Student: Johan Slatin
Student ID: 010933728
Prof: Dr. Chang Choo
Class: ee278
School: San Jose State University
Project: Dot product, with pipelined adder and multiplier.
Introduction:

In the following project we are supposed to create a dot product, with a pipelined adder and multiplier. However, it did not say that the whole project has to be pipelined, meaning that one could have a pipelined adder but not store the last value but just continuously add the inputs. In the following project I planned to have a design as in figure 1, where there is a fully pipelined dot product.

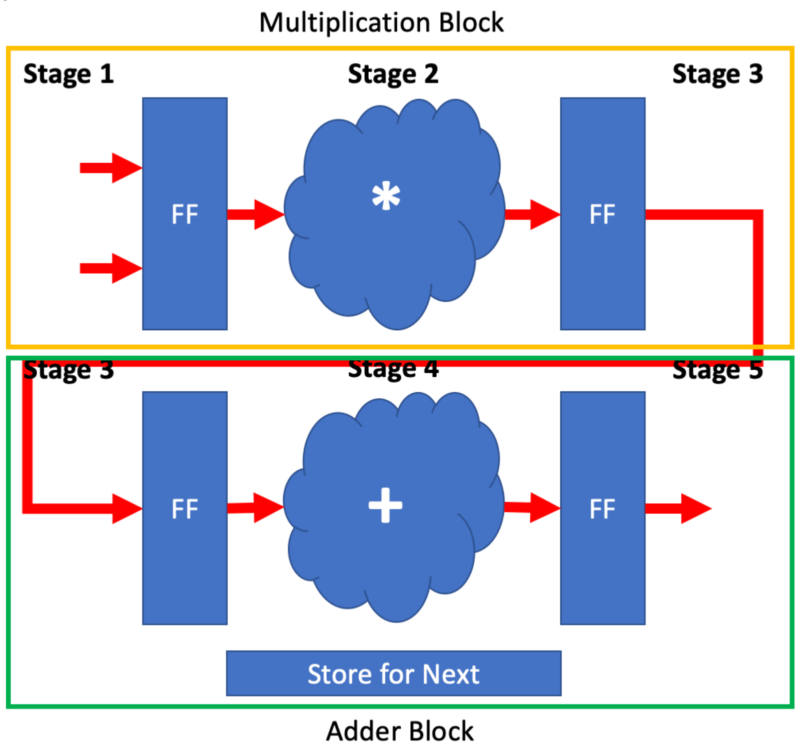


Figure 1: Fully Pipelined dot product.

However due to time constraints the finalized product became pipelined adders connected to a pipelined multiplier. To make it easier to understand the results, after each output there is a array that stores the following values.

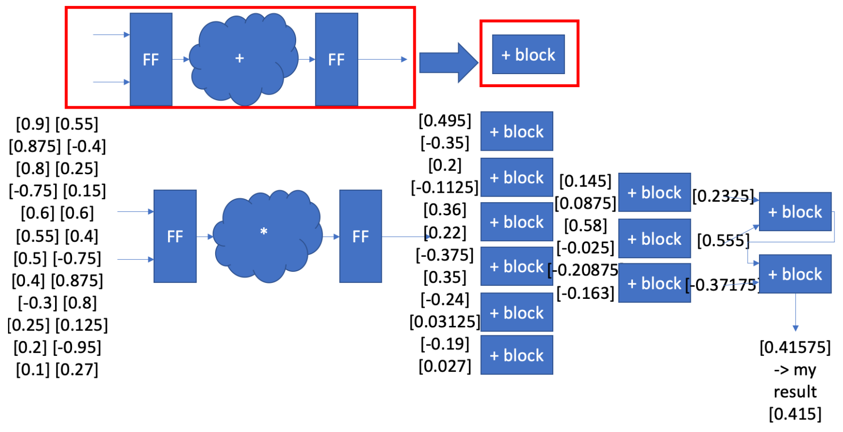


Figure 2: Finalized pipelined multiplier and adder.

As seen in the figure, a bunch of values are sent into the dot product. The dot product is calculated through, A_1*B_1+A_2*B_2…. = dot product. So therefore all vales are first multiplied for each vector value, ranging from 0 -> 11. Resulting in a 12 by 16 bit array. As the last value has passed through the pipelined multiplier, it is passed through pipelined adders, which then pushes it into the the next array. Then it is passed through another set of piplined adders to another array. Which in the end goes through a final adder hat creates a dot product value of ((0.415-0.41575)/0.41575))*100 = 0.18% error. When calculating the value with a standard calculator on the computer.

In summary of the results, one has a 0.998% error, possibly due to rounding errors within the code. If I had more time, I would have spent more time on the rounding for the adders. However, the half precision will also create some errors. However, a 0.18% error is appreciated.

Description of the design:

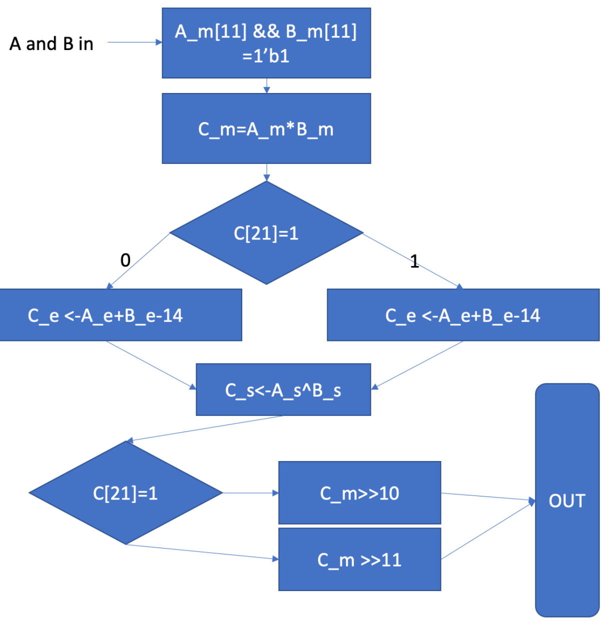


Figure 3: Multiplier

Multiplier explanation:

As the value is pushed in, the mantissa is taken out and granted a 1 in the eleventh bit. Thereafter, the mantissa is multiplied by eachother, there is a possibility that the value will have a bit in the 21st bit position, which will later force us to increase the exponent. Thereafter, the sign is checked by examining, if 1 value is different, the multiplier will push out a negative value. Since the exponent had to be increased, one also has to shift back the multiplied data by the amount from the highest bit, most significant bit -10, as th value could be at 21st position, it would then be pushed out as shifted to the left by 11 and the resulting last [9:0] bit will be the resulting matissa. The calculation is then finished.

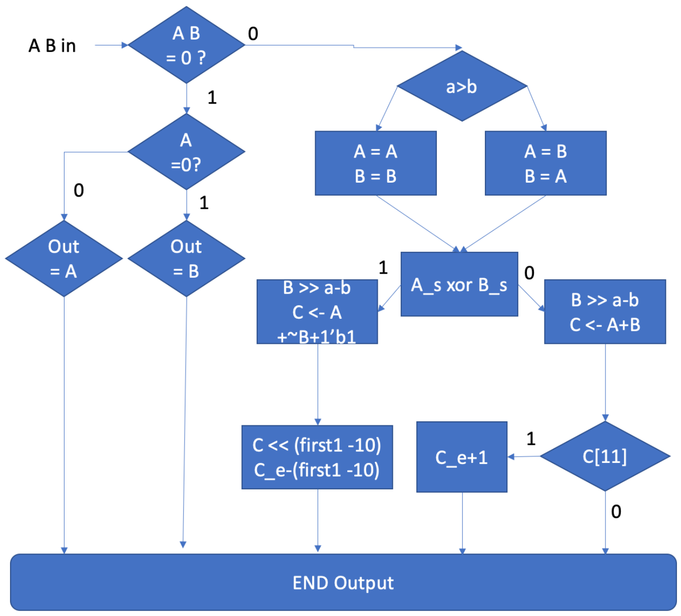


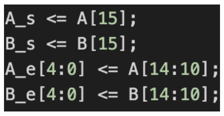
Figure 4: Adder

Adder explanation:

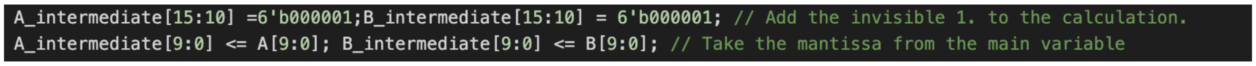
Seen in the block diagram, we can see that data A and B enters the adder. First thing is that it checks if there are any zeros in the input. If so, the other value is directly the output. If there are no 0s the code looks for the larger value. If A has a larger exponent value, it stays as A, if B has a larger value in the exponent, A =B and B=A, if the values have the same size it checks against the mantissa. After having checked the sizes, the code pushes in the code after the clock signal, as it has a non-blocking statement. The first thing that it does is checking the signs, are they different? If they are, the negative value goes for 2s complement. But before 2s complement is done, it is shifted by the exponent difference. Thereafter they are added and put against the flip-flop, waiting to be pushed out. If the value had the same signs, the last sign will be the same as the large sign. If the value had the same sign, the value are shifted and then added, thereafter, put against the flip-flop. At the next clock cycle the data is aligned. If it was a negative sign for the smaller value we shift the value back to its place dependent on the first 1, excluding the negative sign 1. We then have to decrease the exponent of the large value, which will be the output. If the values were both positive and at the last but is 1, then we need to shift the mantissa back and add to the large exponent. The values has now been aligned, added, and normalized.

Simulation/Verification:

Seen in the multiplier waveform, values are inputted at A and B. As A and B are put into the system we can see that The values are directly aligned.

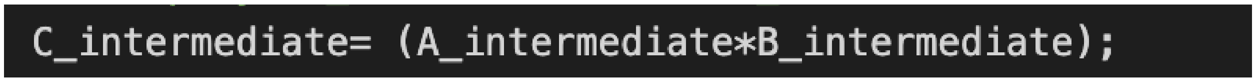


```
A_s <= A[15];
B_s <= B[15];
A_e[4:0] <= A[14:10];
B_e[4:0] <= B[14:10];
```



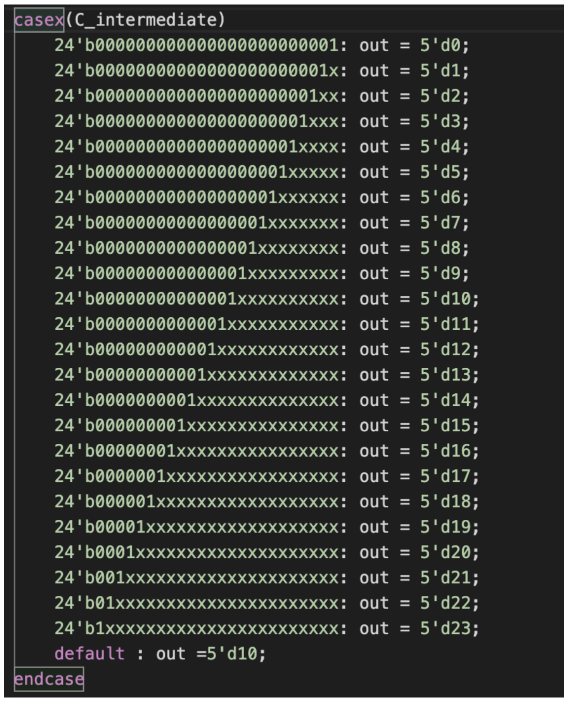
```
A_intermediate[15:10] =6'b000001;B_intermediate[15:10] = 6'b000001; // Add the invisible 1. to the calculation.
A_intermediate[9:0] <= A[9:0]; B_intermediate[9:0] <= B[9:0]; // Take the mantissa from the main variable
```

As soon as the data has been aligned, the code now calculates the resulting mantissa which will be much larger than the values applied in A and B.



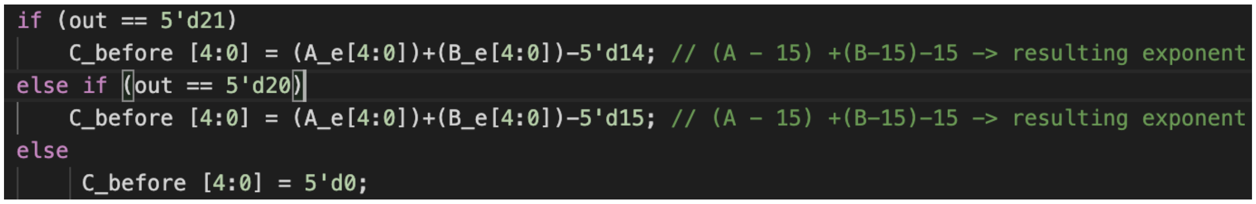
```
C_intermediate= (A_intermediate*B_intermediate);
```

As the result is given, we have to observe the position of the first one in the mantissa, as if it is at position 1 or higher, one must increase the exponent by such. The following code only accounts for up to 21 as the input values only went up to that. This can be seen i the following case statement.



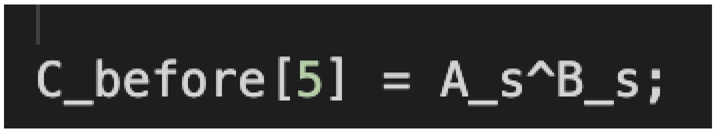
```
casex(C_intermediate)
    24'b000000000000000000000001: out = 5'd0;
    24'b00000000000000000000001x: out = 5'd1;
    24'b0000000000000000000001xx: out = 5'd2;
    24'b000000000000000000001xxx: out = 5'd3;
    24'b00000000000000000001xxxx: out = 5'd4;
    24'b0000000000000000001xxxxx: out = 5'd5;
    24'b000000000000000001xxxxxx: out = 5'd6;
    24'b00000000000000001xxxxxxx: out = 5'd7;
    24'b0000000000000001xxxxxxxx: out = 5'd8;
    24'b000000000000001xxxxxxxxx: out = 5'd9;
    24'b00000000000001xxxxxxxxxx: out = 5'd10;
    24'b0000000000001xxxxxxxxxxx: out = 5'd11;
    24'b000000000001xxxxxxxxxxxx: out = 5'd12;
    24'b00000000001xxxxxxxxxxxxx: out = 5'd13;
    24'b0000000001xxxxxxxxxxxxxx: out = 5'd14;
    24'b000000001xxxxxxxxxxxxxxx: out = 5'd15;
    24'b00000001xxxxxxxxxxxxxxxx: out = 5'd16;
    24'b0000001xxxxxxxxxxxxxxxxx: out = 5'd17;
    24'b000001xxxxxxxxxxxxxxxxxx: out = 5'd18;
    24'b00001xxxxxxxxxxxxxxxxxxx: out = 5'd19;
    24'b0001xxxxxxxxxxxxxxxxxxxx: out = 5'd20;
    24'b001xxxxxxxxxxxxxxxxxxxxx: out = 5'd21;
    24'b01xxxxxxxxxxxxxxxxxxxxxx: out = 5'd22;
    24'b1xxxxxxxxxxxxxxxxxxxxxxx: out = 5'd23;
    default : out =5'd10;
endcase
```

Now been given the value of the out, we can figure out if the exponent will need to increase or not. As said, in the case statement we check against the resulting matissa from the calculator.



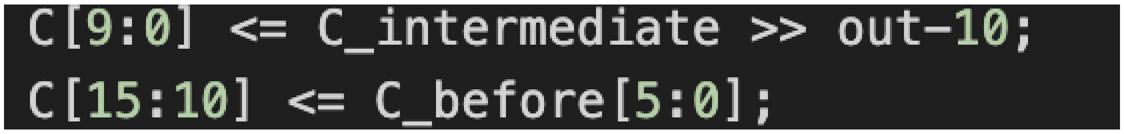
```
if (out == 5'd21)
    C_before [4:0] = (A_e[4:0])+(B_e[4:0])-5'd14; // (A - 15) +(B-15)-15 -> resulting exponent
else if (out == 5'd20)
    C_before [4:0] = (A_e[4:0])+(B_e[4:0])-5'd15; // (A - 15) +(B-15)-15 -> resulting exponent
else
     C_before [4:0] = 5'd0;
```

Now to check the sign value, we just take the xor value of the input sign value of A and b.



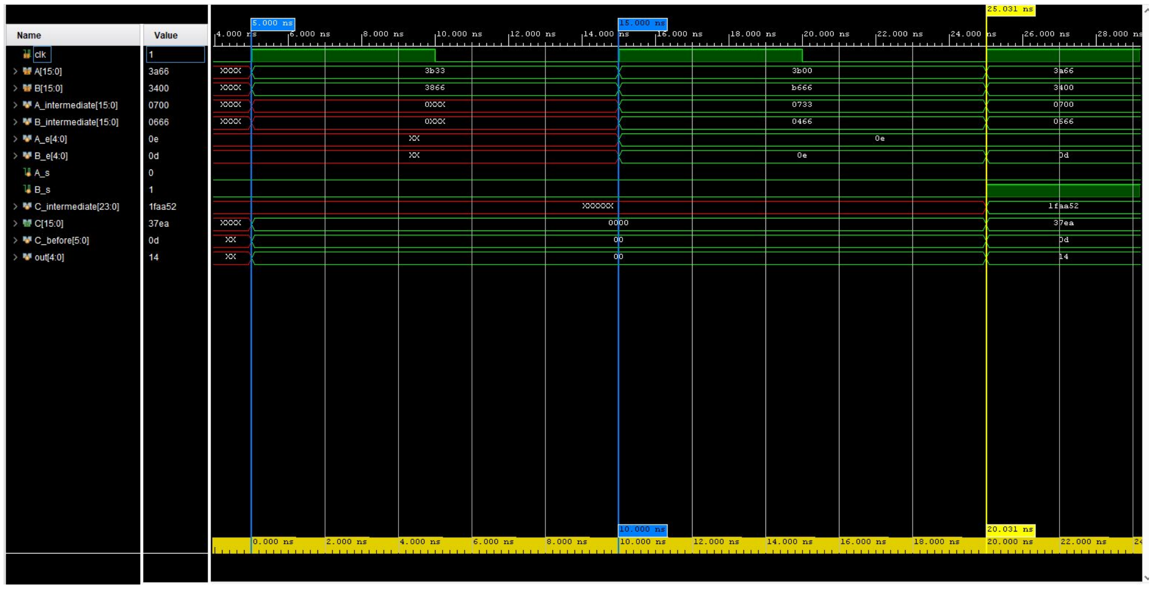
```
C_before[5] = A_s^B_s;
```

Now as the values are prepared, the end of next clock cycle we will wait utnill we can normalize that data and push out the resulting mantissa with the shift dependent on the most significant bit.



```
C[9:0] <= C_intermediate >> out-10;
C[15:10] <= C_before[5:0];
```

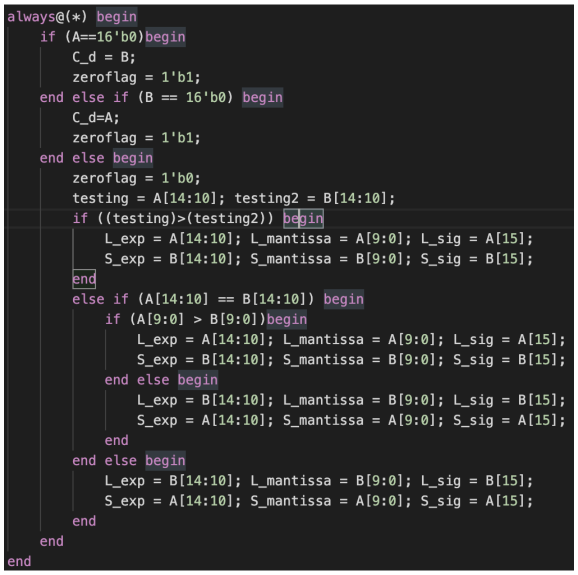
Seen in the vave form below we can see that for each new clock cycle we receive a new value, making sure that we have a pipelined multiplier. Also as seen, in the wave form we normalize the values as soon as the clock has appeared and so aligned it as soon as the clock appears.



Name	Value
clk	1
A[15:0]	3a66
B[15:0]	3400
A_intermediate[15:0]	0700
B_intermediate[15:0]	0666
A_e[4:0]	0e
B_e[4:0]	0d
A_s	0
B_s	1
C_intermediate[23:0]	1faa52
C[15:0]	37ea
C_before[5:0]	0d
out[4:0]	14

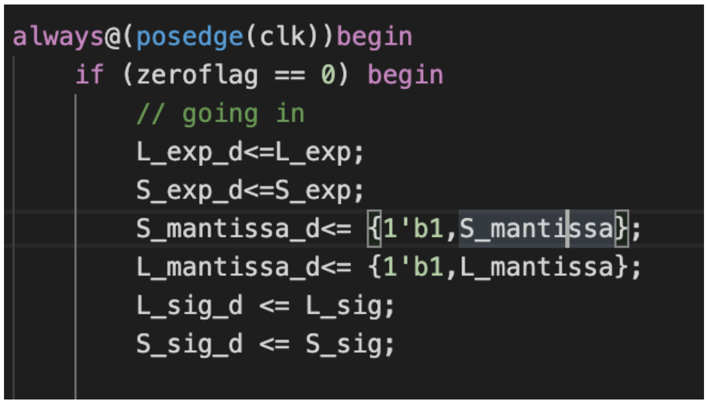
Fig 5. Multiplier

As data is seen in the outside of the adder, it directly goes through checking if what is the larges value, it then organizes the exp, mantissa, and sign. The data will be aligned before the first clk sign as cane be seen in the following code. Were first check if there are any 0s. If there are any 0, it outputs the other value. Then it goes to see the exponent, if exponent are the same it goes to the mantissas.



```
always@(*) begin
    if (A==16'b0)begin
        C_d = B;
        zeroflag = 1'b1;
    end else if (B == 16'b0) begin
        C_d=A;
        zeroflag = 1'b1;
    end else begin
        zeroflag = 1'b0;
        testing = A[14:10]; testing2 = B[14:10];
        if ((testing)>(testing2)) begin
            L_exp = A[14:10]; L_mantissa = A[9:0]; L_sig = A[15];
            S_exp = B[14:10]; S_mantissa = B[9:0]; S_sig = B[15];
        end
        else if (A[14:10] == B[14:10]) begin
            if (A[9:0] > B[9:0])begin
                L_exp = A[14:10]; L_mantissa = A[9:0]; L_sig = A[15];
                S_exp = B[14:10]; S_mantissa = B[9:0]; S_sig = B[15];
            end else begin
                L_exp = B[14:10]; L_mantissa = B[9:0]; L_sig = B[15];
                S_exp = A[14:10]; S_mantissa = A[9:0]; S_sig = A[15];
            end
        end else begin
            L_exp = B[14:10]; L_mantissa = B[9:0]; L_sig = B[15];
            S_exp = A[14:10]; S_mantissa = A[9:0]; S_sig = A[15];
        end
    end
end
```

After that the data has bee aligned it goes through the flip flop. AS it foes in we want to add the 1 bit to the 11th position of the mantissa to include the 1.



```
always@(posedge(clk))begin
    if (zeroflag == 0) begin
        // going in
        L_exp_d<=L_exp;
        S_exp_d<=S_exp;
        S_mantissa_d<= {1'b1,S_mantissa};
        L_mantissa_d<= {1'b1,L_mantissa};
        L_sig_d <= L_sig;
        S_sig_d <= S_sig;
```

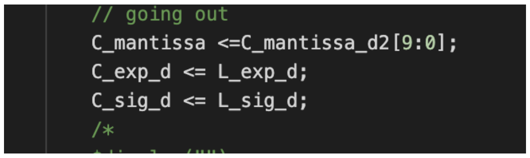
The sign is directly checked for if its negative or not, if it is then we have to do the 2s complement and first shift it by the difference of the exponents. If its positive we just shift them, as seen in C_mantissa_d we the add the S_mantissa_d_sh_sign which has been shifted and in 2s complement dependent on sign to the larger matissa.

```
assign C_mantissa_d = S_mantissa_d_sh_sign + L_mantissa_d;
   assign S_mantissa_d_sh_sign = (L_sig_d^S_sig_d)?(~(S_mantissa_d>>(L_exp_d -
S_exp_d)) +11'b1):(S_mantissa_d>>(L_exp_d - S_exp_d));// ahhh 2s complement
```

As soon as the C_mantissa_d we are checking for the most significant bit. Because we have to shift the data back to the position. If it is negative it is possible that the value is smaller and will have a smaller exponent as seen in the code below.

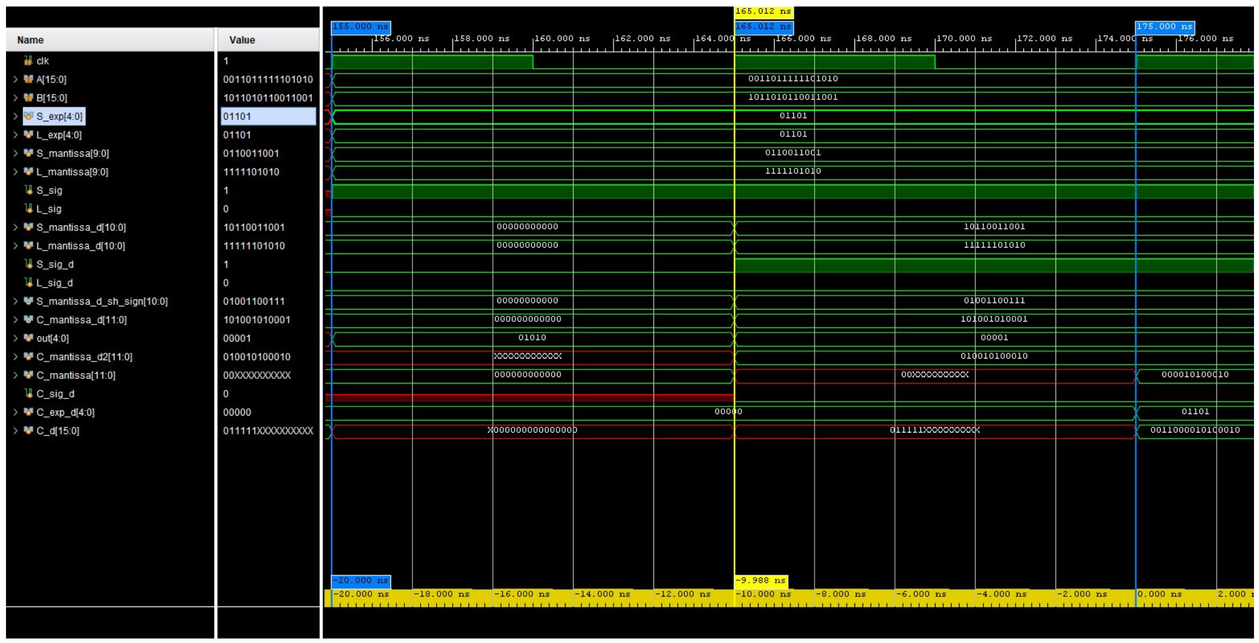
```
always@(*)begin
       if (zeroflag == 0) begin
           casex(C_mantissa_d[10:0])
               11'b00000000001: out = 5'd10;
               11'b0000000001x: out = 5'd9;
               11'b000000001xx: out = 5'd8;
               11'b00000001xxx: out = 5'd7;
               11'b0000001xxxx: out = 5'd6;
               11'b000001xxxxx: out = 5'd5;
               11'b00001xxxxxx: out = 5'd4;
               11'b0001xxxxxxx: out = 5'd3;
               11'b001xxxxxxxx: out = 5'd2;
               11'b01xxxxxxxxx: out = 5'd1;
               11'b1xxxxxxxxxx: out = 5'd0;
               default : out =5'd10;
           endcase
           if ((L_sig_d^S_sig_d) == 1'b1)begin
               C_mantissa_d2 = C_mantissa_d<<(out);
               C_exp = (C_exp_d-out);
               C_d = ({C_sig_d,C_exp,(C_mantissa[9:0])});
           end else begin
           casex(C_mantissa_d[10:0])
               11'b01xxxxxxxxx: out = 5'd0;
               11'b1xxxxxxxxxx: out = 5'd1;
               default : out =5'd10;
           endcase
               //C_mantissa_d2 = C_mantissa_d>>(out);
               C_exp = (C_mantissa_d[11])?(C_exp_d+5'b1):(C_exp_d);
               C_d =
({C_sig_d,C_exp,(C_mantissa_d[11])?(C_mantissa_d[10:1]):(C_mantissa_d[9:0])});
           end
       end
   end
```

As seen in the code the C_mantissa_d2 and C_exp_d are waiting to be give values by the next clk as it will be pushing it out.



```
// going out
C_mantissa <=C_mantissa_d2[9:0];
C_exp_d <= L_exp_d;
C_sig_d <= L_sig_d;
/*
```

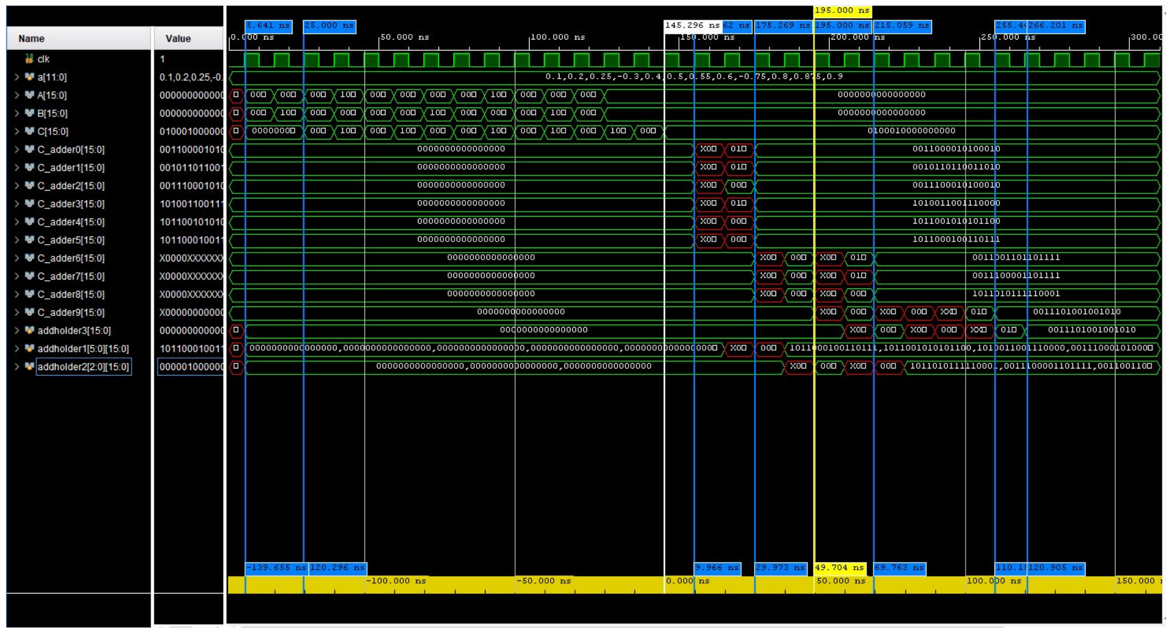
In this step we align them. And the resulting value can be seen of C_d which will take the last 10 bits if the values were negative, but dependent on the mantissa value in the 11th position the value will either be taken in the 10:1 or 9:0 as the exponent will be increased in the normalization. Seen in the wave forms the values with _d is the ones during them multiplication part except for C_d, while the others L and S are for alignment before pushed in, and then C is for normalization after the last flipflop.



Name	Value
clk	1
A[15:0]	0011011111101010
B[15:0]	1011010110011001
S_exp[4:0]	01101
L_exp[4:0]	01101
S_mantissa[9:0]	0110011001
L_mantissa[9:0]	1111101010
S_sig	1
L_sig	0
S_mantissa_d[10:0]	10110011001
L_mantissa_d[10:0]	11111101010
S_sig_d	1
L_sig_d	0
S_mantissa_d_sh_sign[10:0]	01001100111
C_mantissa_d[11:0]	101001010001
out[4:0]	00001
C_mantissa_d2[11:0]	010010100010
C_mantissa[11:0]	00XXXXXXXXXX
C_sig_d	0
C_exp_d[4:0]	00000
C_d[15:0]	011111XXXXXXXXXX

Fig 6. Adder

In the following testbench we can see that I start with pushing my values from a and b into a converter that uses system verilog function $realtobits(), each value is then pushed into the multiplier that can be seen to push out new data out in C. The inputs to the multipliers a A, and B. As we wanted a 3 stage multiplier, one can see that it takes 2 clock cycles for the data to go out, meaning that we have 2 flip flops and finalizes the requirement for the multiplier. For ever value that goes out it goes into an array. As soon as the multiplier is done an the array is full, it takes 1 clk cycle until that data is imputed to the adders. As it takes 2 clock cycles for the data to leave, we can see that it is a 3 stage adder! As the data has been processed 4 times we finally have the correct value at addholeder3, which can be seen at mar 266ns.



Name	Value
clk	1
a[11:0]	0.1,0.2,0.25,-0.
A[15:0]	000000000000
B[15:0]	000000000000
C[15:0]	010001000000
C_adder0[15:0]	001100001010
C_adder1[15:0]	001011011001
C_adder2[15:0]	001110001010
C_adder3[15:0]	101001100111
C_adder4[15:0]	101100101010
C_adder5[15:0]	101100010011
C_adder6[15:0]	X0000XXXXXXX
C_adder7[15:0]	X0000XXXXXXX
C_adder8[15:0]	X0000XXXXXXX
C_adder9[15:0]	X00000000000
addholder3[15:0]	000000000000
addholder1[5:0][15:0]	101100010011
addholder2[2:0][15:0]	000001000000

Fig 7. Testbech

Conclusion:
In conclusion, the report has described a pipelined floating point multiplier and floating point adder to create the dot product. The simulation was ideally based on the first figure 1, however, due to time constraints the simulation was simplified to figure 2, which stores up the results of the simulation in binary arrays. However, this creates a great tool to figure out if the simulation actually provides the correct values for each device. Overall, the simulation and design was finalized and provided proficient results. **It reaches the goals of the lab**. If I had more time, the first figure would be implemented.

Bio:
I'm an international student from Sweden. My first degree was Chemical engineering, however, I had a passion for semiconductor development and physics. Within my BS degree I therefore focused highly on semiconductor processing. Now I'm an MS electrical engineering student at San Jose State University. As I did during my BSCHE, I still have a huge interest for semiconductors and I continue learning more about them. I have been able to publish papers, created presentations, and now have been working with IBM for my master thesis with physical-modeling of materials, thermodynamics, and electrodynamics within Phase change memories, which will be the future of AI technology.

Appendix:

Nothing to Add here.