 concat `{A,B}`, 01 add, 10 left shift, 11 multiply. In the concat band, `Y` forms a stair-step pattern as `A/B` count because the bits are just packed together. In the add band, `Y` is the sum of `A+B` with added zeros to the left, so it ramps smoothly and never truncates (we have 8 bits). In the shift band, `Y = A << B` with zeros shifted in; when `B > 7` the output clamps to `00` as intended. In the multiply band, the 8-bit product tracks the reference (like `F*F = E1`) and you can see denser toggling from the larger range. We sample after a short settle, so no X/Z show up on `Y`. Across all 1024 cases, `Y` matches the reference and `errors` stays 0.



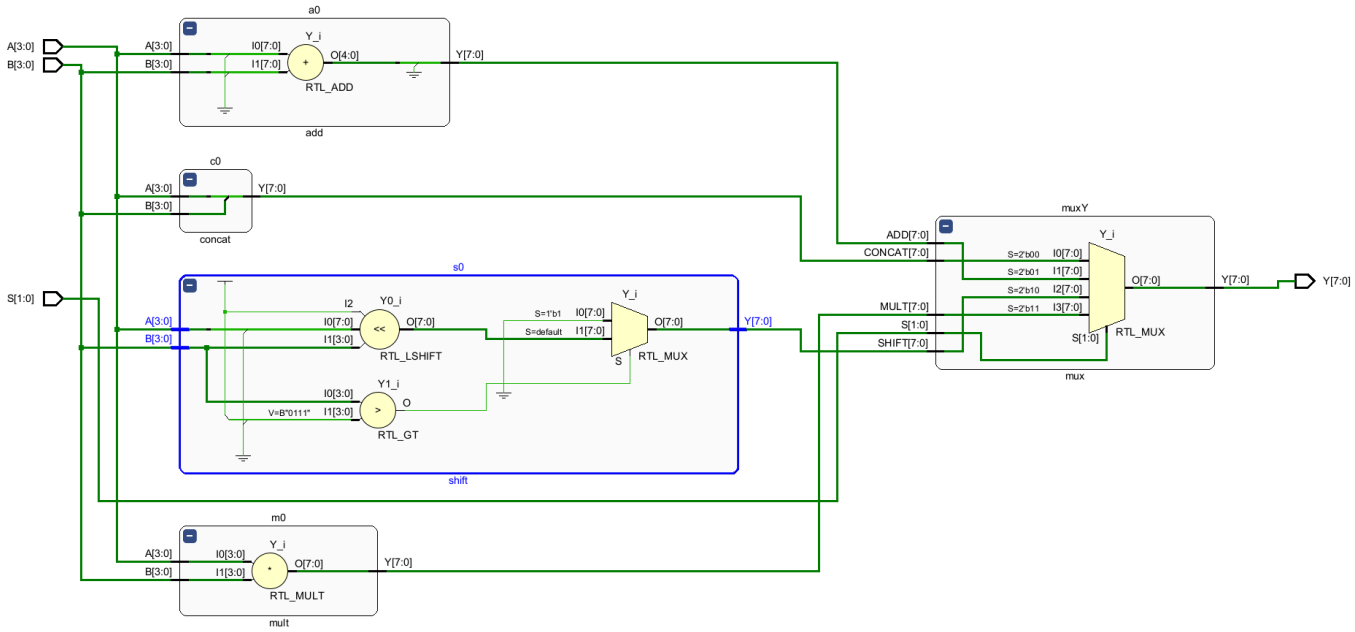
3.2. RTL Schematic

3.2.1. Adder-Subtractor RTL Schematic



In the RTL schematic of the addsub4 module, the subtraction arithmetic implementation is now visible, and it is possible then to confirm that its implementation is equivalent to the one proposed in the lab instructions. Observe that `m` is used both to flip the bits of the `B` input, and to feed the carry-in of the first full adder in order to complete the 2's complement arithmetic process. At the output of the 4-bit RCA we can see the exposed `C3` and `C4` signals, which are used to detect overflow and compute `vout`.

3.2.2. ALU RTL Schematic



In the RTL schematic of the alu module, it is possible to observe the combinational implementation of the 4 arithmetic submodules, which were coded in behavioural verilog and therefore the gate-level primitives were implemented by the synthesizer. The 4:1 8-bit multiplexer is also visible, and it is used to select the output of the desired arithmetic submodule based on the 2-bit control signal.

4. Conclusion

Summarize findings and reflect on the laboratory experience.

4.1. Summary of Results

This lab was succesful in completing the task to design both a 4-bit adder-subtractor and a 4-bit ALU, and to test them using testbenches. The simulation results confirmed the correct functionality of all modules and submodules, and the RTL schematics confirmed the correct implementation of the adder-subtractor and the ALU.

4.2. Challenges and Solutions

The biggest challenge found in this lab was the implementation of the adder-subtractor, due to the fact that the last two carry-outs of the full adders were not exposed in the adder module initially, which first lead me to computing `C3` by creating separate logic from the inputs for this. I later realised that this was inefficient from a point of view of gate count, and so I exposed the third carry-out of the full adders in the adder module and used it to compute `C3` and `C4` in the adder-subtractor module. In hindsight, this does slightly reduce the abstraction level of the 4-bit adder module, but I believe it was a worthwhile trade-off in order to reduce the number of gates used in the adder-subtractor module as well as it's complexity.

5. Appendix

5.1. Module Listings

5.1.1. Half Adder (`half-adder.v`)

```
`timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
// Company: Boston University
// Engineer: Giacomo Cappelletto
//
// Create Date: 10/03/2025 11:52:46 AM
// Design Name: Half Adder
// Module Name: half-adder
// Project Name: 4-bit Adder-Subtractor module
// Target Devices:
// Tool Versions: Vivado 2024.1
// Description: Half Adder module
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


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
```

5.1.2. Full Adder (`full-adder.v`)

```
`default_nettype none
`timescale 1ns/1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Boston University
// Engineer: Giacomo Cappelletto
//
// Create Date: 10/03/2025 11:52:46 AM
// Design Name: Full Adder
// Module Name: full-adder
// Project Name: 4-bit Adder-Subtractor module
// Target Devices: None
// Tool Versions: Vivado 2024.1
// Description: Full Adder module
// Dependencies: half_adder.v
//
```

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


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 (cout, c1, c2);

endmodule
```

5.1.3. 4-bit Adder (`adder4.v`)

```
`default_nettype none
`timescale 1ns/1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Boston University
// Engineer: Giacomo Cappelletto
//
// Create Date: 10/03/2025 11:52:46 AM
// Design Name: 4-bit Adder
// Module Name: adder4
// Project Name: 4-bit Adder-Subtractor module
// Target Devices: None
// Tool Versions: Vivado 2024.1
// Description: 4-bit Adder module
// Dependencies: full_adder.v, half_adder.v
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


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
```

5.1.4. 4-bit Adder-Subtractor (`addsub4.v`)

```
`timescale 1ns / 1ps
`default_nettype none

//////////////////////////////////////////////////////////////////////////////////
// Company: Boston University
// Engineer: Giacomo Cappelletto
//
// Create Date: 10/03/2025 11:52:46 AM
// Design Name: 4-bit Adder-Subtractor
// Module Name: addsub4
// Project Name: 4-bit Adder-Subtractor module
// Target Devices: None
// Tool Versions: Vivado 2024.1
// Description: 4-bit Adder-Subtractor module
//
// Dependencies:  adder4.v, half_adder.v, full_adder.v
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


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
```

5.1.5. ALU Submodules (`alu.v`)

```
`timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
// Company: Boston University
// Engineer: Giacomo Cappelletto
//
// Create Date: 10/03/2025 01:46:06 PM
// Design Name: ALU submodules and ALU module
// Module Name: alu
// Project Name: 4-bit ALU module
// Target Devices: None
// Tool Versions: Vivado 2024.1
// Description: ALU submodules and ALU module
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
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

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

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
```

5.1.6. Top-level ALU (`alu.v`)

```
`timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
// Company: Boston University
// Engineer: Giacomo Cappelletto
//
// Create Date: 10/03/2025 01:46:06 PM
// Design Name: ALU submodules and ALU module
// Module Name: alu
// Project Name: 4-bit ALU module
// Target Devices: None
// Tool Versions: Vivado 2024.1
// Description: ALU submodules and ALU module
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
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

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

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
```

5.2. Testbench Listings

5.2.1. Half Adder Module (`tb_half_adder.v`)

```
`timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
// Company: Boston University
// Engineer: Giacomo Cappelletto
//
// Create Date: 10/03/2025 12:16:59 PM
// Design Name: Half Adder testbench
// Module Name: tb_half_adder
// Project Name: 4-bit Adder-Subtractor module
// Target Devices: None
```

```
// Tool Versions: Vivado 2024.1
// Description: Testbench for the Half Adder module
// Dependencies: half_adder.v
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////



module tb_half_adder;
  reg  a, b;
  wire sum, carry;

  half_adder dut(.a(a), .b(b), .sum(sum), .carry(carry));

  integer i, errors = 0;
  reg exp_sum, exp_carry;

  initial begin
    for (i = 0; i < 4; i = i + 1) begin
      {a,b} = i[1:0];
      #1;
      exp_sum   = a ^ b;
      exp_carry = a & b;
      if (sum !== exp_sum || carry !== exp_carry) begin
        $error("HA MISMATCH a=%0b b=%0b : got sum=%0b carry=%0b exp sum=%0b carry=%0b",
               a,b,sum,carry,exp_sum,exp_carry);
        errors = errors + 1;
      end
      #9;
    end
    if (errors == 0) $display("HALF ADDER: ALL TESTS PASSED");
    else             $display(1, "HALF ADDER: %0d test(s) FAILED", errors);
  end
endmodule
```

5.2.2. Full Adder Module (`tb_full_adder.v`)

```
`timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
// Company: Boston University
// Engineer: Giacomo Cappelletto
//
// Create Date: 10/03/2025 12:16:59 PM
// Design Name: Full Adder testbench
// Module Name: tb_full_adder
// Project Name: 4-bit Adder-Subtractor module
// Target Devices: None
// Tool Versions: Vivado 2024.1
// Description: Testbench for the Full Adder module
//
// Dependencies: full_adder.v, half_adder.v
//
```

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

module tb_full_adder;
  reg  a, b, cin;
  wire sum, cout;

  full_adder dut(.a(a), .b(b), .cin(cin), .sum(sum), .cout(cout));

  integer i, errors = 0;
  reg exp_sum, exp_cout;

  initial begin
    for (i = 0; i < 8; i = i + 1) begin
      {a,b,cin} = i[2:0];
      #1;
      exp_sum  = a ^ b ^ cin;
      exp_cout = (a & b) | (a & cin) | (b & cin);
      if (sum !== exp_sum || cout !== exp_cout) begin
        $error("FA MISMATCH a=%0b b=%0b cin=%0b : got sum=%0b cout=%0b exp sum=%0b cout=%0b",
               a,b,cin,sum,cout,exp_sum,exp_cout);
        errors = errors + 1;
      end
      #9;
    end
    if (errors == 0) $display("FULL ADDER: ALL TESTS PASSED");
    else             $display(1, "FULL ADDER: %0d test(s) FAILED", errors);
  end
endmodule
```

5.2.3. 4-bit Adder Module (`tb_4adder.v`)

```
`timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
// Company: Boston University
// Engineer: Giacomo Cappelletto
//
// Create Date: 10/03/2025 12:39:31 PM
// Design Name: 4-bit Adder
// Module Name: tb_4adder
// Project Name: 4-bit Adder-Subtractor module
// Target Devices: None
// Tool Versions: Vivado 2024.1
// Description: Testbench for the 4-bit adder module
//
// Dependencies: adder4.v, full_adder.v, half_adder.v
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
```

```
module tb_4adder;
  reg  [3:0] A, B;
  reg        cin;
  wire [3:0] S;
  wire       C3, C4;   // exposed for overflow

  adder4 add4(.A(A), .B(B), .cin(cin), .S(S), .C3(C3), .C4(C4));

  reg [3:0] S_exp;
  reg c1, c2, c3_exp, C4_exp;

  task calc_expected;
    begin
      {c1,     S_exp[0]} = A[0] + B[0] + cin;
      {c2,     S_exp[1]} = A[1] + B[1] + c1;
      {c3_exp, S_exp[2]} = A[2] + B[2] + c2;     // C3
      {C4_exp, S_exp[3]} = A[3] + B[3] + c3_exp; // C4
    end
  endtask

  integer k, errors;
  initial begin
    errors = 0;

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

      if (S !== S_exp || C3 !== c3_exp || C4 !== C4_exp) begin
        errors = errors + 1;
        $error("ADDER4 FAIL: cin=%b A=%b B=%b  ->  S=%b C3=%b C4=%b  (exp S=%b C3=%b C4=%b)",
               cin, A, B, S, C3, C4, S_exp, c3_exp, C4_exp);
      end
      #9;
    end

    if (errors == 0)  $display("ADDER4: ALL TESTS PASSED");
    else $display("ADDER4: %0d test(s) FAILED", errors);

    $finish;
  end
endmodule
```

5.2.4. 4-bit Adder-Subtractor Testbench (`tb_addsub4.v`)

```
`timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
// Company: Boston University
// Engineer: Giacomo Cappelletto
//
// Create Date: 10/03/2025 01:27:34 PM
// Design Name: 4-bit Adder-Subtractor testbench
```

```
// Module Name: tb_addsub4
// Project Name: 4-bit Adder-Subtractor module
// Target Devices: None
// Tool Versions: Vivado 2024.1
// Description: Testbench for the 4-bit adder-subtractor module
//
// Dependencies: addsub4.v, adder4.v, half_adder.v, full_adder.v
//
// 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
```

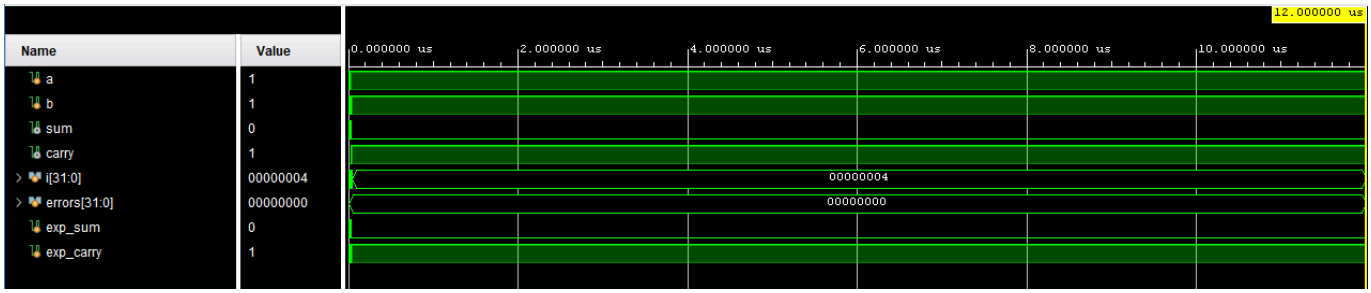
5.2.5. ALU Testbench (`tb_alu.v`)

```
`timescale 1ns/1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
// Company: Boston University
// Engineer: Giacomo Cappelletto
//
// Create Date: 10/03/2025 02:44:51 PM
// Design Name: 4-bit ALU testbench
// Module Name: tb_alu
// Project Name: 4-bit ALU module
// Target Devices:
// Tool Versions: Vivado 2024.1
// Description: Testbench for the 4-bit ALU module
// Dependencies: alu.v, alu_parts.v, adder4.v, full_adder.v, half_adder.v
// 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
```

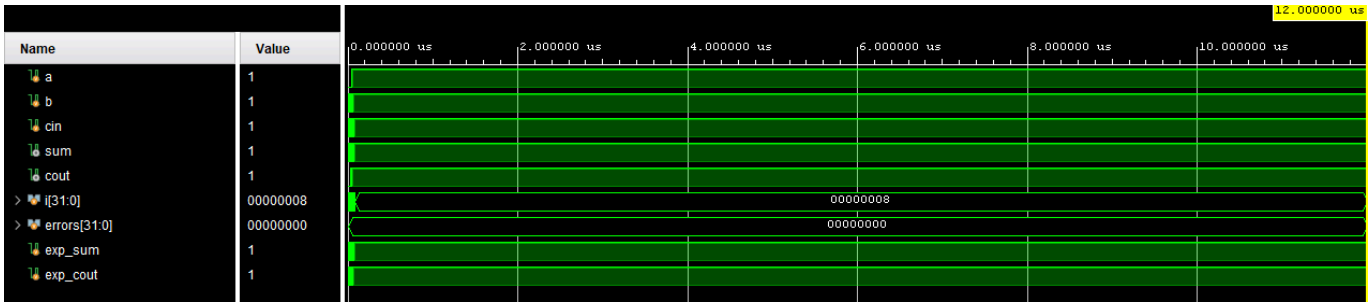
```
      {S, A, B} = k[9:0];

      #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
```

5.3. Simulation Waveforms

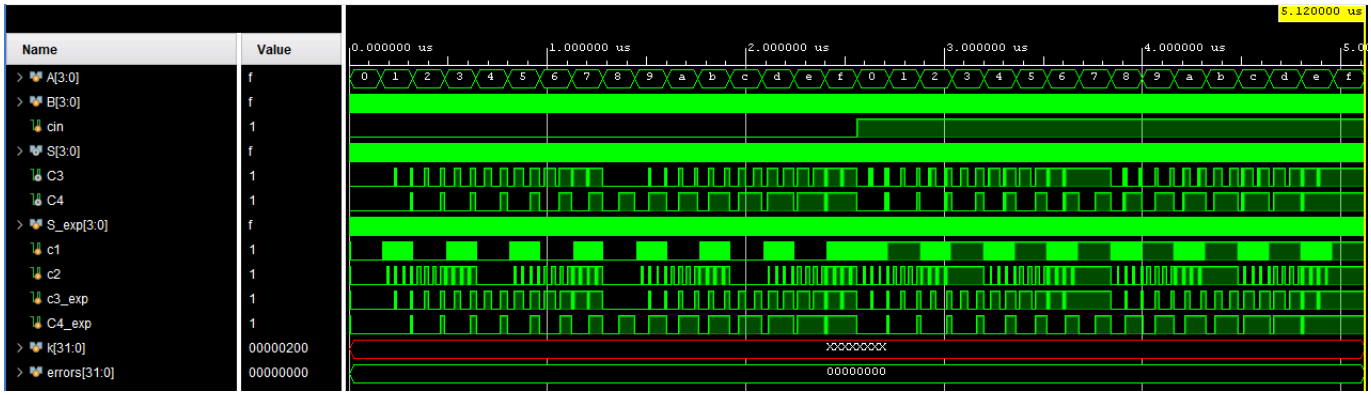
5.3.1. Half-Adder Waveform



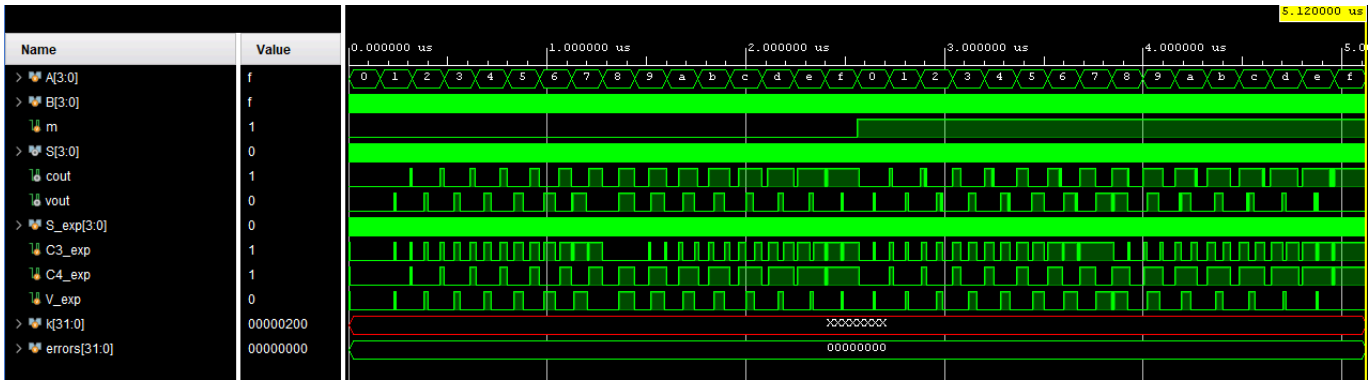
5.3.2. Full-Adder Waveform



5.3.3. 4-bit Adder Waveform



5.3.4. 4-bit Adder-Subtractor Waveform



5.3.5. ALU Waveform