

UNIVERSITY OF NEVADA LAS VEGAS, DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING LABORATORIES.

Class:	CPE300L Digital Systems Architecture and Design 1001		Semester:	Fall 2024
Points		Document author:	Narek Kalikian	
		Author's email:	kalikn1@unlv.nevada.edu	
		Document topic:	Postlab 6	
Instructor's comments:				

1. Introduction / Theory of Operation

In this lab, we worked on IEEE 754 floating point arithmetic. First, we did hand calculations to understand how to convert decimal numbers such as 7.875 and 0.1875 to binary and then perform arithmetic operations like addition, subtraction, and multiplication on them. After doing so, we were able to better understand the steps that need to be taken to accomplish each operation. Then, we created verilog modules for each floating point arithmetic operation and test benches for testing these operations via waveform simulations.

2. Description of Experiments

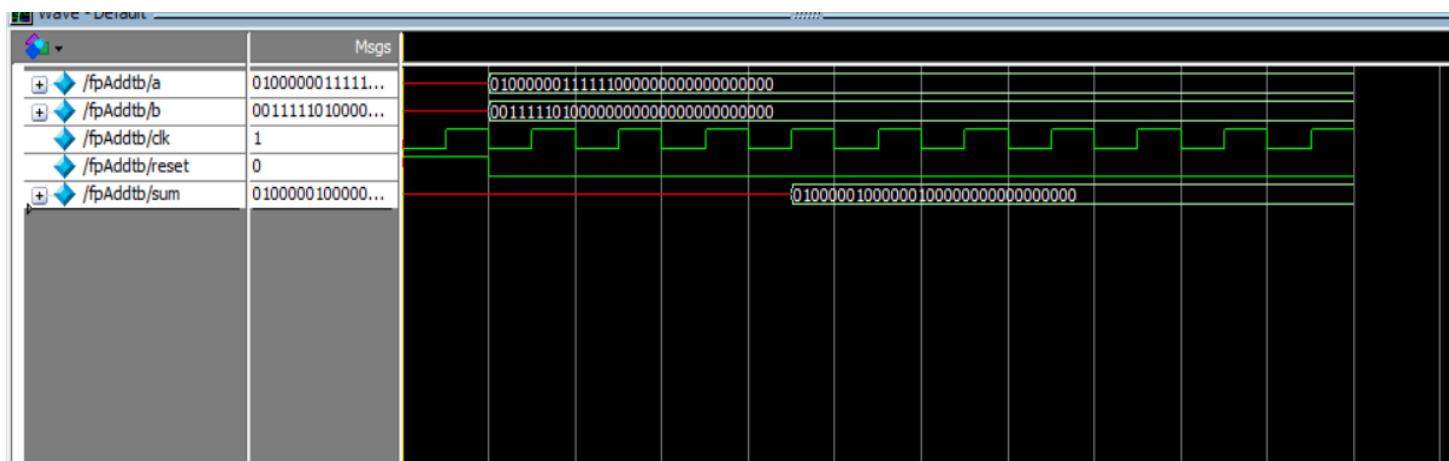
Experiment 1 - Verilog Code:

```
44  module fpAdd(
45    input [31:0] a,
46    input [31:0] b,
47    input wire clk,
48    input wire reset,
49    output reg [31:0] sum
50  );
51  reg [2:0] nextstate, state;
52  parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010, S3 = 3'b011, S4 = 3'b100;
53
54  reg sign_a, sign_b, sign_res;
55  reg [7:0] exp_a, exp_b, exp_res;
56  reg [23:0] frac_a, frac_b, frac_res;
57  reg [24:0] frac_sum;
58  reg overflow, underflow;
59
60  always @(posedge clk or posedge reset) begin
61    if (reset)
62      state <= S0;
63    else
64      state <= nextstate;
65  end
66
67  always @(*) begin
68    case (state)
69    S0: begin
70      // Extract components from input a
71      sign_a = a[31];
72      exp_a = a[30:23];
73      frac_a = {1'b1, a[22:0]};
74      overflow = 1'b0;
75      underflow = 1'b0;
76      nextstate = S1;
77    end
78
79    S1: begin
80      // Extract components from input b
81      sign_b = b[31];
82      exp_b = b[30:23];
83      frac_b = {1'b1, b[22:0]};
84      nextstate = S2;
85    end
86
87    S2: begin
88      // Compare exponents
89      if (exp_a > exp_b) begin
90        frac_b = frac_b >> exp_a - exp_b;
91        exp_res = exp_a;
92      end else if (exp_b > exp_a) begin
93        frac_a = frac_a >> exp_b - exp_a;
94        exp_res = exp_b;
95      end else begin
96        exp_res = exp_a;
97      end
98      nextstate = S3;
99    end
100
101   S3: begin
102     // Add or subtract the fractions
103     if (sign_a == sign_b) begin
104       frac_sum = frac_a + frac_b;
105       sign_res = sign_a;
106     end else begin
107       if (frac_a > frac_b) begin
108         frac_sum = frac_a - frac_b;
109         sign_res = sign_a;
110       end else begin
111         frac_sum = frac_b - frac_a;
112         sign_res = sign_b;
113       end
114     end
115
116     if (frac_sum[24] == 1'b1) begin
117       frac_res = frac_sum >> 1;
118       exp_res = exp_res + 1;
119     end else begin
120       frac_res = frac_sum[23:0];
121     end
122
123     if (frac_sum == 0) begin
124       exp_res = 8'b0;
125     end
126
127     nextstate = S4;
128   end
129
130   S4: begin
131     if (exp_res == 8'hFF) begin
132       overflow = 1'b1;
133     end else if (exp_res == 8'h00) begin
134       underflow = 1'b1;
135     end
136
137     sum = {sign_res, exp_res, frac_res[22:0]};
138
139     nextstate = S0;
140   end
141   default: nextstate = S0;
142 endcase
143 end
144 endmodule
```

Experiment 1 - Testbench:

```
1  module fpAddtb;
2
3      reg [31:0] a;
4      reg [31:0] b;
5      reg clk;
6      reg reset;
7
8      wire [31:0] sum;
9
10     fpAdd uut (
11         .a(a),
12         .b(b),
13         .clk(clk),
14         .reset(reset),
15         .sum(sum)
16     );
17
18     initial begin
19         clk = 0;
20         forever #5 clk = ~clk;
21     end
22
23     initial begin
24         reset = 1;
25         #10;
26         reset = 0;
27
28         a = 32'b01000000111110000000000000000000; // 7.875
29         b = 32'b00111110100000000000000000000000; // 0.1875
30
31         #100;
32
33         $stop;
34     end
35
36     initial begin
37         $monitor("At time %t: a = %b, b = %b, sum = %b",
38                  $time, a, b, sum);
39     end
40
41 endmodule
42
43
```

Experiment 1 - Waveform Simulation:



Experiment 2 - Verilog Code:

```

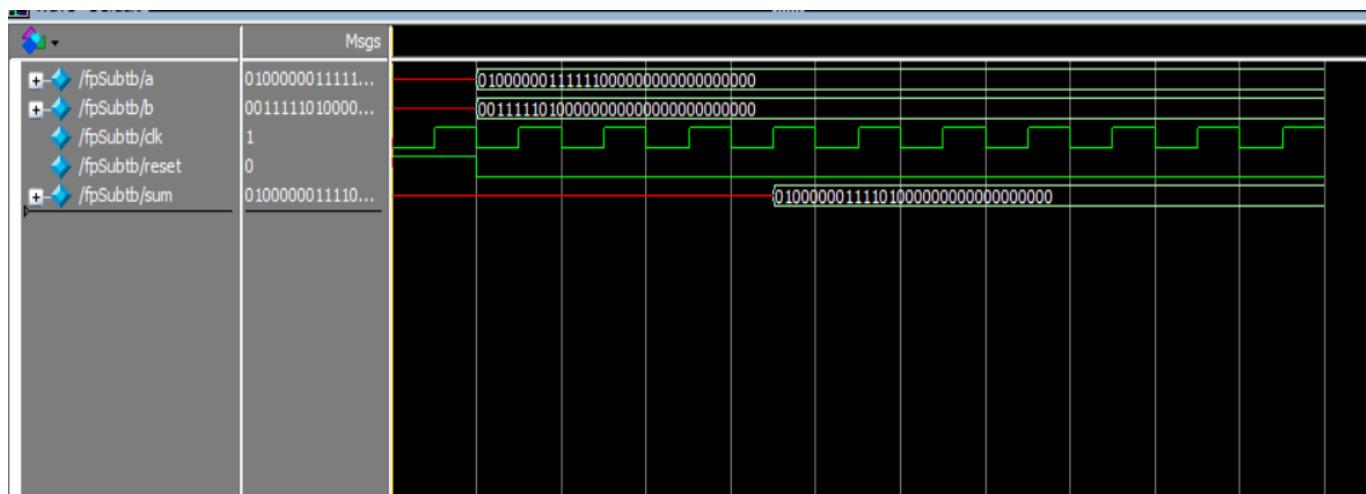
44  module fpSub(
45      input [31:0] a,
46      input [31:0] b,
47      input wire clk,
48      input wire reset,
49      output reg [31:0] sum
50  );
51      reg [2:0] nextstate, state;
52      parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010, S3 = 3'b011, S4 = 3'b100;
53
54      reg sign_a, sign_b, sign_res;
55      reg [7:0] exp_a, exp_b, exp_res;
56      reg [23:0] frac_a, frac_b, frac_res;
57      reg [24:0] frac_sum;
58      reg overflow, underflow;
59
60      always @(posedge clk or posedge reset) begin
61          if (reset)
62              state <= S0;
63          else
64              state <= nextstate;
65      end
66
67      always @(*) begin
68          case (state)
69          S0: begin
70              // Extract components from input a
71              sign_a = a[31];
72              exp_a = a[30:23];
73              frac_a = {1'b1, a[22:0]};
74              overflow = 1'b0;
75              underflow = 1'b0;
76              nextstate = S1;
77          end
78
79          S1: begin
80              // Extract components from input b
81              sign_b = b[31];
82              exp_b = b[30:23];
83              frac_b = {1'b1, b[22:0]};
84              nextstate = S2;
85          end
86
87          S2: begin
88              // Compare exponents
89              if (exp_a > exp_b) begin
90                  frac_b = frac_b >> exp_a - exp_b;
91                  exp_res = exp_a;
92              end else if (exp_b > exp_a) begin
93                  frac_a = frac_a >> exp_b - exp_a;
94                  exp_res = exp_b;
95              end else begin
96                  exp_res = exp_a;
97              end
98              nextstate = S3;
99          end
100
101         S3: begin
102             // Add or subtract the fractions
103             if (sign_a == sign_b) begin
104                 frac_sum = frac_a - frac_b;
105                 sign_res = sign_a;
106             end else begin
107                 if (frac_a > frac_b) begin
108                     frac_sum = frac_a - frac_b;
109                     sign_res = sign_a;
110                 end else begin
111                     frac_sum = frac_b - frac_a;
112                     sign_res = sign_b;
113                 end
114             end
115
116             if (frac_sum[24] == 1'b1) begin
117                 frac_res = frac_sum >> 1;
118                 exp_res = exp_res + 1;
119             end else begin
120                 frac_res = frac_sum[23:0];
121             end
122
123             if (frac_sum == 0) begin
124                 exp_res = 8'b00;
125             end
126
127             nextstate = S4;
128         end
129
130         S4: begin
131             if (exp_res == 8'hFF) begin
132                 overflow = 1'b1;
133             end else if (exp_res == 8'h00) begin
134                 underflow = 1'b1;
135             end
136
137             sum = (sign_res, exp_res, frac_res[22:0]);
138
139             nextstate = S0;
140         end
141         default: nextstate = S0;
142     endcase
143
144 endmodule

```

Experiment 2 - Testbench:

```
1  module fpSubtb;
2
3      reg [31:0] a;
4      reg [31:0] b;
5      reg clk;
6      reg reset;
7
8      wire [31:0] sum;
9
10     fpSub uut (
11         .a(a),
12         .b(b),
13         .clk(clk),
14         .reset(reset),
15         .sum(sum)
16     );
17
18     initial begin
19         clk = 0;
20         forever #5 clk = ~clk;
21     end
22
23     initial begin
24         reset = 1;
25         #10;
26         reset = 0;
27
28         a = 32'b01000000111110000000000000000000; // 7.875
29         b = 32'b00111110100000000000000000000000; // 0.1875
30
31         #100;
32
33         $stop;
34     end
35
36     initial begin
37         $monitor("At time %t: a = %b, b = %b, diff = %b",
38                  $time, a, b, sum);
39     end
40
41 endmodule
```

Experiment 2 - Waveform Simulation:



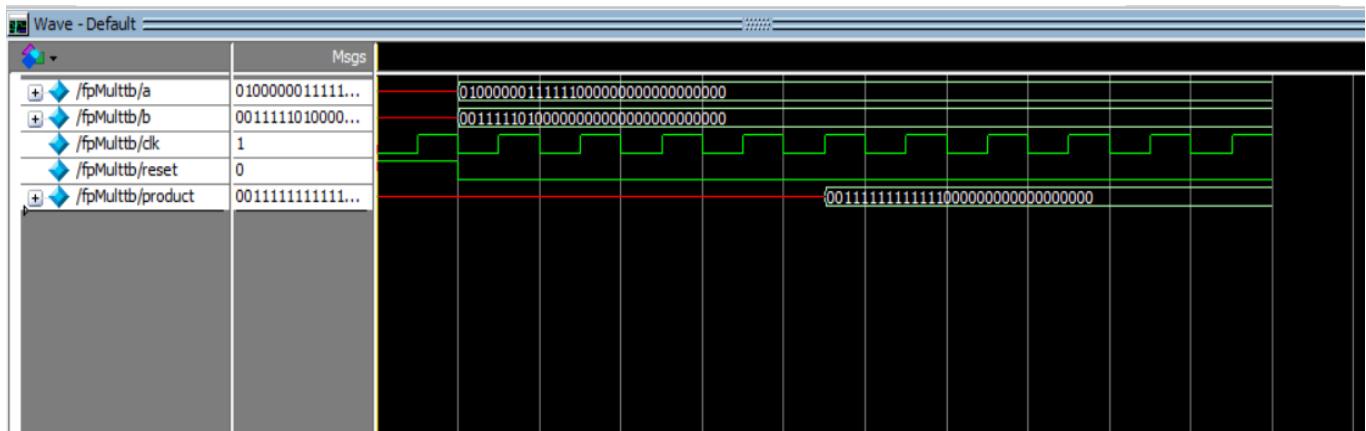
Experiment 3 - Verilog Code:

```
42  module fpMult(
43    input [31:0] a,
44    input [31:0] b,
45    input wire clk,
46    input wire reset,
47    output reg [31:0] product
48  );
49  reg [23:0] mantissa_a, mantissa_b, mantissa_res;
50  reg [7:0] exponent_a, exponent_b, exponent_res;
51  reg sign_a, sign_b, sign_res;
52  reg [47:0] mantissa_mult;
53  reg [7:0] exponent_sum;
54  reg [7:0] exponent_bias = 8'd127;
55
56  reg [2:0] state, nextstate;
57  parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010, S3 = 3'b011, S4 = 3'b100, S5 = 3'b101;
58
59  always @(posedge clk or posedge reset) begin
60    if (reset)
61      state <= S0;
62    else
63      state <= nextstate;
64  end
65
66  always @(*) begin
67    case (state)
68      S0: begin
69        sign_a = a[31];
70        sign_b = b[31];
71        exponent_a = a[30:23];
72        exponent_b = b[30:23];
73        mantissa_a = (1'b1, a[22:0]);
74        mantissa_b = (1'b1, b[22:0]);
75        nextstate = S1;
76      end
77
78      S1: begin
79        exponent_sum = (exponent_a + exponent_b) - exponent_bias;
80        nextstate = S2;
81      end
82
83      S2: begin
84        mantissa_mult = mantissa_a * mantissa_b;
85        nextstate = S3;
86      end
87
88      S3: begin
89        if (mantissa_mult[47] == 1'b1) begin
90          mantissa_res = mantissa_mult[47:24];
91          exponent_res = exponent_sum + 1;
92        end
93        else begin
94          mantissa_res = mantissa_mult[46:23];
95          exponent_res = exponent_sum;
96        end
97        nextstate = S4;
98      end
99
100     S4: begin
101       if (exponent_res >= 8'hFF) begin
102         exponent_res = 8'hFF;
103         mantissa_res = 0;
104       end
105       else if (exponent_res <= 8'h00) begin
106         exponent_res = 0;
107         mantissa_res = 0;
108       end
109       nextstate = S5;
110     end
111
112     S5: begin
113       sign_res = sign_a ^ sign_b;
114
115       product = {sign_res, exponent_res, mantissa_res[22:0]};
116       nextstate = S0;
117     end
118   endcase
119 end
120 endmodule
```

Experiment 3 - Testbench:

```
1  module fpMulttb;
2
3      reg [31:0] a;
4      reg [31:0] b;
5      reg clk;
6      reg reset;
7
8      wire [31:0] product;
9
10     fpMult uut (
11         .a(a),
12         .b(b),
13         .clk(clk),
14         .reset(reset),
15         .product(product)
16     );
17
18     initial begin
19         clk = 0;
20         forever #5 clk = ~clk;
21     end
22
23     initial begin
24         reset = 1;
25         #10;
26         reset = 0;
27
28         a = 32'b01000000111110000000000000000000; // 7.875
29         b = 32'b00111101000000000000000000000000; // 0.1875
30
31         #100;
32         $stop;
33     end
34
35     initial begin
36         $monitor("At time %t: a = %b (%f), b = %b (%f), product = %b (%f)",
37                  $time, a, $bitstoreal(a), b, $bitstoreal(b), product, $bitstoreal(product));
38     end
39
40 endmodule
41
```

Experiment 3 - Waveform Simulation:



3. Questions and Answers

1. Rounding: Rounding in IEEE 754 floating point arithmetic is the process of reducing the precision of the operation's result so that it can fit within the available number of bits. Rounding should be performed when the operation exceeds the precision of the IEEE 754 format.

2. Single Precision vs Double Precision: The difference between the two types of floating point precision is the number of bits used to represent the numbers. Single precision floating point arithmetic uses 32-bit numbers while double precision floating point arithmetic uses 64-bit numbers. Double precision should be used if much larger numbers need to be represented that wouldn't fit the 32-bit single precision format.

4. Conclusions

Understanding the IEEE 754 standard for floating point arithmetic is important for being able to convert decimal numbers to binary and executing various operations on given numbers. This is a necessity in logic design because it provides the standard for robust numerical calculations on computers.