Design of Integrated Circuits: Phase 2 Report

Christian Moser

October 29, 2022

Chapter 1

Introduction

This phase starts to focus on the Verilog implementations for sub-modules of neural networks. A neural network-based classifier consists of adders, multipliers, and activation functions. In this phase, each student is required to implement the Verilog modules of adders, multipliers, the activation function (e.g., Sigmoid function and ReLU function), etc. You should implement all of these modules, simulate them, and validate their functionalities, separately. The default value representation is 16-bit fixed point value (8-bit integer and 8-bit fraction). According to your weights from neural network training (such as maximum values or precision), you can change the value representation to a wider-range value or floating-point value. The associated justification might be added into your report.

In this phase, all necessary modules in the neural network like adders and multipliers are required to be implemented. For example, 16-bit ripple-carry and the 16-bit bit-array multiplier are as shown in Figure 1 and 2 (also as shown in Chapter 11 of Textbook). Other more advanced adders and multipliers (e.g., carry-lookahead adder or Kogge-Stone adder) are welcome and will be graded with the consideration of extra credit. The simulation results should be included to validate the implemented adder/multiplier working well.

Chapter 2

Adder Implementation

We implement a pipe-lined architecture of the adder to reperesent a 32-bit floating point number, but can be expanded with any N-bit using parameterization. After extensive research, this was chosen over a Kogge-Stone or Brent-Kung adder in order to have more accurate data representation. This design has some issues, and looking into the roadmap of the hardware implementation, some corrections need to be made to have a completely working adder. So far, the implementation will run through 3 clock cycles and compile correctly in DUT testbench, but the final output is incorrect.

2.1 Simulation

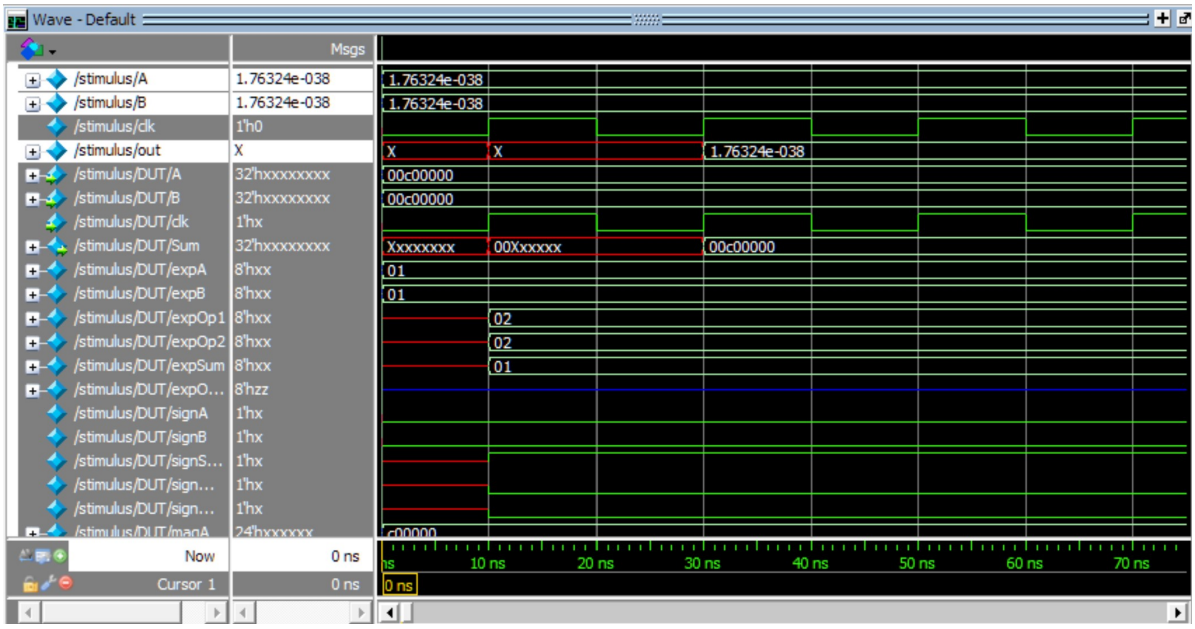

Figure 2.1: Waveform of Pipelined Adder Architecture

As seen in Figure 2.1, we can see that the architecture is very quick, but it appears to be only moving a copy of an operand into the summation. This will be discussed further during more thorough implementation. For now, we have the following code base to work with:

```
/*
Author: Christian Moser
Date: 10-28-2022
Description: Full-Precision (N-bit) floating point adder implementation

example of 32-bit single precision float:
1   11111111   11111111111111111111111
S   E          M
S = 0 | 1 implies positive (0) or negative (1)
E = 1111 1111 equivalent to whole number component
M = 1111 1111 1111 1111 1111 111 equivalent to decimal component
*/

module adder
    #(parameter N=32)
    (input  wire[N-1:0] A,
    input  wire[N-1:0] B,
    input clk,
    output  wire[N-1:0] Sum
```

```
    );
    //def wires
    wire   [(N/4)-1:0]    expA, expB, expOp1, expOp2, expSum, expOp1Hold;
    wire   signA, signB, signSum, signOp1, signOp2;
    wire   [N-(N/4)-1:0]    magA, magB, magOp1, magOp2, magSum;
    wire   [N-(N/4):0]    magHold;
    wire   rst;
    wire   done1, done2;
    reg    done3;
    reg    [N-1:0] sum;

    //initial assignments
    assign signA = A[N-1];
    assign signB = B[N-1];
    assign expA = A[N-2:(N-1)-(N/4)];
    assign expB = B[N-2:(N-1)-(N/4)];
    assign magA[N-(N/4)-1] = 1'b1;
    assign magB[N-(N/4)-1] = 1'b1;
    assign magA[N-(N/4)-2:0] = A[N-(N/4)-1:0];
    assign magB[N-(N/4)-2:0] = B[N-(N/4)-1:0];

    shiftreg #(.N(32)) compare(expA, expB, expOp1, expOp2, clk, signA, signB, signSum,
        signOp1, signOp2, magA, magB, magOp1, magOp2, done1);

    FA #(.N(32)) additive(magA, magB, expOp1Hold, clk, rst, magHold, done2, expSum);

    normalizer #(.N(32)) form(magHold, signA, signB, signSum, signOp1, signOp2, clk, expA,
        expOp1, expSum, magOp1, magOp2, magSum);



    assign Sum = {signA, expSum, magSum[(N-(N/4)-2):0]};

endmodule

module shiftreg
#(parameter N=32)
(   input  [(N/4)-1:0] expA,
    input  [(N/4)-1:0] expB,
    output reg [(N/4)-1:0] expOp1,
    output reg [(N/4)-1:0] expOp2,
    input  clk,
    input  signA,
    input  signB,
    output reg signSum,
    output reg signOp1,
    output reg signOp2,
    input  [N-(N/4)-1:0] magA,
    input  [N-(N/4)-1:0] magB,
    output reg [N-(N/4)-1:0] magOp1,
```

```
    output reg [N-(N/4)-1:0] magOp2,
    output reg done1 = 0
);

    reg[(N-4)-1:0] diff;

    always @(posedge clk) begin
        signOp1 = signA;
        signOp2 = signB;


        if(expA == expB) begin
            expOp1 = expA + 8'b1;
            expOp2 = expB + 8'b1;

            magOp1 = magA;
            magOp2 = magB;

            signSum = 1'b1;
        end

        else if(expA > expB) begin
            diff = expA - expB;
            expOp1 = expA + 8'b1;
            expOp2 = expB + 8'b1;

            magOp1 = magA;
            magOp2 = magB >> diff;

            signSum = 1'b1;
        end

        else if(expB > expA) begin
            diff = expB - expA;
            expOp1 = expB + 8'b1;
            expOp2 = expA + 8'b1;

            magOp1 = magB;
            magOp2 = magA >> diff;

            signSum = 1'b0;
        end

        else begin
            diff = expB - expA;
            expOp1 = expB + 8'b1;
            expOp2 = expB + 8'b1;
            magOp1 = magB;
            magOp2 = magA >> diff;
        end
```

```
        done1 <= 1;
    end


endmodule

module normalizer#(parameter N=32)
(
    input   [N-(N/4):0]    magHold,
    input   signA,
    input   signB,
    input  signSum,
    input  signOp1,
    input  signOp2,
    input  clk,
    input  [(N/4)-1:0]    expA,
    input  [(N/4)-1:0]    expOp1,
    output reg [(N/4)-1:0]   expSum,
    output reg [N-(N/4)-1:0] magOp1,
    output reg [N-(N/4)-1:0] magOp2,
    output reg [N-(N/4)-1:0] magSum
);

    always @(posedge clk) begin
        magSum = magHold[N-(N/4):1];
        expSum = expA;
        repeat(N-(N/4)) begin

            if(magSum[N-(N/4)-1] == 1'b0) begin
                magSum = magSum << 1'b1;
                expSum = expSum - 8'b1;
            end
        end
    end
endmodule

module FA#(parameter N=32)
(
    input  [N-(N/4)-1:0] magA,
    input  [N-(N/4)-1:0] magB,
    input  [(N/4)-1:0]    tmpExp,
    input  clk,
    input  rst,
    output reg [N-(N/4):0] magSum,
    output reg done2 = 0,
    output reg [(N/4)-1:0] tmpExpAdd
);

always @(posedge clk) begin
```

```
    magSum <= magA + magB;
    tmpExpAdd <= tmpExp;

    if(magSum == (magA + magB)) begin
        done2 <= 1;
    end
end
endmodule
```

Testbench:

```
module stimulus();

    reg     [31:0] A,  B;
    reg     clk;
    wire    [31:0] out;

    adder #(.N(32)) DUT(A, B, clk, out);
    initial begin
        A = 32'b00000000110000000000000000000000;
        B = 32'b00000000110000000000000000000000;
        clk = 1'b0;
    end

    always #10 clk = !clk;

endmodule
```

Chapter 3

Multiplier Implementation

Our multiplier also uses 32-bit floating point representation, but is not parameterized as the adder was due to time constraints. The multiplier has some issues that need to be fixed before hardware implementation.

3.1 Simulation

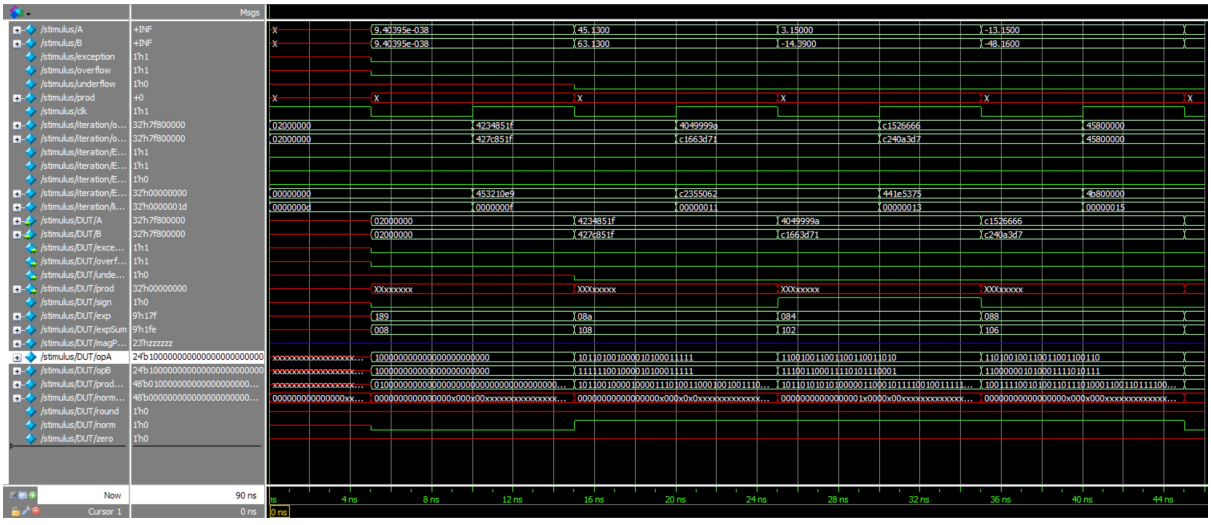

Figure 3.1: Multiplier Waveforms with 32-bit floating point data

We also used some CLI to test certain values. This can be seen in the attached testbench.

```
/*
Author: Christian Moser
Date: 10-28-2022
```

```
Project Description:

Create floating-point multiplier used for MLP neural network implementation.
*/

module multiplier
(   input  [31:0] A,
    input  [31:0] B,
    output exception,
    output overflow,
    output underflow,
    output [31:0] prod
);

//def wires
wire   sign;
wire   [8:0]  exp, expSum;
wire   [22:0] magProd;
wire   [23:0] opA, opB;
wire   [47:0] prodTmp, normalizedProd;

assign sign       = A[31] ^ B[31]; // get sign
assign exception  = (&A[30:23]) | (&B[30:23]);

assign opA        = (|A[30:23]) ? {1'b1, A[22:0]} : {1'b0, A[22:0]};
assign opB        = (|A[30:23]) ? {1'b1, B[22:0]} : {1'b0, B[22:0]};

assign prodTmp      = opA * opB;
assign round        = |normalizedProd[22:0];
assign norm         = prodTmp[47] ? 1'b1 : 1'b0;
assign normalizedProd = norm ? prod : prod << 1;
assign zero         = exception ? 1'b0 : (magProd == 23'b0) ? 1'b1 : 1'b0;
assign expSum       = A[30:23] + B[30:23];
assign exp          = expSum - 8'd127 + norm;

assign overflow   =  ((exp[8] & !exp[7]) & !zero);
assign underflow  =  ((exp[8] & exp[7]) & !zero) ? 1'b1 : 1'b0;

assign prod       =  exception ? 32'd0 : zero ? {sign, 31'd0} : overflow ? {sign, 8'hFF,
    23'd0} : underflow ? {sign, 31'd0} : {sign, exp[7:0], magProd};
endmodule
```

Testbench:

```
module stimulus();

    reg[31:0] A, B;
    wire exception, overflow, underflow;
    wire [31:0] prod;
```

```
    reg clk = 1'b1;

    multiplier DUT(A, B, exception, overflow, underflow, prod);

    always clk = #5 ~clk;
    initial begin
        iteration (32'h0200_0000,32'h0200_0000,1'b0,1'b0,1'b0,32'h0000_0000,`__LINE__);

        iteration (32'h4234_851F,32'h427C_851F,1'b0,1'b0,1'b0,32'h4532_10E9,`__LINE__); //
            45.13 * 63.13 = 2849.0569;

        iteration (32'h4049_999A,32'hC166_3D71,1'b0,1'b0,1'b0,32'hC235_5062,`__LINE__);
            //3.15 * -14.39 = -45.3285

        iteration (32'hC152_6666,32'hC240_A3D7,1'b0,1'b0,1'b0,32'h441E_5375,`__LINE__);
            //-13.15 * -48.16 = 633.304

        iteration (32'h4580_0000,32'h4580_0000,1'b0,1'b0,1'b0,32'h4B80_0000,`__LINE__);
            //4096 * 4096 = 16777216

        iteration (32'h3ACA_62C1,32'h3ACA_62C1,1'b0,1'b0,1'b0,32'h361F_FFE7,`__LINE__);
            //0.00154408081 * 0.00154408081 = 0.00000238418

        iteration (32'h0000_0000,32'h0000_0000,1'b0,1'b0,1'b0,32'h0000_0000,`__LINE__); //
            0 * 0 = 0;

        iteration (32'hC152_6666,32'h0000_0000,1'b0,1'b0,1'b0,32'h441E_5375,`__LINE__);
            //-13.15 * 0 = 0;

        iteration (32'h7F80_0000,32'h7F80_0000,1'b1,1'b1,1'b0,32'h0000_0000,`__LINE__);

        $stop;
    end

    task iteration(
input [31:0] op_a,op_b,
input Expected_Exception,Expected_Overflow,Expected_Underflow,
input [31:0] Expected_result,
input integer linenum
);
begin
@(negedge clk)
begin
   A = op_a;
   B = op_b;
end

@(posedge clk)
begin
if ((Expected_result == prod) && (Expected_Exception == exception) && (Expected_Overflow
```

```
    == overflow) && (Expected_Underflow == underflow))
   $display ("Success : %d",linenum);

else
   $display ("Failed : Expected_result = %h, Result = %h, \n Expected_Exception = %d,
       Exception = %d,\n Expected_Overflow = %d, Overflow = %d, \n Expected_Underflow =
       %d, Underflow = %d - %d \n
       ",Expected_result,prod,Expected_Exception,exception,Expected_Overflow,overflow,Expected_Underflow,underf
end
end
endtask

endmodule
```

Chapter 4

Activation Function Implementation

We will implement the tanh(x) (hyperbolic tangent) function as our activation function such to match the architecture of the neural network created in Phase I. This is still being developed but will be closely monitored after completing the adder and multiplier correctly.

Chapter 5

References

- https://github.com/rez39/Floatingpointadder
- https://github.com/debtanu09/fmultiplier
- https://github.com/sudhamshu091/32-Verilog-Mini-Projects
- T. K. R. Arvind, M. Brand, C. Heidorn, S. Boppu, F. Hannig and J. Teich, "Hardware Implementation of Hyperbolic Tangent Activation Function for Floating Point Formats," 2020 24th International Symposium on VLSI Design and Test (VDAT), 2020, pp. 1-6, doi: 10.1109/VDAT50263.2020.9190305.