```verilog
`timescale 1ns / 10ps

module FPU(
    input              clk       ,
    input              rst       ,
    input       [31:0] dividend  ,
    input       [31:0] divisor   ,
    output reg  [31:0] quotient
);
    //define state numbers
    parameter RESET = 0, WAIT = 1, NEXT = 2, SHIFT = 3, CHECK = 4, ROUND = 5, DONE = 6;

    reg [47:0] new_divisor       ;
    reg [47:0] remainder         ;
    reg [24:0] temp_quotient     ;
    reg [8 :0] quotient_exponent ; // 9bits, since (exponent-127) ranges from  -127 to 380
    reg [7 :0] temp_exponent     ;
    reg [4 :0] flag              ;
    reg [2 :0] state             ;
    reg [2 :0] next_state        ;
    reg [3 :0] special = 0       ; // special cases (NaN,infinity or zero)

    //Three-Stage FSM
    always @(posedge clk) begin
        if (rst)
            state <= RESET;
        else
            state <= next_state;
    end

    always @(state) begin
        case(state)
            RESET:
                next_state = WAIT;
            WAIT:
                next_state = NEXT;
            NEXT:
                next_state = SHIFT;
            SHIFT:
                if(special == 0)        // normal case
                    next_state = CHECK;
                else if(special == 10)  // downflow
                    next_state = NEXT;
                else                    // special cases expect downflow
                    next_state = DONE;
            CHECK:
                next_state = (flag == 0)? ROUND : SHIFT;
            ROUND:
                next_state = DONE;
            DONE:
```

```
                next_state = DONE;
            default: ;
        endcase
    end

    always @(posedge clk) begin
        case(state)
            RESET: begin
                // initial
                special           <= 0 ;
                quotient          <= 0 ;
                remainder         <= 0 ;
                new_divisor       <= 0 ;
                quotient_exponent <= 0 ;
            end
            WAIT: begin
                // special cases (NaN,infinity or zero)
                if((dividend[30:23] != 8'b1111_1111 && dividend[30:0] != 0) && (divisor[30:23] =
                    // special case 1 : x / infinity = 0 (x not NaN,infinity or zero)
                    special <= 1 ;
                end
                else if((dividend[30:23] != 8'b1111_1111 && dividend[30:0] != 0) && (divisor[30:
                    // special case 2 : x / 0 = infinity (x not NaN,infinity or zero)
                    special <= 2 ;
                end
                else if((dividend[30:0] == 0) && (divisor[30:0] == 0)) begin
                    // special case 3 : 0 / 0 = NaN
                    special <= 3 ;
                end
                else if((dividend[30:23] == 8'b1111_1111 && dividend[22:0] == 0) && (divisor[30:
                    // special case 4 : infinity / infinity = NaN
                    special <= 4 ;
                end
                else if((dividend[30:23] == 8'b1111_1111 && dividend[22:0] != 0) || (divisor[30:
                    // special case 5 : dividend or divisor is NaN, quotient is NaN
                    special <= 5 ;
                end
                else if((dividend[30:23] == 8'b1111_1111 && dividend[22:0] == 0) && (divisor[30:
                    // special case 6 : infinity / 0 = infinity
                    special <= 6 ;
                end
                else if((dividend[30:23] == 8'b1111_1111 && dividend[22:0] == 0) && (divisor[30:
                    // special case 7 : infinity / x = infinity (x not NaN,infinity or zero)
                    special <= 7 ;
                end
                else if((dividend[30:0] == 0) && (divisor[30:23] != 8'b1111_1111 && divisor[30:0
                    // special case 8 : zero / x = zero (x not NaN,infinity or zero)
                    special <= 8 ;
                end
                else begin
                    // special case 0 : normal case , special case 9 : overflow and special case
```

```
            remainder     <= (dividend[30:23] == 0)? {1'b0, dividend[22:0], {24{1'b0}}}
            temp_quotient <= 0 ;
            if(dividend[22:0] >= divisor[22:0]) begin
                // dividend's mantissa >= divisor's mantissa
                quotient_exponent <= dividend[30:23] - divisor[30:23] + 127 ;
                new_divisor       <= (divisor[30:23] == 0)? {1'b0, divisor[22:0], {24{1'
                temp_exponent     <= dividend[30:23] ;
            end
            else begin
                // dividend's mantissa < divisor's mantissa
                // when divisor is unnormalized, it should be specially considered
                quotient_exponent <= (divisor[30:23] == 0)? (dividend[30:23] - divisor[3
                new_divisor       <= (divisor[30:23] == 0)? {divisor[22:0], {25{1'b0}}}
                temp_exponent     <= dividend[30:23] - 1 ;
            end
        end
    end
    NEXT: begin
        // determine the number of SHIFT
        // when the quotient of two normalized floating point number is unnormalized, it
        flag <= (quotient_exponent == 0 && dividend[30:23] != 0 && divisor[30:23] != 0)?

        // check overflow or downflow
        if((dividend[22:0] >= divisor[22:0] && dividend[30:23] + 127 >= divisor[30:23] &
        || (dividend[22:0] <  divisor[22:0] && dividend[30:23] + 126 >= divisor[30:23] &
        ) begin
            // check overflow
            // special case 9 : overflow, quotient is infinity
            special <= (quotient_exponent > 254)? 9 : 0 ;
        end
        else if(special == 0 || special == 10) begin
            // downflow
            // special case 10 : downflow, quotient will shift right until the exponent
            special           <= (temp_exponent - divisor[30:23] + 126 == 0)? 0 : 10 ;
            quotient_exponent <= 0                                 ;
            remainder         <= {1'b0, remainder[47:1]} ;
        end
        else ;
    end
    SHIFT: begin
        if(special != 10) begin
            if(remainder >= new_divisor) begin
                temp_quotient <= {temp_quotient[23:0], 1'b1} ;
                remainder     <= remainder - new_divisor     ;
                new_divisor   <= {1'b0, new_divisor[47:1]}   ;
            end
            else begin
                temp_quotient <= {temp_quotient[23:0], 1'b0} ;
                remainder     <= remainder                   ;
                new_divisor   <= {1'b0, new_divisor[47:1]}   ;
            end
```

```
            flag <= flag - 1;
        end
        else begin
            temp_exponent <= temp_exponent + 1 ;
        end
    end
    CHECK: begin
        if(flag == 0) begin //Rounding mode
            if((remainder == new_divisor && temp_quotient[0] == 1'b0) || (remainder < ne
                ;
            end
            else begin
                temp_quotient <= temp_quotient + 1 ;
            end
        end
        else ;
    end
    ROUND: begin
        if(temp_quotient[24] == 1) begin
            quotient_exponent <= quotient_exponent + 1       ;
            temp_quotient     <= {1'b0, temp_quotient[23:0]} ;
        end
        else ;
    end
    DONE: begin
        quotient[31] <= dividend[31] ^ divisor[31] ; // determine the sign bit
        case(special)
            0: begin
                // normal case
                quotient[30:23] <= quotient_exponent[7:0] ;
                quotient[22: 0] <= (quotient_exponent[7:0] == 8'b1111_1111)? 0 : temp_qu
            end
            1: begin
                //  x / infinity = 0 (x not NaN,infinity or zero)
                quotient[30:23] <= 0 ;
                quotient[22: 0] <= 0 ;
            end
            2: begin
                //  x / 0 = infinity (x not NaN,infinity or zero)
                quotient[30:23] <= 8'b1111_1111 ;
                quotient[22: 0] <= 0            ;
            end
            3: begin
                //  0 / 0 = NaN
                quotient[30:23] <= 8'b1111_1111 ;
                quotient[22: 0] <= {23{1'b1}}   ;
            end
            4: begin
                //  infinity / infinity = NaN
                quotient[30:23] <= 8'b1111_1111 ;
                quotient[22: 0] <= {23{1'b1}}   ;
```

```
                end
                5: begin
                    //  dividend or divisor is NaN, quotient is NaN
                    quotient[30:23] <= 8'b1111_1111 ;
                    quotient[22: 0] <= {23{1'b1}}   ;
                end
                6: begin
                    //  infinity / 0 = infinity
                    quotient[30:23] <= 8'b1111_1111 ;
                    quotient[22: 0] <= 0            ;
                end
                7: begin
                    //  infinity / x = infinity (x not NaN,infinity or zero)
                    quotient[30:23] <= 8'b1111_1111 ;
                    quotient[22: 0] <= 0            ;
                end
                8: begin
                    //  zero / x = zero (x not NaN,infinity or zero)
                    quotient[30:23] <= 0            ;
                    quotient[22: 0] <= 0            ;
                end
                9: begin
                    // overflow, output should be infinity
                    quotient[30:23] <= 8'b1111_1111 ;
                    quotient[22: 0] <= 0            ;
                end
                default: ;
            endcase
        end
        default: ;
    endcase

end

endmodule
```