Project 2

Tyler Williams
ECE 310
Dr. C

Overview

In this project, we designed and implemented a Serial Binary Coded Decimal (BCD) Arithmetic Logic Unit (ALU) capable of performing addition and subtraction operations on 4-digit BCD numbers. The ALU processes input data and control signals serially as structured "packets," converting them into parallel data internally for computation. Similarly, the computed results are serialized and output as a formatted packet. This implementation highlights the unique challenges of BCD arithmetic, including handling invalid BCD values and managing carries and borrows. By incorporating Serial-In Parallel-Out (SIPO) and Parallel-In Serial-Out (PISO) registers, the ALU seamlessly bridges the serial communication requirements of the system with parallel data processing. This report discusses the design, functionality, and timing considerations of the ALU, as well as its successful integration of sequence detection and packetized data flow.

Diagram and Discussion

My design is broken down into an overarching Project2 Module that uses the modules combBCDadd_4d and combBDCsub_4d to perform the actual addition and subtraction. These two modules use the fourth module, combBCDadd_digit, to do the computations.

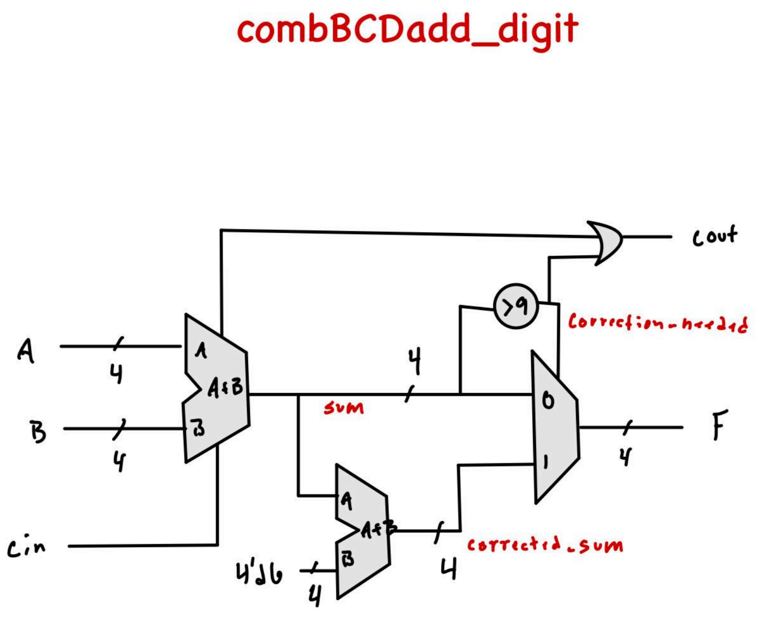


Figure 1: combBCDadd_digit

Figure 1 shows the pen-and-paper design of the digit adder that was implemented in lab 7. A, B, and Cin are added together to make Cout_intermediate and sum. If sum is greater than 9, then F is sum + 4’d6, and if not, sum is fed straight to F. Cout will be 1 if Cout_intermediate is 1 or if sum is greater than 9.

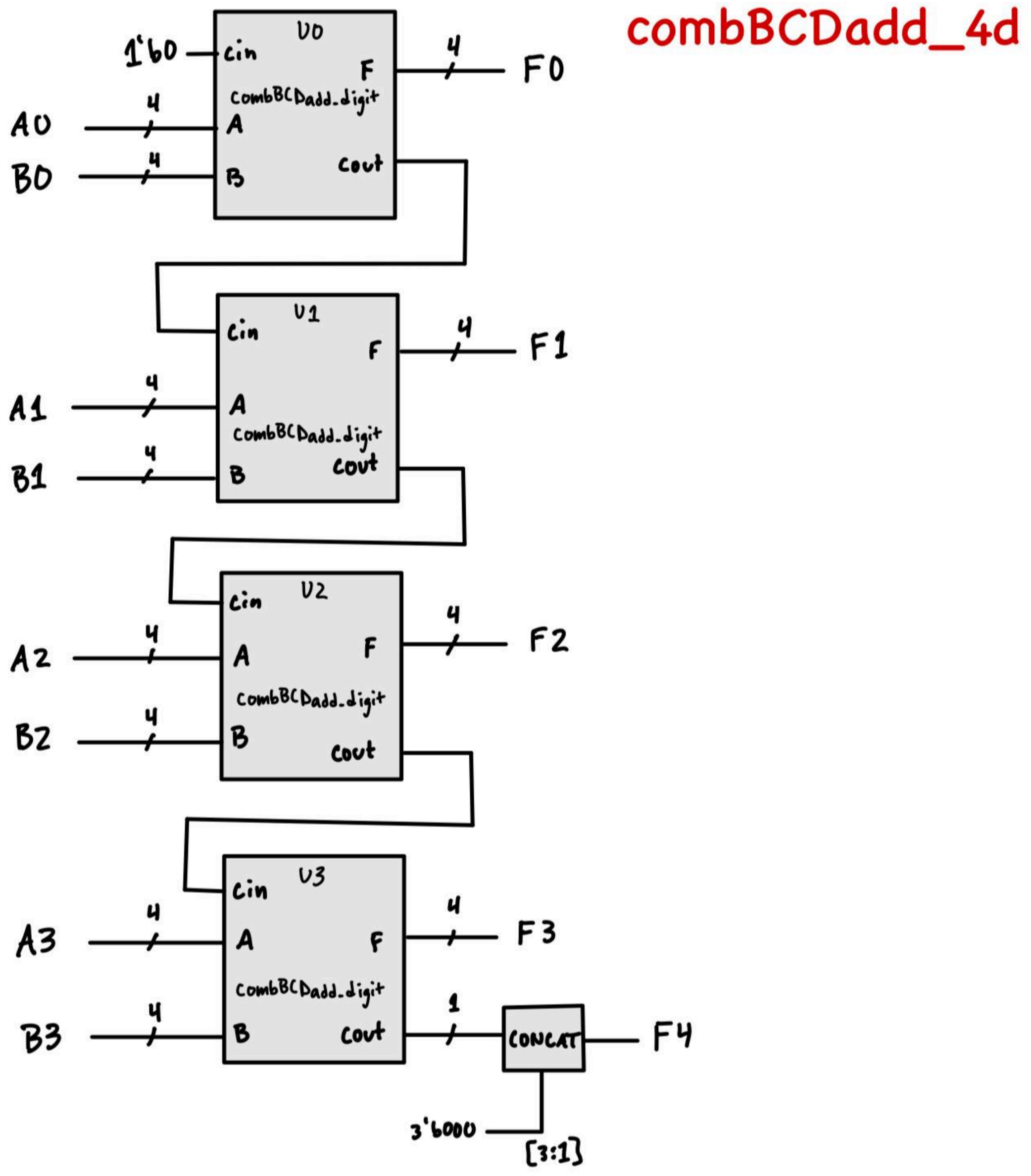


Figure 2: combBCDadd_4d

Figure 2 shows the pen-and-paper design of the 16 bit BCD adder. This is just an implementation of a ripple carry adder that utilizes the combBCDadd_digit module to add the actual digit addition. This module handles all of the carries of addition and returns the output in BCD values via F0, F1, F2, F3, and F4. A and B are the BCD representations of the two input numbers being added.

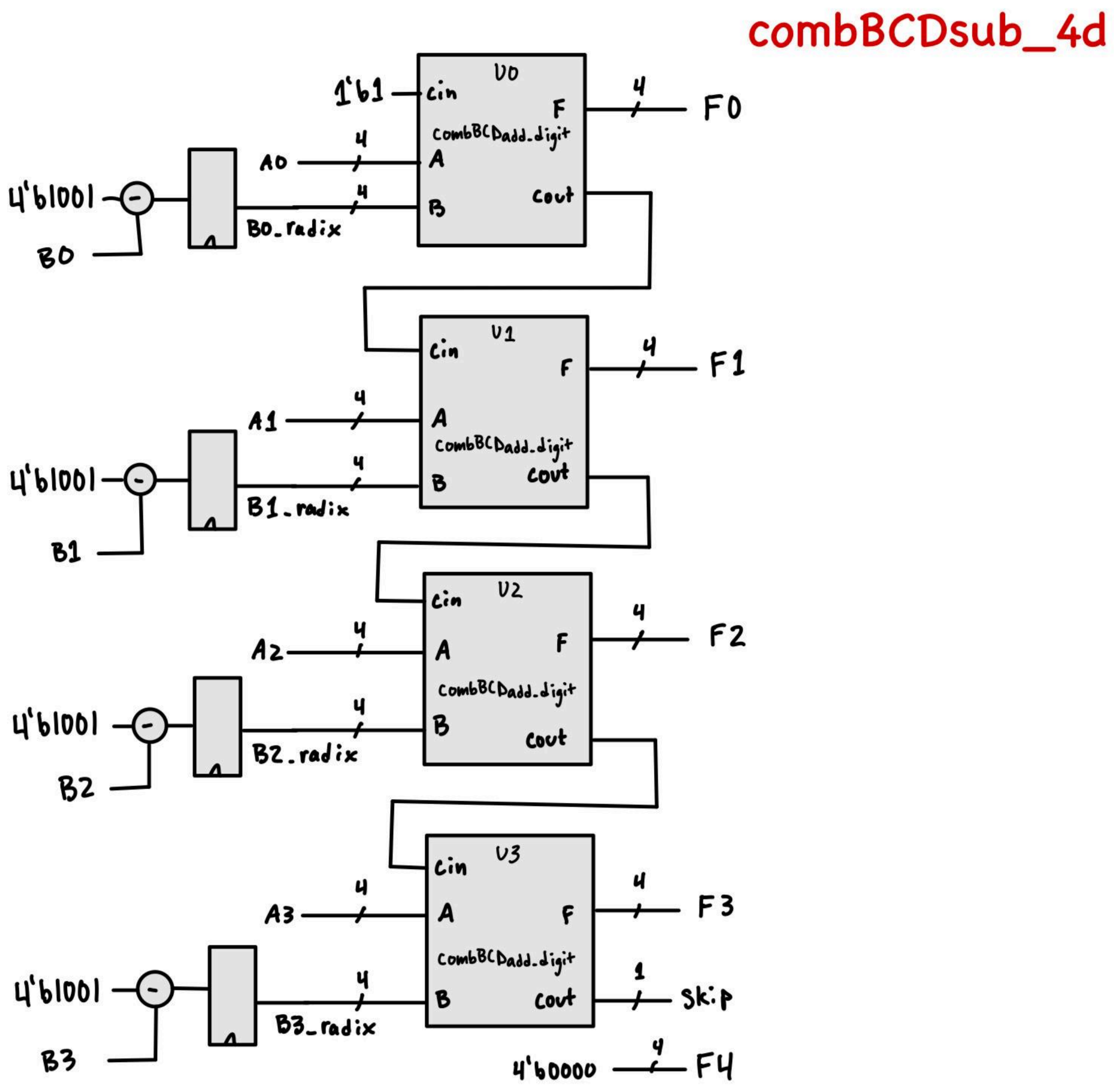


Figure 3: combBCDsub_4d

Figure 3 shows the pen-and-paper design of the 16 bit BCD subtractor. It is a similar ripple carry structure but registers B0_radix, B1_radix, B2_radix, and B3_radix to calculate the 10's complement of each digit. We can still use the combBCDadd_digit module because we are taking the complement of the B digits so it is really just performing A + (-B) which is the same as subtracting.

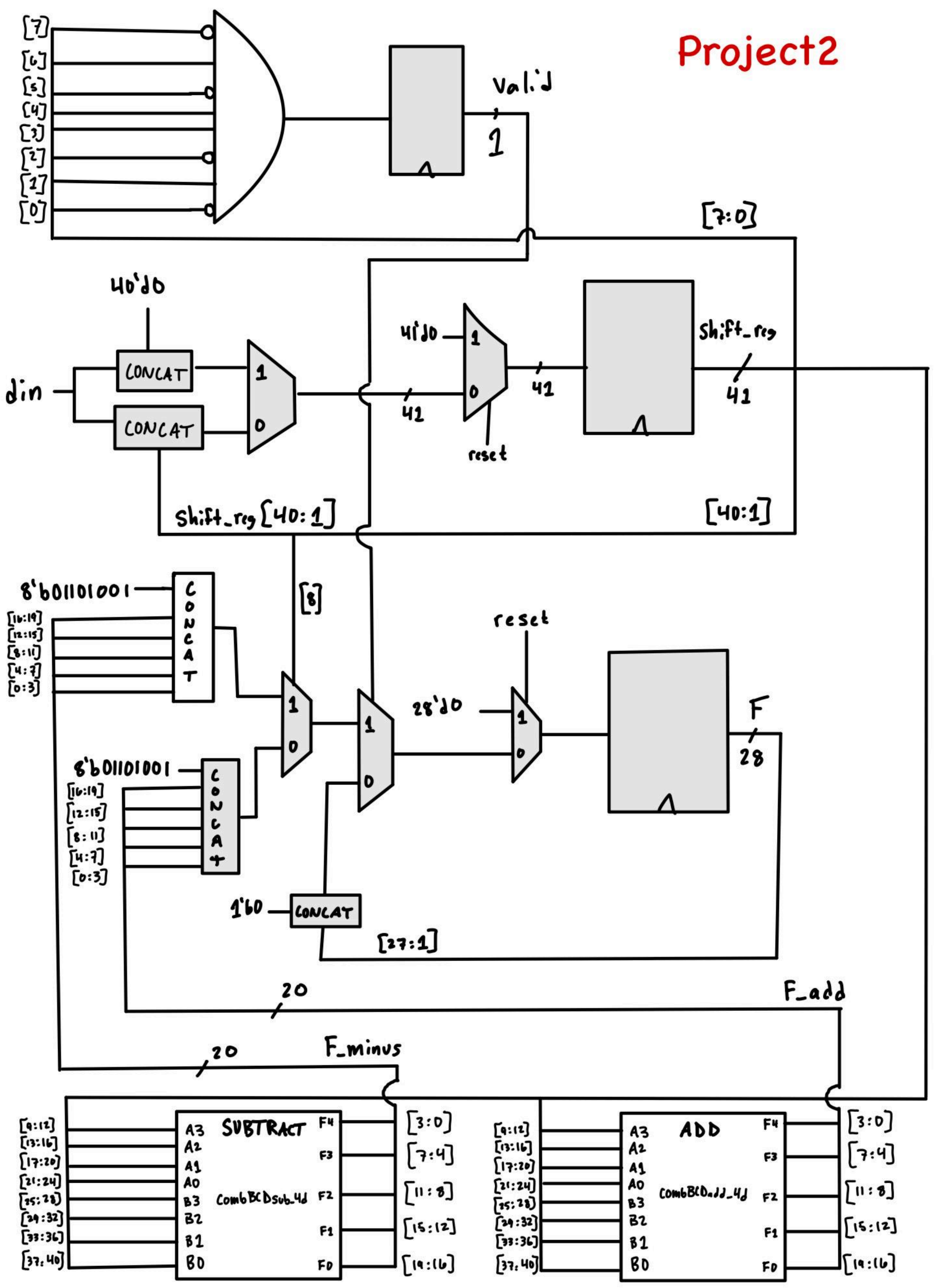


Figure 4: Project2.v

Figure 4 shows the pen-and-paper design of the high-level system and how everything comes together. It is looking for the input packet control pattern to set valid. Once valid is set the shifting in shift_reg will stop and the actual calculation will be performed to write the output of the calculation to F. Shift_reg is the register that stores the input packet until it is ready to be read. It then feeds the add and subtract modules, and the actual module's output that is used is based on the 9th bit of the shift_reg, which determines if it is addition or subtraction.

Synthesis
Figure 5 shows the synthesized schematic of my design. While this is a hard way to visualize my design since all 4 modules are shown, the design matches my expectations from my pen-and-paper designs. It is so large since we are working with registers of size 41 and 28, so a lot of the design is very repetitive to keep track of the serial bit stream.

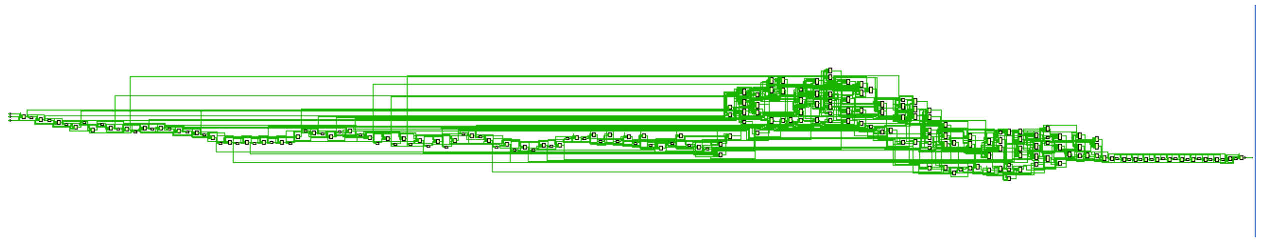
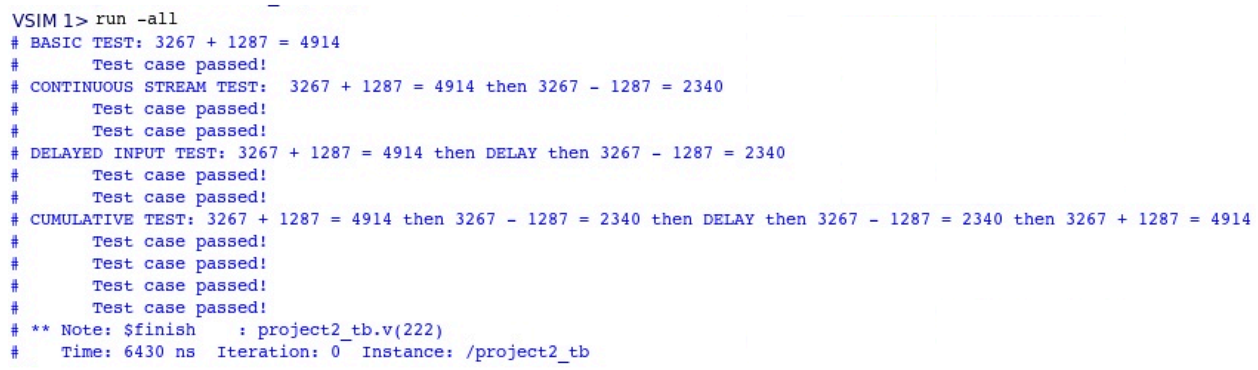


Figure 5: Design Schematic

Waveform Results and Discussion

Here is the output from the provided testbench:



```
VSIM 1> run -all
# BASIC TEST: 3267 + 1287 = 4914
#       Test case passed!
# CONTINUOUS STREAM TEST:  3267 + 1287 = 4914 then 3267 - 1287 = 2340
#       Test case passed!
#       Test case passed!
# DELAYED INPUT TEST: 3267 + 1287 = 4914 then DELAY then 3267 - 1287 = 2340
#       Test case passed!
#       Test case passed!
# CUMULATIVE TEST: 3267 + 1287 = 4914 then 3267 - 1287 = 2340 then DELAY then 3267 - 1287 = 2340 then 3267 + 1287 = 4914
#       Test case passed!
#       Test case passed!
#       Test case passed!
#       Test case passed!
# ** Note: $finish    : project2_tb.v(222)
#    Time: 6430 ns  Iteration: 0  Instance: /project2_tb
```

This shows an extensive testbench that tested several different functionalities. Basic addition and subtraction worked, as well as the result signal to bring the system back to a known state. The reset also worked without a delay, showing that we are able to do back to back computations.

The waveform was very long for this extensive test, so Figure 6 shows a small screenshot of what the waveform was telling me.

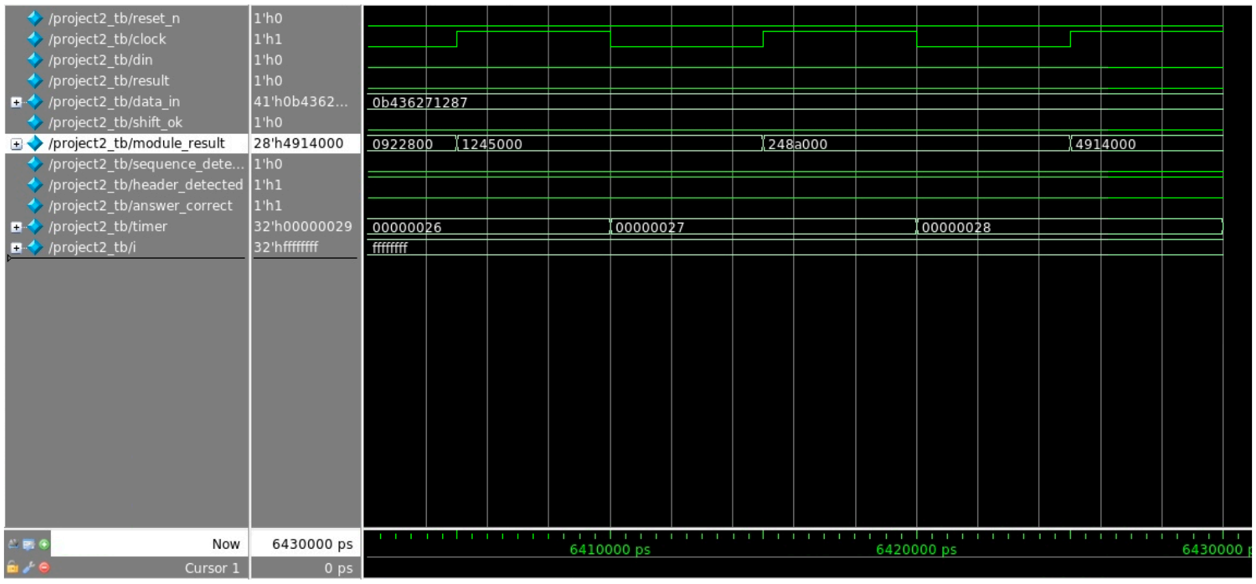
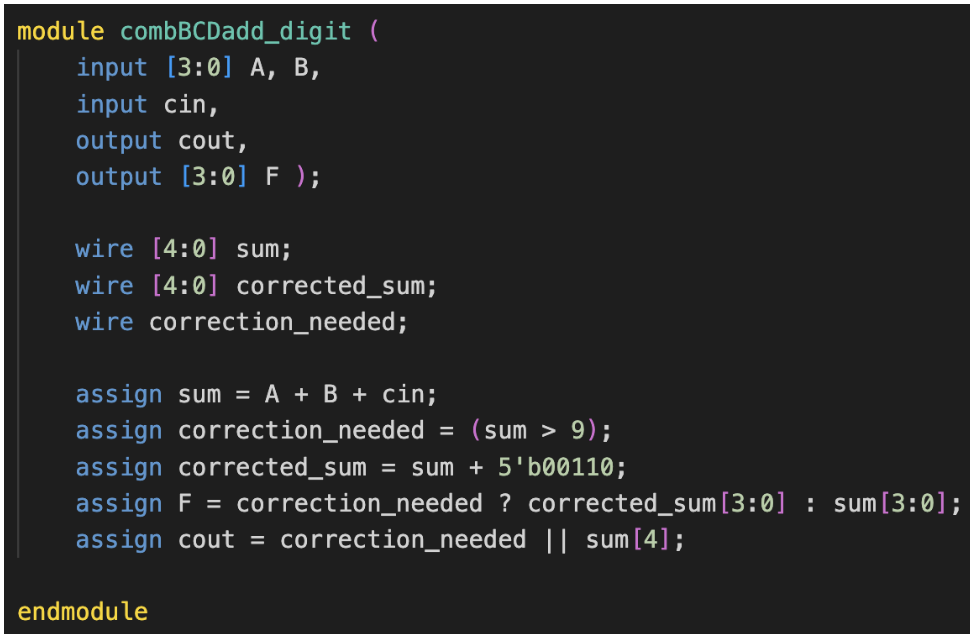

/project2_tb/reset_n 1'h0
/project2_tb/clock 1'h1
/project2_tb/din 1'h0
/project2_tb/result 1'h0
/project2_tb/data_in 41'h0b4362...
0b436271287
/project2_tb/shift_ok 1'h0
/project2_tb/module_result 28'h4914000
0922800
1245000
248a000
4914000
/project2_tb/sequence_dete... 1'h0
/project2_tb/header_detected 1'h1
/project2_tb/answer_correct 1'h1
/project2_tb/timer 32'h00000029
00000026
00000027
00000028
/project2_tb/i 32'hffffffff
ffffffff
Now 6430000 ps
Cursor 1 0 ps
6410000 ps
6420000 ps
6430000

Figure 6: Waveform Screenshot

This waveform shows the final calculation from the testbench. Data_in shows 3627 + 1287, because the bit just before 3627 is a 0, indicating addition. The final output in module_result shows the sum, 4914, padded with 3 zeros.

Code Discussion

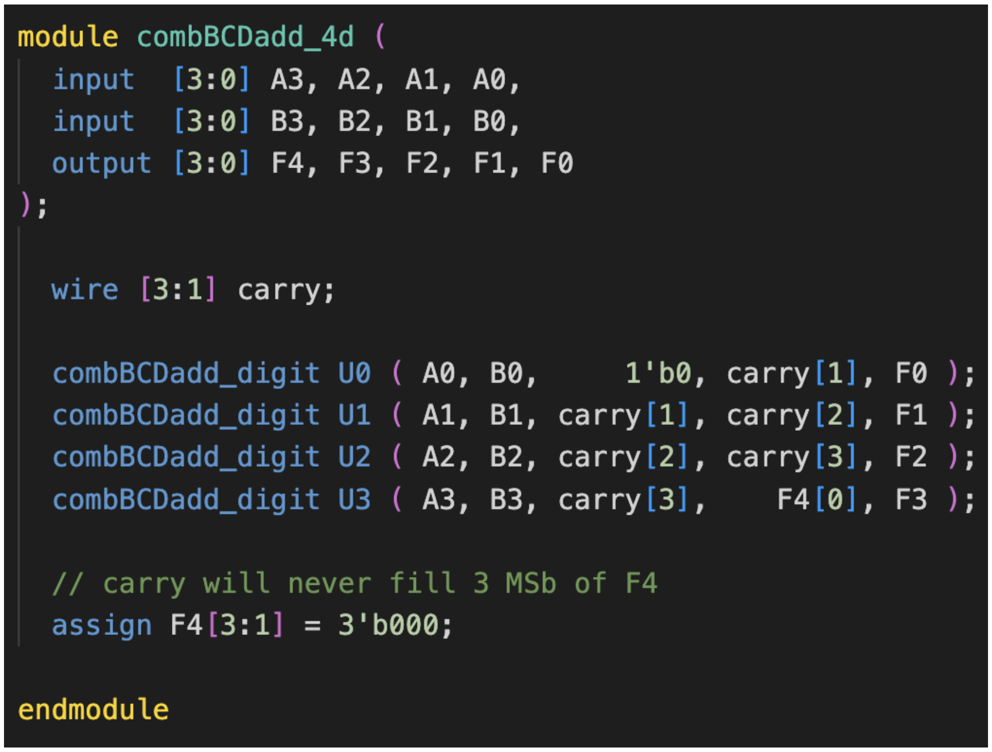
Figure 7 shows the code for the BCD digit addition. It takes two 4-bit BCD digits (A and B) and a carry-in bit (cin) as inputs and produces a 4-bit BCD sum (F) and a carry-out bit (cout). The module first computes the binary sum of the inputs and checks if the result exceeds 9, which would make it invalid in BCD format. If correction is needed, it adds 6 to the binary sum to adjust it to a valid BCD digit and sets the carry-out accordingly.



```verilog
module combBCDadd_digit (
    input [3:0] A, B,
    input cin,
    output cout,
    output [3:0] F );

    wire [4:0] sum;
    wire [4:0] corrected_sum;
    wire correction_needed;

    assign sum = A + B + cin;
    assign correction_needed = (sum > 9);
    assign corrected_sum = sum + 5'b00110;
    assign F = correction_needed ? corrected_sum[3:0] : sum[3:0];
    assign cout = correction_needed || sum[4];

endmodule
```

Figure 7

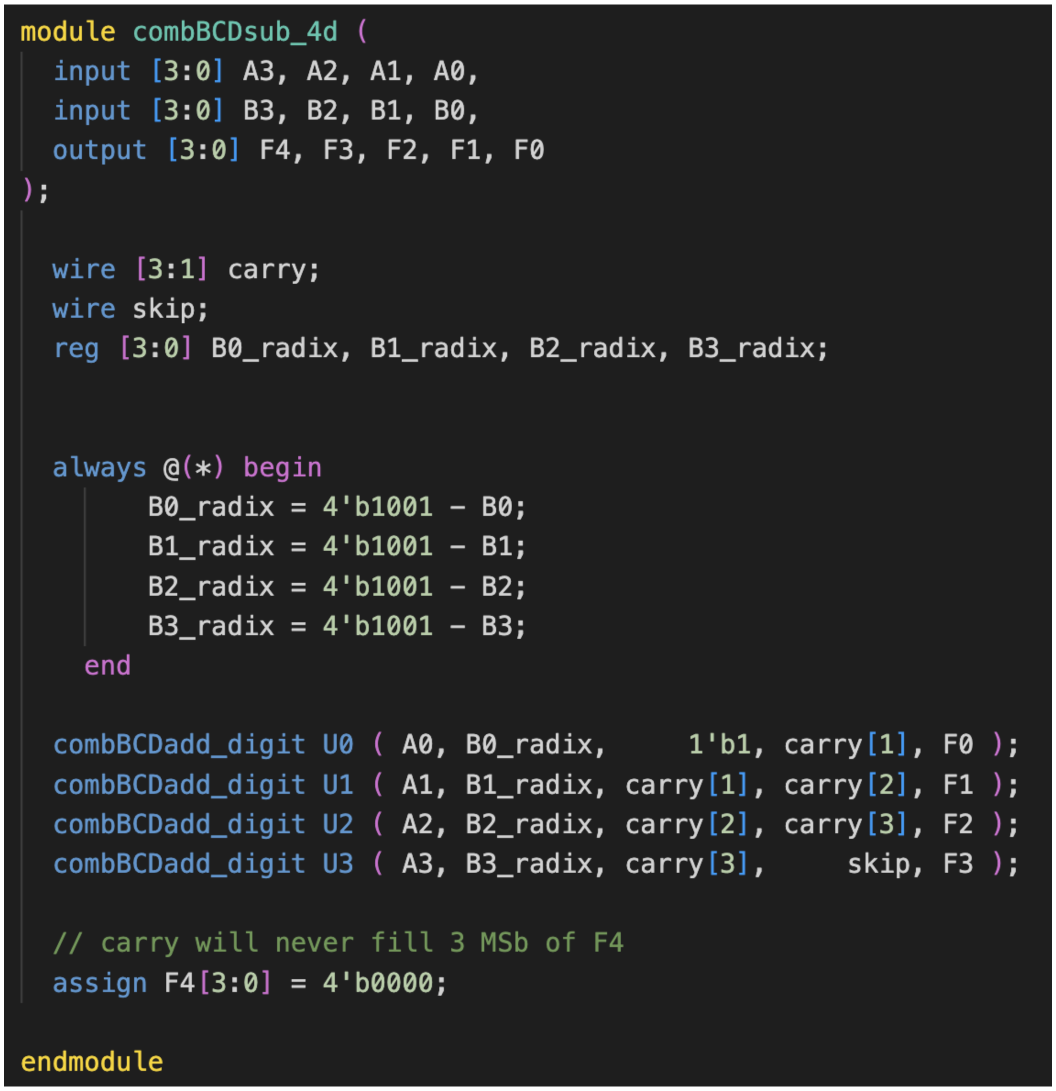
Figure 8 shows the code that performs the 4 digit BCD addition. It takes as inputs two 16-bit numbers, each represented as four separate 4-bit BCD digits (A3 to A0 and B3 to B0), and produces a 5-digit BCD result (F4 to F0). The addition is performed digit by digit using the combBCDadd_digit module, which ensures that each digit adheres to the BCD format. Carry bits are propagated between adjacent digit additions to account for overflow, starting with a carry-in of 0 for the least significant digit. The fifth digit (F4) only uses its least significant bit to capture any final carry-out, as the most significant bits are always zero.



```
module combBCDadd_4d (
  input  [3:0] A3, A2, A1, A0,
  input  [3:0] B3, B2, B1, B0,
  output [3:0] F4, F3, F2, F1, F0
);

  wire [3:1] carry;

  combBCDadd_digit U0 ( A0, B0,     1'b0, carry[1], F0 );
  combBCDadd_digit U1 ( A1, B1, carry[1], carry[2], F1 );
  combBCDadd_digit U2 ( A2, B2, carry[2], carry[3], F2 );
  combBCDadd_digit U3 ( A3, B3, carry[3],    F4[0], F3 );

  // carry will never fill 3 MSb of F4
  assign F4[3:1] = 3'b000;

endmodule
```

Figure 8

Figure 9 shows the code that performs the 4 digit BCD subtraction. It takes as inputs two 16-bit numbers, each represented as four separate 4-bit BCD digits (A3 to A0 and B3 to B0), and produces a 5-digit BCD result (F4 to F0). The subtraction is performed by taking the 10's complement of the B inputs, then digit by digit using the combBCDadd_digit module to perform (A + (-B)), which ensures that each digit adheres to the BCD format. Carry bits are propagated between adjacent digit additions to account for overflow, starting with a carry-in of 1 for the least significant digit. The fifth digit (F4) will always be all 0's since there won't ever be a carry-out in subtraction.

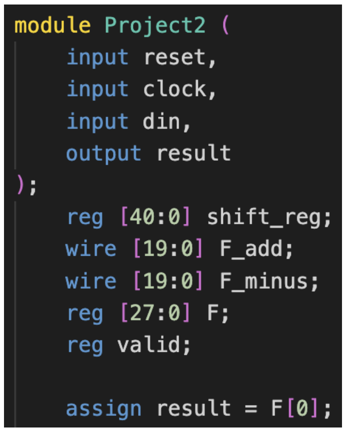


```
module combBCDsub_4d (
  input [3:0] A3, A2, A1, A0,
  input [3:0] B3, B2, B1, B0,
  output [3:0] F4, F3, F2, F1, F0
);

  wire [3:1] carry;
  wire skip;
  reg [3:0] B0_radix, B1_radix, B2_radix, B3_radix;


  always @(*) begin
        B0_radix = 4'b1001 - B0;
        B1_radix = 4'b1001 - B1;
        B2_radix = 4'b1001 - B2;
        B3_radix = 4'b1001 - B3;
    end

  combBCDadd_digit U0 ( A0, B0_radix,     1'b1, carry[1], F0 );
  combBCDadd_digit U1 ( A1, B1_radix, carry[1], carry[2], F1 );
  combBCDadd_digit U2 ( A2, B2_radix, carry[2], carry[3], F2 );
  combBCDadd_digit U3 ( A3, B3_radix, carry[3],     skip, F3 );

  // carry will never fill 3 MSb of F4
  assign F4[3:0] = 4'b0000;

endmodule
```

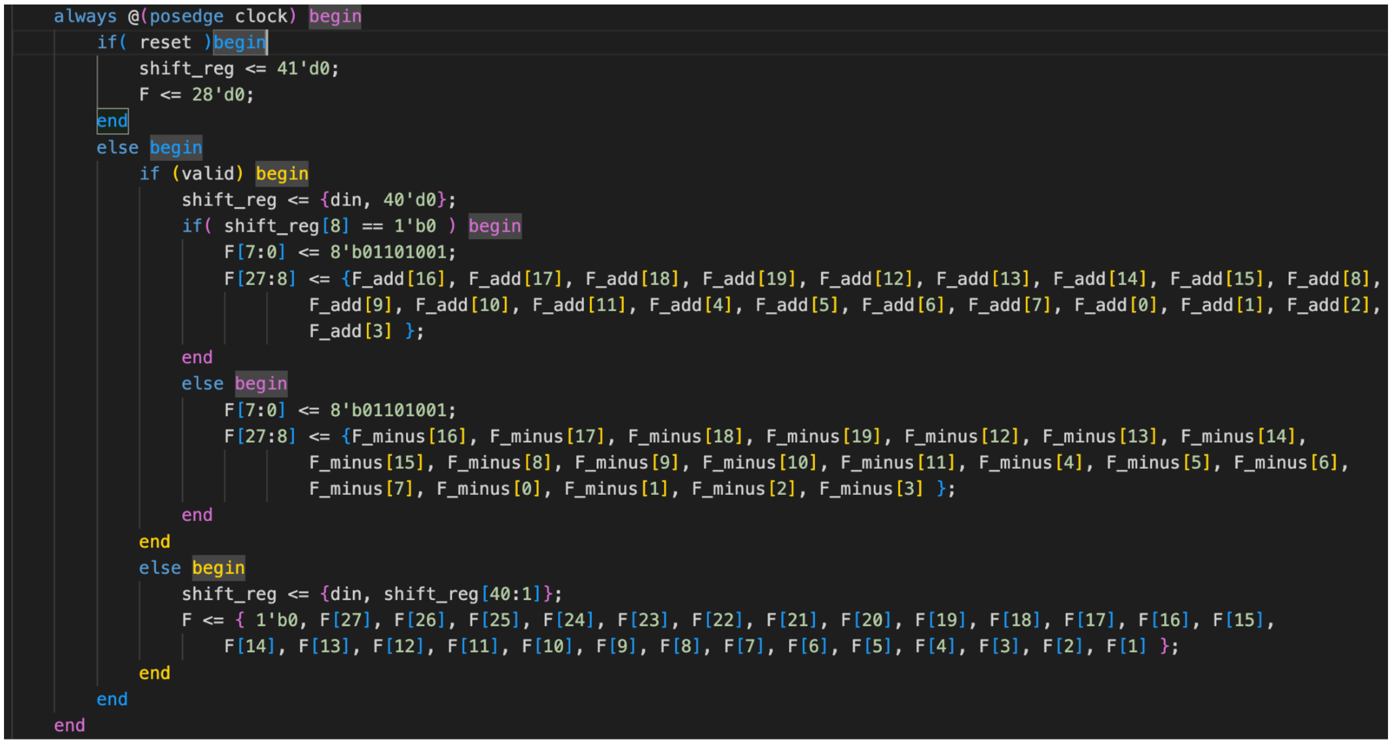
Figure 9

Figure 10 shows the Project2 module broken up into three screenshots. It processes a 41-bit shift register, performing multi-digit Binary-Coded Decimal (BCD) addition or subtraction depending on input conditions. It takes a serial input (din), a clock, and a reset signal as inputs, and outputs a single-bit result. Internally, the module maintains a 41-bit shift register that serially shifts in the input data. When the packet header is found, it marks the data as valid and performs either BCD addition or subtraction using the combBCDadd_4d and combBCDsub_4d modules, respectively. The result of these operations updates a 28-bit register (F), with the least significant bit of F assigned to result.

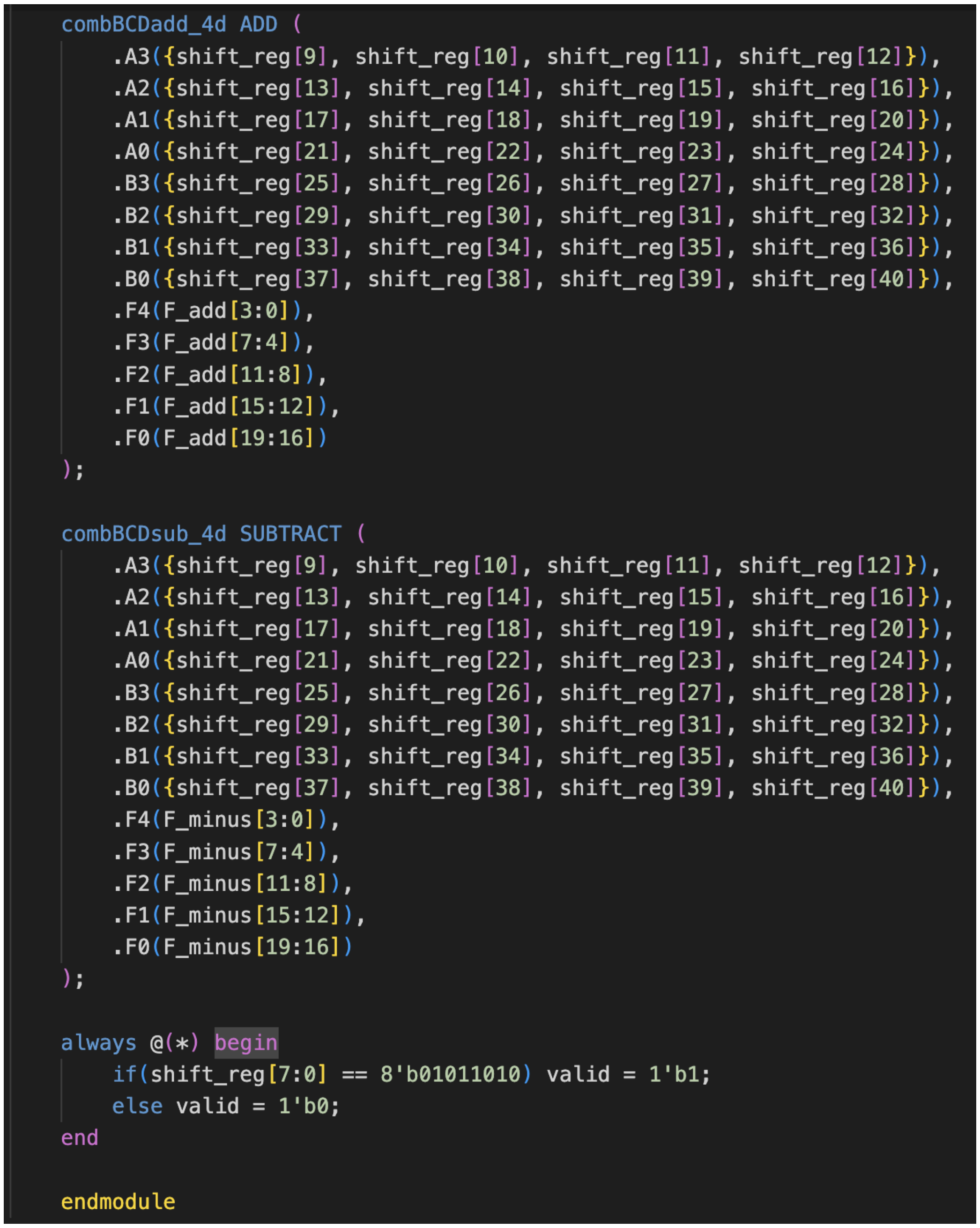
The module also integrates logic to determine whether to use the addition or subtraction result, based on the 9th bit of the shift register (shift_reg[8]). If the condition is invalid, the shift register continues to update with incoming serial data while the output register (F) shifts its bits. This design is suitable for applications requiring serial processing and arithmetic operations in BCD format, such as signal decoding or custom serial data processing systems.



```verilog
module Project2 (
    input reset,
    input clock,
    input din,
    output result
);
    reg [40:0] shift_reg;
    wire [19:0] F_add;
    wire [19:0] F_minus;
    reg [27:0] F;
    reg valid;

    assign result = F[0];
```



```verilog
    always @(posedge clock) begin
        if( reset )begin
            shift_reg <= 41'd0;
            F <= 28'd0;
        end
        else begin
            if (valid) begin
                shift_reg <= {din, 40'd0};
                if( shift_reg[8] == 1'b0 ) begin
                    F[7:0] <= 8'b01101001;
                    F[27:8] <= {F_add[16], F_add[17], F_add[18], F_add[19], F_add[12], F_add[13], F_add[14], F_add[15], F_add[8],
                            F_add[9], F_add[10], F_add[11], F_add[4], F_add[5], F_add[6], F_add[7], F_add[0], F_add[1], F_add[2],
                            F_add[3] };
                end
                else begin
                    F[7:0] <= 8'b01101001;
                    F[27:8] <= {F_minus[16], F_minus[17], F_minus[18], F_minus[19], F_minus[12], F_minus[13], F_minus[14],
                            F_minus[15], F_minus[8], F_minus[9], F_minus[10], F_minus[11], F_minus[4], F_minus[5], F_minus[6],
                            F_minus[7], F_minus[0], F_minus[1], F_minus[2], F_minus[3] };
                end
            end
            else begin
                shift_reg <= {din, shift_reg[40:1]};
                F <= { 1'b0, F[27], F[26], F[25], F[24], F[23], F[22], F[21], F[20], F[19], F[18], F[17], F[16], F[15],
                    F[14], F[13], F[12], F[11], F[10], F[9], F[8], F[7], F[6], F[5], F[4], F[3], F[2], F[1] };
            end
        end
    end
```



```
combBCDadd_4d ADD (
    .A3({shift_reg[9], shift_reg[10], shift_reg[11], shift_reg[12]}),
    .A2({shift_reg[13], shift_reg[14], shift_reg[15], shift_reg[16]}),
    .A1({shift_reg[17], shift_reg[18], shift_reg[19], shift_reg[20]}),
    .A0({shift_reg[21], shift_reg[22], shift_reg[23], shift_reg[24]}),
    .B3({shift_reg[25], shift_reg[26], shift_reg[27], shift_reg[28]}),
    .B2({shift_reg[29], shift_reg[30], shift_reg[31], shift_reg[32]}),
    .B1({shift_reg[33], shift_reg[34], shift_reg[35], shift_reg[36]}),
    .B0({shift_reg[37], shift_reg[38], shift_reg[39], shift_reg[40]}),
    .F4(F_add[3:0]),
    .F3(F_add[7:4]),
    .F2(F_add[11:8]),
    .F1(F_add[15:12]),
    .F0(F_add[19:16])
);

combBCDsub_4d SUBTRACT (
    .A3({shift_reg[9], shift_reg[10], shift_reg[11], shift_reg[12]}),
    .A2({shift_reg[13], shift_reg[14], shift_reg[15], shift_reg[16]}),
    .A1({shift_reg[17], shift_reg[18], shift_reg[19], shift_reg[20]}),
    .A0({shift_reg[21], shift_reg[22], shift_reg[23], shift_reg[24]}),
    .B3({shift_reg[25], shift_reg[26], shift_reg[27], shift_reg[28]}),
    .B2({shift_reg[29], shift_reg[30], shift_reg[31], shift_reg[32]}),
    .B1({shift_reg[33], shift_reg[34], shift_reg[35], shift_reg[36]}),
    .B0({shift_reg[37], shift_reg[38], shift_reg[39], shift_reg[40]}),
    .F4(F_minus[3:0]),
    .F3(F_minus[7:4]),
    .F2(F_minus[11:8]),
    .F1(F_minus[15:12]),
    .F0(F_minus[19:16])
);

always @(*) begin
    if(shift_reg[7:0] == 8'b01011010) valid = 1'b1;
    else valid = 1'b0;
end

endmodule
```

Figure 10

Conclusion

The successful development of the Serial BCD ALU demonstrates its ability to efficiently handle BCD arithmetic operations in a compact, serial communication framework. By leveraging a structured packetized approach and incorporating robust sequence detection, the ALU ensures accurate and reliable addition and subtraction of 4-digit BCD numbers. Timing analyses confirmed the system's capability to process consecutive operations without interruptions, reflecting its high efficiency and throughput. This project reinforced critical concepts in digital design, including arithmetic logic, sequence detection, and the integration of serial and parallel data systems, while providing valuable hands-on experience with Verilog and advanced hardware modeling techniques. The ALU’s design lays a solid foundation for further applications requiring serial communication and real-time arithmetic computation.