IEEE 754 Double Precision FPU Operation Explanations

Test Case Explanations

1. ADD: 1.5 + 2.0 = 3.5

- A = 1.5 ⇒ Hex: `3FF8000000000000`
- B = 2.0 ⇒ Hex: `4000000000000000`
- IEEE 754 Format: $(-1)^S \times 1.M \times 2^{(E-1023)}$
- A: Sign = 0, Exponent = 1023, Mantissa = 1.5
  ⇒ $1.5 = 1.1_2$
- B: Sign = 0, Exponent = 1024, Mantissa = 1.0
  ⇒ $2.0 = 10.0_2$
- Align mantissas (shift A's mantissa):
  $1.5 \rightarrow 0.75$ when aligning exponents
- Add: $0.75 + 1.0 = 1.75$
- Normalize: $1.75 = 1.11_2$, exponent = 1025
- Result = `400C000000000000` (3.5)

2. SUB: 3.5 - 2.0 = 1.5

- A = 3.5 ⇒ `400C000000000000`
- B = 2.0 ⇒ `4000000000000000`
- A: $3.5 = 11.1_2$, exponent = 1026
- B: $2.0 = 10.0_2$, exponent = 1024
- Align B's mantissa to A's exponent: shift right 2 places ⇒ 0.5
- Subtract: $1.75 - 0.5 = 1.25$
- Normalize: $1.25 = 1.01_2$, exponent = 1023
- Result = `3FF8000000000000` (1.5)

3. MUL: 1.5 × 2.0 = 3.0

- A = 1.5 ⇒ `3FF8000000000000`
- B = 2.0 ⇒ `4000000000000000`
- Exponents: 1023 + 1024 = 2047 ⇒ Bias = 1023 ⇒ Final = 1024
- Mantissas: $1.5 \times 2.0 = 3.0$
- Normalize: $3.0 = 1.5 \times 2^1$, exponent = 1025
- Result = `4008000000000000`

4. DIV: 3.5 ÷ 0.5 = 7.0

- A = 3.5 ⇒ `400C000000000000`
- B = 0.5 ⇒ `3FE0000000000000`
- Exponents: 1026 - 1022 = 4 + bias = 1027
- Mantissas: 3.5/0.5 = 7.0
- Normalize: $7.0 = 1.75 \times 2^2$, exponent = 1025 + 2 = 1027
- Result = `401C000000000000`

5. ADD ZERO: 0 + 0 = 0

- A = 0.0 ⇒ `0000000000000000`
- B = 0.0 ⇒ `0000000000000000`
- Both are zeros, result is zero.
- Result = `0000000000000000`

6. MUL INF: INF × 2.0 = INF

- A = +INF ⇒ `7FF0000000000000`
- B = 2.0 ⇒ `4000000000000000`
- INF × finite = INF
- Result = `7FF0000000000000`

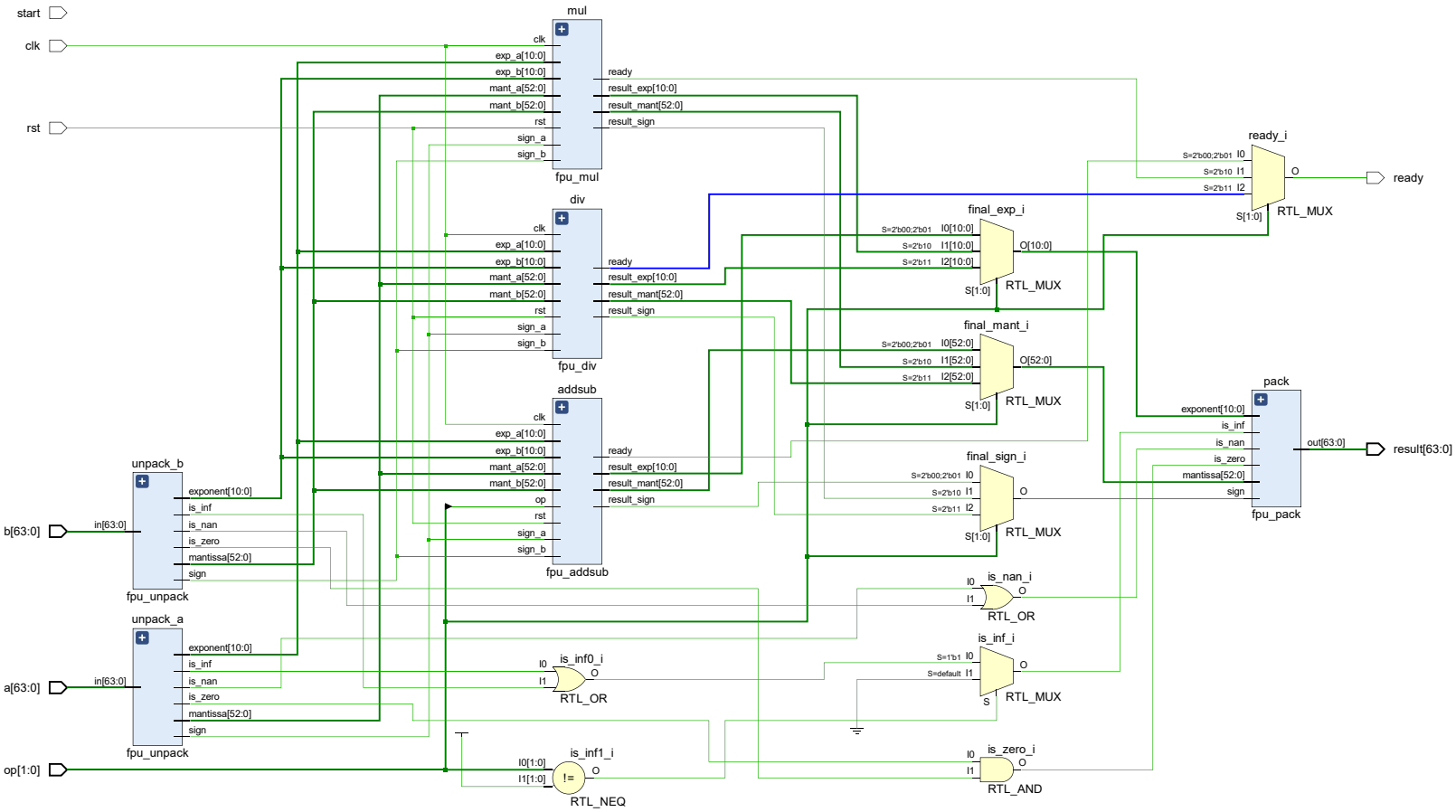
7. ADD NaN: NaN + 1.5 = NaN

- A = NaN ⇒ `7FF8000000000000`
- B = 1.5 ⇒ `3FF8000000000000`
- Any operation with NaN results in NaN
- Result = `7FF8000000000000`

```
`timescale 1ns / 1ps

module fpu_addsub (
    input  wire        clk,
    input  wire        rst,
    input  wire        op,              // 0 = add, 1 = subtract
    input  wire        sign_a,
    input  wire [10:0] exp_a,
    input  wire [52:0] mant_a,          // includes implicit 1 (total 53 bits)
    input  wire        sign_b,
    input  wire [10:0] exp_b,
    input  wire [52:0] mant_b,          // includes implicit 1 (total 53 bits)
    output reg         result_sign,
    output reg  [10:0] result_exp,
    output reg  [52:0] result_mant,
    output reg         ready
);

    // Internal variables
    reg [10:0] exp_diff;
    reg [52:0] aligned_mant_a, aligned_mant_b;
    reg [53:0] mant_sum;
    reg [53:0] mant_diff;
    reg        sign_b_eff;
    reg [10:0] exp_max;
    reg [5:0]  shift_amt;
    reg [53:0] mant_norm;
    integer    i;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            result_sign <= 0;
            result_exp  <= 0;
            result_mant <= 0;
            ready       <= 0;
        end else begin
            ready <= 0;

            // Effective sign for operand B (flip if subtract)
            sign_b_eff = (op) ? ~sign_b : sign_b;

            // Exponent alignment
            if (exp_a > exp_b) begin
                exp_diff       = exp_a - exp_b;
                aligned_mant_a = mant_a;
                aligned_mant_b = mant_b >> exp_diff;
```

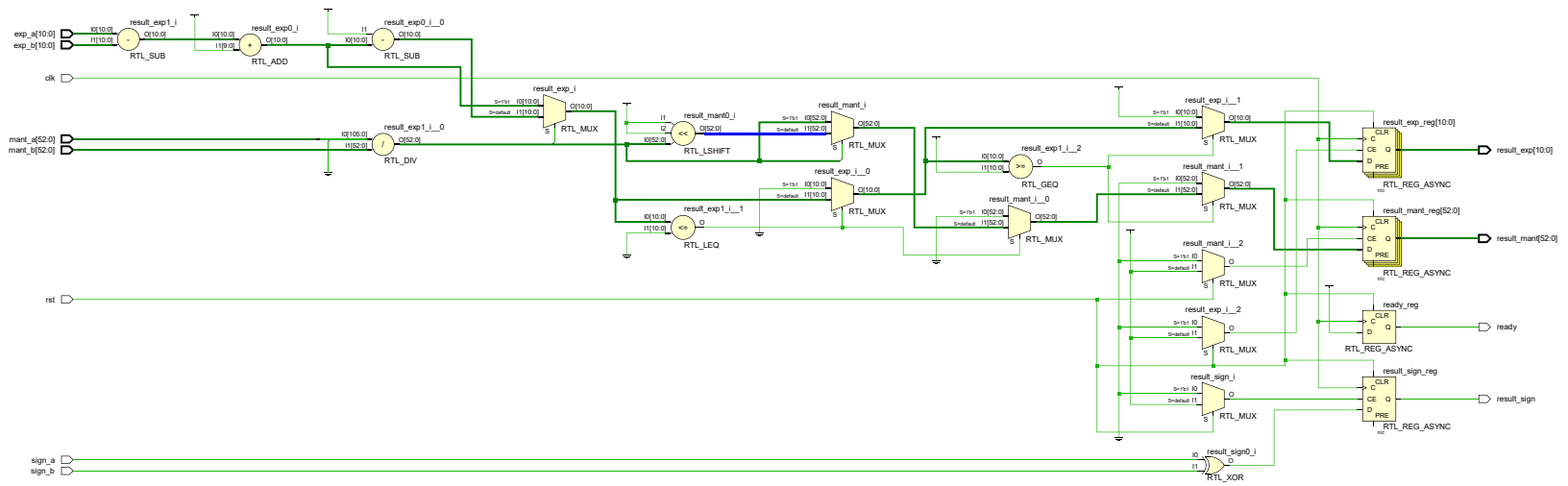
```
    exp_max        = exp_a;
end else begin
    exp_diff       = exp_b - exp_a;
    aligned_mant_a = mant_a >> exp_diff;
    aligned_mant_b = mant_b;
    exp_max        = exp_b;
end

// ADDITION (same signs)
if (sign_a == sign_b_eff) begin
    mant_sum = {1'b0, aligned_mant_a} + {1'b0, aligned_mant_b};
    if (mant_sum[53]) begin
        result_mant = mant_sum[53:1];
        result_exp  = exp_max + 1;
    end else begin
        result_mant = mant_sum[52:0];
        result_exp  = exp_max;
    end
    result_sign = sign_a;
end

// SUBTRACTION (different signs)
else begin
    if (aligned_mant_a >= aligned_mant_b) begin
        mant_diff   = {1'b0, aligned_mant_a} - {1'b0, aligned_mant_b};
        result_sign = sign_a;
    end else begin
        mant_diff   = {1'b0, aligned_mant_b} - {1'b0, aligned_mant_a};
        result_sign = sign_b_eff;
    end

    // Normalize the result
    mant_norm = mant_diff;
    shift_amt = 0;
    for (i = 53; i >= 0; i = i - 1) begin
        if (!shift_amt && mant_norm[i]) begin
            shift_amt = 53 - i;
        end
    end
    result_mant = mant_norm << shift_amt;
    result_exp  = (exp_max > shift_amt) ? (exp_max - shift_amt) : 0;

    // Optional: If result is zero, sign is positive
    if (mant_diff == 0)
        result_sign = 0;
end
```

```
            ready <= 1;
        end
    end
endmodule
```


start
clk
rst
b[63:0]
a[63:0]
op[1:0]
mul
clk
exp_a[10:0]
exp_b[10:0]
mant_a[52:0]
mant_b[52:0]
rst
sign_a
sign_b
ready
result_exp[10:0]
result_mant[52:0]
result_sign
fpu_mul
div
fpu_div
addsub
op
fpu_addsub
unpack_b
unpack_a
in[63:0]
exponent[10:0]
is_inf
is_nan
is_zero
mantissa[52:0]
sign
fpu_unpack
is_inf0_i
RTL_OR
is_inf1_i
!=
RTL_NEQ
ready_i
RTL_MUX
ready
final_exp_i
final_mant_i
final_sign_i
is_nan_i
is_inf_i
is_zero_i
RTL_AND
pack
out[63:0]
result[63:0]
fpu_pack

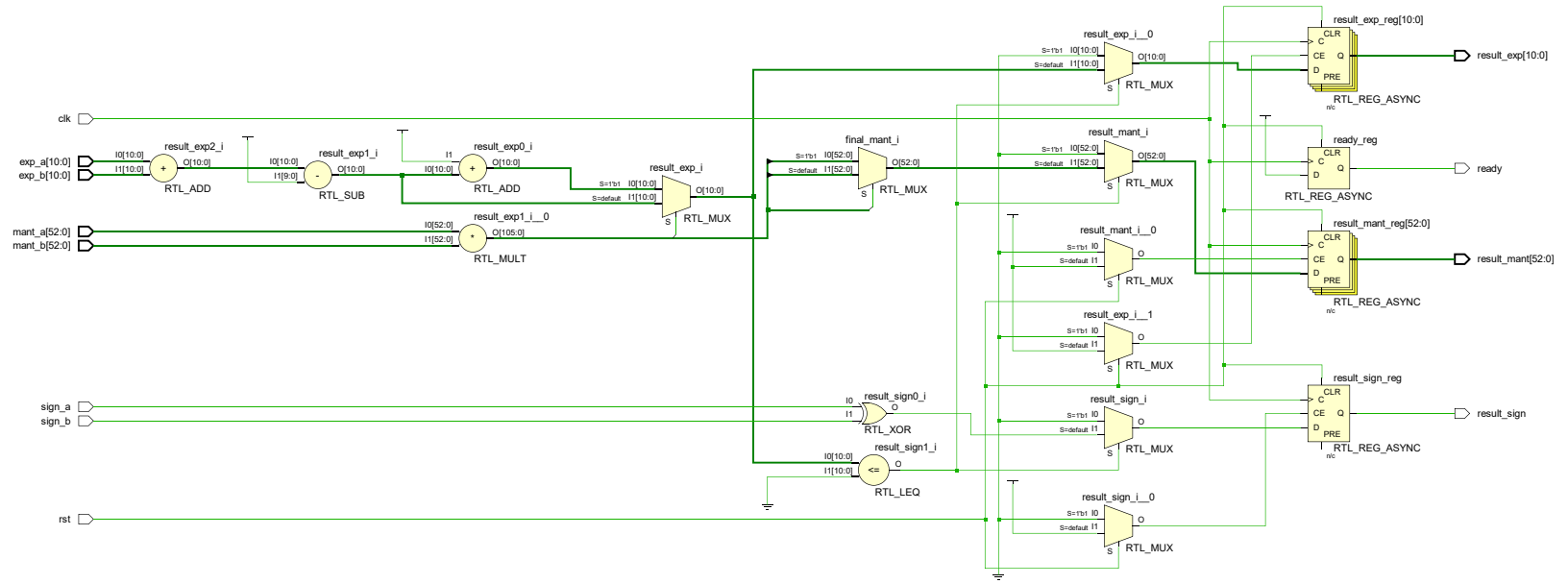
```
`timescale 1ns / 1ps

module fpu_div (
    input  wire        clk,
    input  wire        rst,
    input  wire        sign_a,
    input  wire [10:0] exp_a,
    input  wire [52:0] mant_a,    // 1.M
    input  wire        sign_b,
    input  wire [10:0] exp_b,
    input  wire [52:0] mant_b,    // 1.M
    output reg         result_sign,
    output reg  [10:0] result_exp,
    output reg  [52:0] result_mant,
    output reg         ready
);

    reg [105:0] dividend_ext;
    reg [52:0]  quotient;
    reg [10:0]  raw_exp;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            result_sign <= 0;
            result_exp  <= 0;
            result_mant <= 0;
            ready       <= 0;
        end else begin
            ready <= 0;

            // Sign
            result_sign = sign_a ^ sign_b;

            // Exponent
            raw_exp = exp_a - exp_b + 11'd1023;

            // Divide mantissas with precision
            dividend_ext = {mant_a, 53'b0};   // 106-bit numerator
            quotient     = dividend_ext / mant_b; // Result is 53-bit

            // Normalize quotient
            if (quotient[52]) begin
                result_mant = quotient;
                result_exp  = raw_exp;
            end else begin
                result_mant = quotient << 1;
```

```
            result_exp  = raw_exp - 1;
        end

        // Handle underflow
        if (result_exp <= 0) begin
            result_exp  = 0;
            result_mant = 0;
        end

        // Handle overflow
        if (result_exp >= 11'b11111111111) begin
            result_exp  = 11'b11111111111;
            result_mant = 0;
        end

        ready <= 1;
    end
  end
endmodule
```

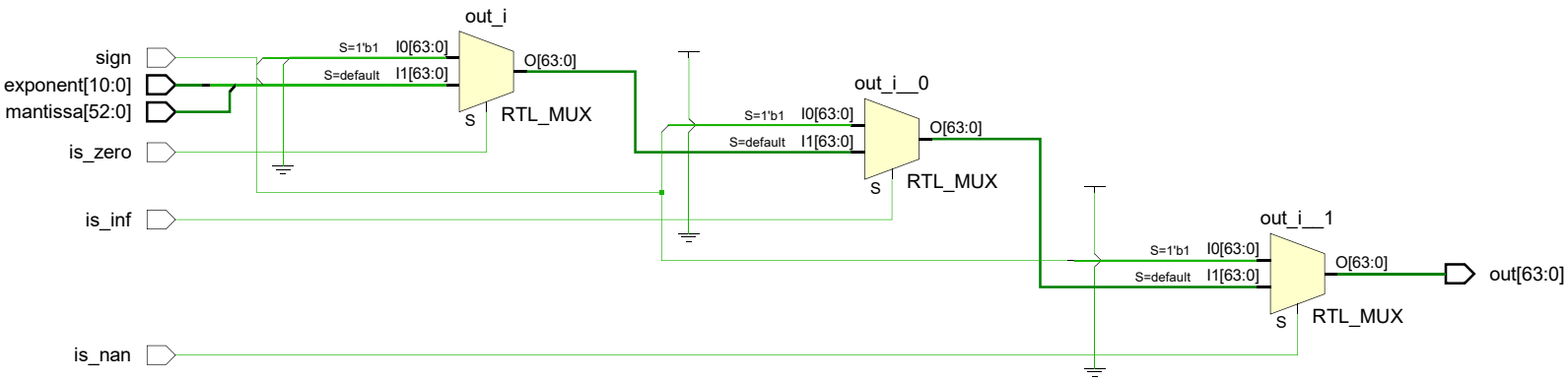

exp_a[10:0]
exp_b[10:0]
result_exp1_i
RTL_SUB
result_exp0_i
RTL_ADD
result_exp0_i__0
RTL_SUB
clk
result_exp_i
RTL_MUX
mant_a[52:0]
mant_b[52:0]
result_exp1_i__0
RTL_DIV
result_mant0_i
RTL_LSHIFT
result_mant_i
RTL_MUX
result_exp_i__0
RTL_MUX
result_exp1_i__1
RTL_LEQ
result_exp1_i__2
RTL_GEQ
result_mant_i__0
RTL_MUX
result_exp_i__1
RTL_MUX
result_mant_i__1
RTL_MUX
result_mant_i__2
RTL_MUX
result_exp_i__2
RTL_MUX
result_sign_i
RTL_MUX
result_exp_reg[10:0]
RTL_REG_ASYNC
result_exp[10:0]
result_mant_reg[52:0]
RTL_REG_ASYNC
result_mant[52:0]
ready_reg
RTL_REG_ASYNC
ready
result_sign_reg
RTL_REG_ASYNC
result_sign
rst
sign_a
sign_b
result_sign0_i
RTL_XOR

```
`timescale 1ns / 1ps

module fpu_mul (
    input  wire        clk,
    input  wire        rst,
    input  wire        sign_a,
    input  wire [10:0] exp_a,
    input  wire [52:0] mant_a,    // 1.M (includes implicit 1)
    input  wire        sign_b,
    input  wire [10:0] exp_b,
    input  wire [52:0] mant_b,    // 1.M (includes implicit 1)
    output reg         result_sign,
    output reg  [10:0] result_exp,
    output reg  [52:0] result_mant,
    output reg         ready
);

    // Internal registers
    reg [105:0] product;  // 53x53 = 106 bits
    reg [10:0]  raw_exp;
    reg [52:0]  final_mant;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            result_sign <= 0;
            result_exp  <= 0;
            result_mant <= 0;
            ready       <= 0;
        end else begin
            ready <= 0;

            // 1. Compute result sign
            result_sign = sign_a ^ sign_b;

            // 2. Compute raw exponent
            raw_exp = exp_a + exp_b - 11'd1023;

            // 3. Multiply mantissas
            product = mant_a * mant_b;  // 106-bit product

            // 4. Normalize result
            if (product[105]) begin
                final_mant = product[105:53];     // Already normalized
                result_exp = raw_exp + 1;
            end else begin
                final_mant = product[104:52];     // Needs normalization
```

```
            result_exp = raw_exp;
        end

        // 5. Handle underflow (flush to zero)
        if (result_exp <= 0) begin
            result_exp  = 0;
            result_mant = 0;
            result_sign = 0;
        end else begin
            result_mant = final_mant;
        end

        ready <= 1;
    end
  end
endmodule
```


clk
exp_a[10:0]
exp_b[10:0]
mant_a[52:0]
mant_b[52:0]
sign_a
sign_b
rst
result_exp2_i
RTL_ADD
result_exp1_i
RTL_SUB
result_exp0_i
RTL_ADD
result_exp_i
RTL_MUX
result_exp1_i__0
RTL_MULT
final_mant_i
RTL_MUX
result_sign0_i
RTL_XOR
result_sign1_i
RTL_LEQ
result_exp_i__0
RTL_MUX
result_mant_i
RTL_MUX
result_mant_i__0
RTL_MUX
result_exp_i__1
RTL_MUX
result_sign_i
RTL_MUX
result_sign_i__0
RTL_MUX
result_exp_reg[10:0]
RTL_REG_ASYNC
result_exp[10:0]
ready_reg
RTL_REG_ASYNC
ready
result_mant_reg[52:0]
RTL_REG_ASYNC
result_mant[52:0]
result_sign_reg
RTL_REG_ASYNC
result_sign

```
`timescale 1ns / 1ps

module fpu_pack (
    input  wire        sign,           // Final sign bit
    input  wire [10:0] exponent,       // Final exponent (after normalization)
    input  wire [52:0] mantissa,       // Final mantissa (with implicit 1)
    input  wire        is_zero,        // Input was zero
    input  wire        is_inf,         // Input was infinity
    input  wire        is_nan,         // Input was NaN
    output reg  [63:0] out             // 64-bit IEEE 754 packed output
);

    // Constants for special values
    localparam [10:0] EXP_INF  = 11'b11111111111;
    localparam [51:0] QNAN_PAYLOAD = 52'h0008000000000; // quiet NaN payload

    always @(*) begin
        if (is_nan) begin
            // NaN: Exponent = all 1s, mantissa ≠ 0 (quiet NaN format)
            out = {sign, EXP_INF, QNAN_PAYLOAD};
        end
        else if (is_inf) begin
            // Infinity: Exponent = all 1s, mantissa = 0
            out = {sign, EXP_INF, 52'b0};
        end
        else if (is_zero) begin
            // Zero: All bits zero except sign
            out = {sign, 11'b0, 52'b0};
        end
        else begin
            // Normalized number: pack result, drop implicit 1
            out = {sign, exponent, mantissa[51:0]};
        end
    end

endmodule
```

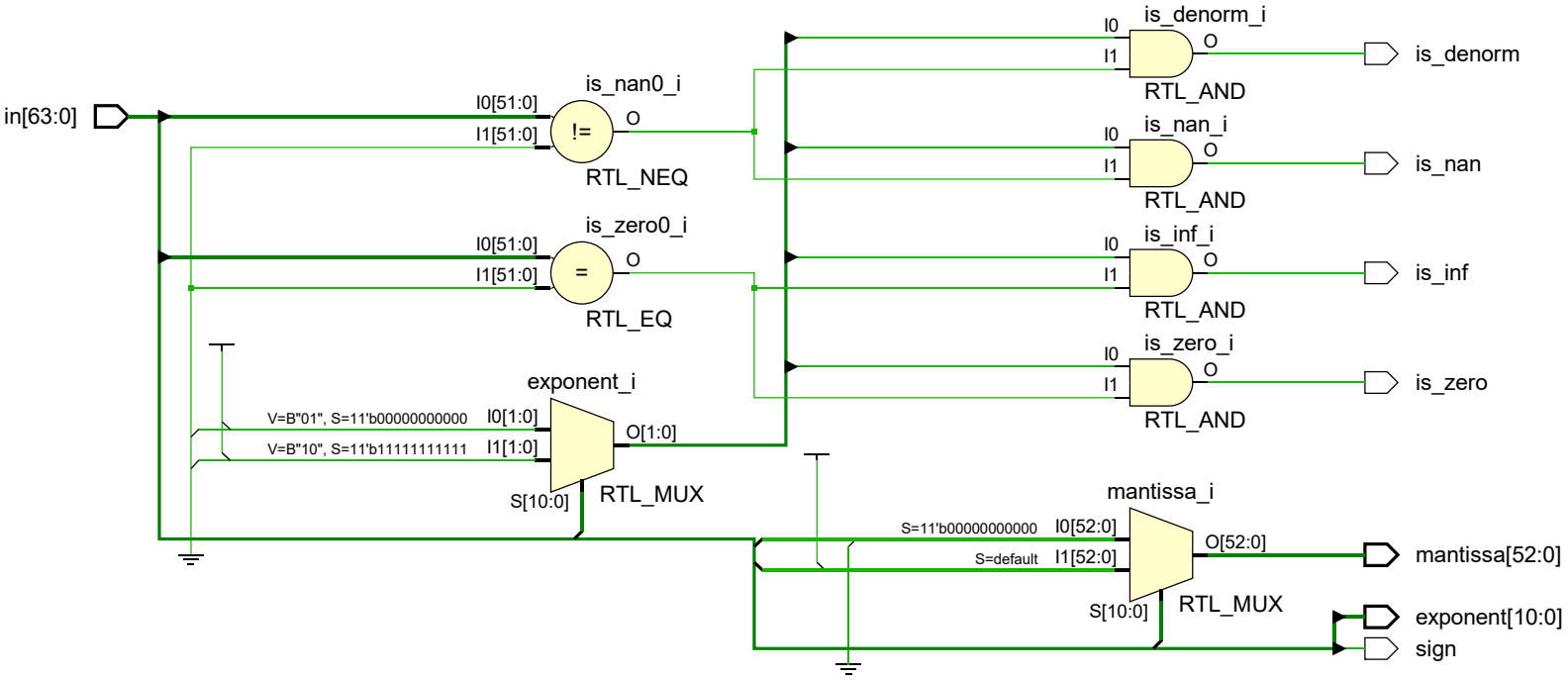

sign
exponent[10:0]
mantissa[52:0]
is_zero
is_inf
is_nan
out_i
S=1'b1
I0[63:0]
S=default
I1[63:0]
O[63:0]
S
RTL_MUX
out_i__0
S=1'b1
I0[63:0]
S=default
I1[63:0]
O[63:0]
S
RTL_MUX
out_i__1
S=1'b1
I0[63:0]
S=default
I1[63:0]
O[63:0]
S
RTL_MUX
out[63:0]

```
`timescale 1ns / 1ps

module fpu_64bit_tb;

    reg         clk;
    reg         rst;
    reg         start;
    reg  [1:0]  op;
    reg  [63:0] a, b;
    wire [63:0] result;
    wire        ready;

    // Instantiate the FPU
    fpu_64bit uut (
        .clk(clk),
        .rst(rst),
        .start(start),
        .op(op),
        .a(a),
        .b(b),
        .result(result),
        .ready(ready)
    );

    // Clock generation
    always #5 clk = ~clk;

    // Task to apply one test
    task test_op(input [1:0] test_op, input [63:0] in_a, input [63:0] in_b, input
[255:0] op_name);
        begin
            @(posedge clk);
            op    = test_op;
            a     = in_a;
            b     = in_b;
            start = 1;

            @(posedge clk);
            start = 0;

            wait (ready);
            $display("[%s] A: %h, B: %h => Result: %h", op_name, in_a, in_b,
result);
        end
    endtask
```

```
// IEEE 754 double precision numbers
// Format: Sign(1) + Exponent(11) + Mantissa(52)
// Examples:
localparam [63:0] A_1_5 = 64'h3FF8000000000000; // 1.5
localparam [63:0] B_2_0 = 64'h4000000000000000; // 2.0
localparam [63:0] C_3_5 = 64'h400C000000000000; // 3.5
localparam [63:0] D_0_5 = 64'h3FE0000000000000; // 0.5
localparam [63:0] E_0   = 64'h0000000000000000; // 0.0
localparam [63:0] F_INF = 64'h7FF0000000000000; // +INF
localparam [63:0] G_NAN = 64'h7FF8000000000000; // NaN

initial begin
    $display("==== FPU 64-bit Testbench Start ====");
    clk   = 0;
    rst   = 1;
    start = 0;
    op    = 2'b00;
    a     = 64'd0;
    b     = 64'd0;

    #20 rst = 0;

    // Add: 1.5 + 2.0 = 3.5
    test_op(2'b00, A_1_5, B_2_0, "ADD");

    // Sub: 3.5 - 2.0 = 1.5
    test_op(2'b01, C_3_5, B_2_0, "SUB");

    // Mul: 1.5 * 2.0 = 3.0
    test_op(2'b10, A_1_5, B_2_0, "MUL");

    // Div: 3.5 / 0.5 = 7.0
    test_op(2'b11, C_3_5, D_0_5, "DIV");

    // Edge case: Add 0 + 0 = 0
    test_op(2'b00, E_0, E_0, "ADD ZERO");

    // Edge case: INF * 2.0 = INF
    test_op(2'b10, F_INF, B_2_0, "MUL INF");

    // Edge case: NaN + 1.5 = NaN
    test_op(2'b00, G_NAN, A_1_5, "ADD NaN");

    $display("==== FPU 64-bit Testbench Done ====");
    #20 $finish;
end
```

```
endmodule
```

```
`timescale 1ns / 1ps

module fpu_unpack (
    input  wire [63:0] in,         // 64-bit IEEE 754 input
    output wire        sign,       // Sign bit
    output wire [10:0] exponent,   // Exponent field
    output wire [52:0] mantissa,   // Mantissa with implicit leading 1
    output wire        is_zero,    // Input is zero
    output wire        is_inf,     // Input is infinity
    output wire        is_nan,     // Input is NaN
    output wire        is_denorm   // Denormalized number
);

    // Extract fields
    assign sign     = in[63];
    assign exponent = in[62:52];
    wire [51:0] frac = in[51:0];

    // IEEE 754 exponent constants
    localparam [10:0] EXP_ZERO = 11'b00000000000;
    localparam [10:0] EXP_INF  = 11'b11111111111;

    // Classify input
    assign is_zero   = (exponent == EXP_ZERO) && (frac == 52'b0);
    assign is_inf    = (exponent == EXP_INF)  && (frac == 52'b0);
    assign is_nan    = (exponent == EXP_INF)  && (frac != 52'b0);
    assign is_denorm = (exponent == EXP_ZERO) && (frac != 52'b0);

    // Add implicit leading 1 for normalized numbers
    assign mantissa = (exponent == EXP_ZERO) ? {1'b0, frac} : {1'b1, frac};

endmodule
```


in[63:0]
is_nan0_i
I0[51:0]
I1[51:0]
!=
O
RTL_NEQ
is_zero0_i
I0[51:0]
I1[51:0]
=
O
RTL_EQ
exponent_i
V=B"01", S=11'b00000000000
V=B"10", S=11'b11111111111
I0[1:0]
I1[1:0]
O[1:0]
S[10:0]
RTL_MUX
is_denorm_i
I0
I1
O
RTL_AND
is_denorm
is_nan_i
I0
I1
O
RTL_AND
is_nan
is_inf_i
I0
I1
O
RTL_AND
is_inf
is_zero_i
I0
I1
O
RTL_AND
is_zero
mantissa_i
S=11'b00000000000
S=default
I0[52:0]
I1[52:0]
O[52:0]
S[10:0]
RTL_MUX
mantissa[52:0]
exponent[10:0]
sign

```
`timescale 1ns / 1ps

module fpu_64bit (
    input  wire        clk,
    input  wire        rst,
    input  wire        start,
    input  wire [1:0]  op,         // 00: add, 01: sub, 10: mul, 11: div
    input  wire [63:0] a,
    input  wire [63:0] b,
    output wire [63:0] result,
    output reg         ready
);

    // Internal signals from unpack
    wire        sign_a, sign_b;
    wire [10:0] exp_a, exp_b;
    wire [52:0] mant_a, mant_b;
    wire        is_nan_a, is_nan_b, is_inf_a, is_inf_b, is_zero_a, is_zero_b;

    // Unpack operands
    fpu_unpack unpack_a (
        .in(a),
        .sign(sign_a),
        .exponent(exp_a),
        .mantissa(mant_a),
        .is_nan(is_nan_a),
        .is_inf(is_inf_a),
        .is_zero(is_zero_a)
    );

    fpu_unpack unpack_b (
        .in(b),
        .sign(sign_b),
        .exponent(exp_b),
        .mantissa(mant_b),
        .is_nan(is_nan_b),
        .is_inf(is_inf_b),
        .is_zero(is_zero_b)
    );

    // Determine global exceptions
    wire is_nan  = is_nan_a | is_nan_b;
    wire is_zero = is_zero_a & is_zero_b;
    wire is_inf  = (op != 2'b11) ? (is_inf_a | is_inf_b) : 1'b0;

    // Outputs from submodules
```

```
wire        addsub_sign, mul_sign, div_sign;
wire [10:0] addsub_exp,  mul_exp,  div_exp;
wire [52:0] addsub_mant, mul_mant, div_mant;
wire        addsub_ready, mul_ready, div_ready;

// ADD/SUB
fpu_addsub addsub (
    .clk(clk),
    .rst(rst),
    .op(op[0]), // 0: add, 1: sub
    .sign_a(sign_a),
    .exp_a(exp_a),
    .mant_a(mant_a),
    .sign_b(sign_b),
    .exp_b(exp_b),
    .mant_b(mant_b),
    .result_sign(addsub_sign),
    .result_exp(addsub_exp),
    .result_mant(addsub_mant),
    .ready(addsub_ready)
);

// MUL
fpu_mul mul (
    .clk(clk),
    .rst(rst),
    .sign_a(sign_a),
    .exp_a(exp_a),
    .mant_a(mant_a),
    .sign_b(sign_b),
    .exp_b(exp_b),
    .mant_b(mant_b),
    .result_sign(mul_sign),
    .result_exp(mul_exp),
    .result_mant(mul_mant),
    .ready(mul_ready)
);

// DIV
fpu_div div (
    .clk(clk),
    .rst(rst),
    .sign_a(sign_a),
    .exp_a(exp_a),
    .mant_a(mant_a),
    .sign_b(sign_b),
```

```
    .exp_b(exp_b),
    .mant_b(mant_b),
    .result_sign(div_sign),
    .result_exp(div_exp),
    .result_mant(div_mant),
    .ready(div_ready)
);

// Output selection
reg        final_sign;
reg [10:0] final_exp;
reg [52:0] final_mant;

always @(*) begin
    case (op)
        2'b00, 2'b01: begin
            final_sign = addsub_sign;
            final_exp  = addsub_exp;
            final_mant = addsub_mant;
            ready      = addsub_ready;
        end
        2'b10: begin
            final_sign = mul_sign;
            final_exp  = mul_exp;
            final_mant = mul_mant;
            ready      = mul_ready;
        end
        2'b11: begin
            final_sign = div_sign;
            final_exp  = div_exp;
            final_mant = div_mant;
            ready      = div_ready;
        end
        default: begin
            final_sign = 1'b0;
            final_exp  = 11'd0;
            final_mant = 53'd0;
            ready      = 1'b0;
        end
    endcase
end

// Final pack
fpu_pack pack (
    .sign(final_sign),
    .exponent(final_exp),
```

```
        .mantissa(final_mant),
        .is_zero(is_zero),
        .is_inf(is_inf),
        .is_nan(is_nan),
        .out(result)
    );

endmodule
```