```
--------------------------------------------------------------------------------
-- LOG(X) ENTITY --
--------------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
USE IEEE.std_logic_unsigned.ALL;

-- Fixed point 32 bit representation
-- 00000000000000000.00000000000000
-- |_____17bit_____|.|____15bit____|
--     integer          fractional

-- Input 16 bit signal is a fixed point number of the form:
-- 00.00000000000000
-- 2 bits before decimal: Integral part
-- 14 bits after decimal: Fractional part

ENTITY log IS
    PORT (
        A : IN std_logic_vector(15 DOWNTO 0);
        S : OUT std_logic_vector(31 DOWNTO 0)
    );
END log;

ARCHITECTURE arch OF log IS

    COMPONENT multiplier IS
        PORT (
            A : IN std_logic_vector(31 DOWNTO 0);
            B : IN std_logic_vector(31 DOWNTO 0);
            S : OUT std_logic_vector(31 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT adder IS
        PORT (
            A : IN std_logic_vector(31 DOWNTO 0);
            B : IN std_logic_vector(31 DOWNTO 0);
            S : OUT std_logic_vector(31 DOWNTO 0);
            carry : OUT std_logic
        );
    END COMPONENT;

    --Inputs
    SIGNAL input_A : std_logic_vector(31 DOWNTO 0) := (OTHERS => '0');
    SIGNAL input_sign : std_logic_vector(31 DOWNTO 0);
```

```
    SIGNAL pseudo_input : std_logic_vector(31 DOWNTO 0) := (OTHERS => '0');

    --Constants
    CONSTANT one : std_logic_vector(31 DOWNTO 0) := "00000000000000000000000000000001"; -- for adding to one's complement
    CONSTANT minus_one : std_logic_vector(31 DOWNTO 0) := "11111111111111110000000000000000"; -- two's complement of fixed point -1 (for x - 1)

    CONSTANT half : std_logic_vector(31 DOWNTO 0) := "00000000000000001000000000000000"; -- 1/2 = 0.5
    CONSTANT third : std_logic_vector(31 DOWNTO 0) := "00000000000000000101010101010101"; -- 1/3 = 0.3333
    CONSTANT fourth : std_logic_vector(31 DOWNTO 0) := "00000000000000000100000000000000"; -- 1/4 = 0.25
    CONSTANT fifth : std_logic_vector(31 DOWNTO 0) := "00000000000000000011001100110011"; -- 1/5 = 0.2
    CONSTANT sixth : std_logic_vector(31 DOWNTO 0) := "00000000000000000010101010101011"; -- 1/6 = 0.1667

    --Addition signals
    SIGNAL add_result_1 : std_logic_vector(31 DOWNTO 0);
    SIGNAL add_result_2 : std_logic_vector(31 DOWNTO 0);
    SIGNAL add_result_3 : std_logic_vector(31 DOWNTO 0);
    SIGNAL add_result_4 : std_logic_vector(31 DOWNTO 0);
    SIGNAL add_result_5 : std_logic_vector(31 DOWNTO 0);

    --Power signals
    SIGNAL power_2 : std_logic_vector(31 DOWNTO 0);
    SIGNAL power_3 : std_logic_vector(31 DOWNTO 0);
    SIGNAL power_4 : std_logic_vector(31 DOWNTO 0);
    SIGNAL power_5 : std_logic_vector(31 DOWNTO 0);
    SIGNAL power_6 : std_logic_vector(31 DOWNTO 0);

    --Multiplication signals
    SIGNAL mul_result_1 : std_logic_vector(31 DOWNTO 0);
    SIGNAL mul_result_2 : std_logic_vector(31 DOWNTO 0);
    SIGNAL mul_result_3 : std_logic_vector(31 DOWNTO 0);
    SIGNAL mul_result_4 : std_logic_vector(31 DOWNTO 0);
    SIGNAL mul_result_5 : std_logic_vector(31 DOWNTO 0);

    --signals used to store the sign bit/carry bit from the adder
    SIGNAL dummy_1 : std_logic;
    SIGNAL dummy_2 : std_logic;
    SIGNAL dummy_3 : std_logic;
    SIGNAL dummy_4 : std_logic;
    SIGNAL dummy_5 : std_logic;
    SIGNAL dummy_6 : std_logic;
    SIGNAL dummy_7 : std_logic;
```

```
    SIGNAL dummy_8 : std_logic;
    SIGNAL sign_MSB : std_logic;

    --One's complement signals
    SIGNAL ones_complement : std_logic_vector(31 DOWNTO 0);

    --Result signals
    SIGNAL above_one : std_logic_vector(31 DOWNTO 0); --
stores the answers for when (x - 1) is positive, i.e. x > 1
    SIGNAL below_one : std_logic_vector(31 DOWNTO 0); --
stores the answers for when (x - 1) is negative, i.e. x < 1
    SIGNAL result : std_logic_vector(32 DOWNTO 0);
    SIGNAL fp_result : std_logic_vector(31 DOWNTO 0);

    --Function to convert a signed binary 32-bit fixed pt number to 32-
bit floating pt sign-magnitude format
    FUNCTION SIGNED_TO_FP(input_number : unsigned; carry : std_logic) RETURN s
td_logic_vector IS

        VARIABLE i : INTEGER RANGE 0 TO input_number'left + 1;
        VARIABLE j : INTEGER RANGE 0 TO 255;
        VARIABLE floating_point : std_logic_vector(31 DOWNTO 0); --returned FP
        VARIABLE exp : INTEGER RANGE -1024 TO 1024; --exponent
        VARIABLE mantissa : unsigned(input_number'length DOWNTO 0); --
mantissa + leading bit

    BEGIN

        IF input_number = (31 DOWNTO 0 => '0') THEN
            floating_point := (OTHERS => '0');
        ELSE
            mantissa := '0' & unsigned(input_number); --
we use the mag of input_number to create a mantissa

            --start with biased exp equiv to 2**(LENGTH-
1), so m becomes the mantissa, m.mmmmm...
            --i.e. mag(input_number) = 2**(exp-127) * m.m m m m m....
            exp := 127 + mantissa'length - 1;
            exp := exp - 15; -- -
15 is to account for the fixed point decimal parts

            --
normalize m as the mantissa with one bit in front of the decimal point
            FOR i IN 0 TO mantissa'left LOOP
                IF mantissa(mantissa'left) = '1' THEN
                    j := i;
                    EXIT;
                ELSE
```

```
                    mantissa := mantissa(mantissa'left - 1 DOWNTO 0) & '0';
                END IF;
            END LOOP;

            exp := exp - j;

            --assigning the exponent bits
            floating_point(30 DOWNTO 23) := std_logic_vector(TO_UNSIGNED(exp,
8));

            --
Make sure we have enough bits for a normalized full mantissa (23)
            --Remove the mantissa leading 1
            IF mantissa'length < 24 THEN -
- <24 bits, bottom bits set to 0, and drop the leading 1
                floating_point(23 - mantissa'length DOWNTO 0) := (OTHERS => '0
');
                floating_point(22 DOWNTO 24 - mantissa'length) := std_logic_ve
ctor(mantissa(mantissa'length - 2 DOWNTO 0));
            ELSE --
if >= 24, drop leading 1 and take next 23 bits for fp (kind of like rounding o
ff)
                floating_point(22 DOWNTO 0) := std_logic_vector(mantissa(manti
ssa'length - 2 DOWNTO mantissa'length - 24));
            END IF;

            --assigning the sign bit in accordance with two's coplement
            IF carry = '1' THEN
                floating_point(31) := '1';
            ELSE
                floating_point(31) := '0';
            END IF;
        END IF;
        RETURN floating_point;
    END FUNCTION SIGNED_TO_FP;

    --Function to finalise the input (+ve or -
ve) depending on the two's complement subtraction
    FUNCTION FINALISE_INPUT(input : std_logic_vector; carry : std_logic) RETUR
N std_logic_vector IS
        VARIABLE result : std_logic_vector(31 DOWNTO 0);
    BEGIN
        IF carry = '1' THEN
            --input: x > 1
            result := input;
        ELSE
            --input: x < 1
            result := NOT(input) + 1;
```

```
        END IF;
        RETURN std_logic_vector(result);
    END FUNCTION FINALISE_INPUT;

    --Function to finalise the output (+ve or -
ve) depending on the two's complement subtraction
    FUNCTION FINALISE_OUTPUT(above_one : std_logic_vector; below_one : std_log
ic_vector; carry : std_logic) RETURN std_logic_vector IS
        VARIABLE result : std_logic_vector(31 DOWNTO 0);
    BEGIN
        IF carry = '1' THEN
            --input: x > 1
            result := above_one;
        ELSE
            --input: x < 1
            result := below_one;
        END IF;
        RETURN std_logic_vector(result);
    END FUNCTION FINALISE_OUTPUT;

BEGIN

    --doing x - 1 to the given input x
    pseudo_input(16 DOWNTO 1) <= A(15 DOWNTO 0); --stores given x
    ADD_0 : adder PORT MAP(pseudo_input, minus_one, input_sign, sign_MSB); -
- x - 1
    input_A <= FINALISE_INPUT(input_sign, sign_MSB); --stores x - 1

    SQUARE : multiplier PORT MAP(input_A, input_A, power_2); -- (x - 1)^2
    CUBE : multiplier PORT MAP(power_2, input_A, power_3); -- (x - 1)^3
    FOURTH_POWER : multiplier PORT MAP(power_3, input_A, power_4); -
- (x - 1)^4
    FIFTH_POWER : multiplier PORT MAP(power_4, input_A, power_5); -- (x - 1)^5
    SIXTH_POWER : multiplier PORT MAP(power_5, input_A, power_6); -- (x - 1)^6

    MULTIPLY_1 : multiplier PORT MAP(power_2, half, mul_result_1); -
- (x - 1)^2/2
    MULTIPLY_2 : multiplier PORT MAP(power_3, third, mul_result_2); -
- (x - 1)^3/3
    MULTIPLY_3 : multiplier PORT MAP(power_4, fourth, mul_result_3); -
- (x - 1)^4/4
    MULTIPLY_4 : multiplier PORT MAP(power_5, fifth, mul_result_4); -
- (x - 1)^5/5
    MULTIPLY_5 : multiplier PORT MAP(power_6, sixth, mul_result_5); -
- (x - 1)^6/6

    ADD_1 : adder PORT MAP(input_A, mul_result_2, add_result_1, dummy_1); -
- (x - 1) + (x - 1)^3/3
```

```
    ADD_2 : adder PORT MAP(add_result_1, mul_result_4, add_result_2, dummy_2);
 -- ((x - 1) + (x - 1)^3/3) + (x - 1)^5/5

    ADD_3 : adder PORT MAP(mul_result_1, mul_result_3, add_result_3, dummy_3);
 -- (x - 1)^2/2 + (x - 1)^4/4
    ADD_4 : adder PORT MAP(add_result_3, mul_result_5, add_result_4, dummy_4);
 -- ((x - 1)^2/2 + (x - 1)^4/4) + (x - 1)^6/6

    ones_complement <= NOT(add_result_4);

    ADD_5 : adder PORT MAP(one, ones_complement, add_result_5, dummy_5); -
- 2s complement of ((x - 1)^2/2 + (x - 1)^4/4 + (x - 1)^6/6) = -
((x - 1)^2/2 + (x - 1)^4/4 + (x - 1)^6/6)

    --for cases when 1 < x < 2
    ADD_6 : adder PORT MAP(add_result_2, add_result_5, above_one, dummy_6); -
- ((x - 1) + (x - 1)^3/3 + (x - 1)^5/5) - ((x - 1)^2/2 + (x - 1)^4/4 + (x - 1)
^6/6)

    --for cases when 0 < x < 1
    ADD_7 : adder PORT MAP(add_result_2, add_result_4, below_one, dummy_7); -
- ((x - 1) + (x - 1)^3/3 + (x - 1)^5/5) + ((x - 1)^2/2 + (x - 1)^4/4 + (x - 1)
^6/6)

    --RESULT
    result(32) <= NOT(sign_MSB); --stores the sign of the result

    --picks result depending on whether (x - 1) is +ve or -ve
    result(31 DOWNTO 0) <= FINALISE_OUTPUT(above_one, below_one, sign_MSB);

    --converting the result to floating point for making it easily readable
    fp_result <= SIGNED_TO_FP(unsigned(result(31 DOWNTO 0)), result(32));
    S <= fp_result;

END arch;

--------------------------------------------------------------------------------
-- MULTIPLIER ENTITY --
--------------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

-- Fixed point 32 bit representation
-- 0000000000000000.00000000000000
-- |_____17bit_____|.|____15bit____|
--     integer          fractional
```

```
ENTITY multiplier IS
    PORT (
        A : IN std_logic_vector(31 DOWNTO 0);
        B : IN std_logic_vector(31 DOWNTO 0);
        S : OUT std_logic_vector(31 DOWNTO 0)
    );
END multiplier;

ARCHITECTURE Behavioral OF multiplier IS
    SIGNAL temp : std_logic_vector(63 DOWNTO 0);

BEGIN

    temp <= std_logic_vector(UNSIGNED(A) * UNSIGNED(B));
    S(31 DOWNTO 0) <= temp(46 DOWNTO 15);

END Behavioral;

--------------------------------------------------------------------------------
-- ADDER ENTITY --
--------------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

-- Fixed point 32 bit representation
-- 00000000000000000.000000000000000
-- |_____17bit_____|.|____15bit____|
--     integer          fractional

ENTITY adder IS
    PORT (
        A : IN std_logic_vector(31 DOWNTO 0); -- 32 bit Addend
        B : IN std_logic_vector(31 DOWNTO 0); -- 32 bit Augend
        S : OUT std_logic_vector(31 DOWNTO 0); -- real result
        carry : OUT std_logic -- 1 bit carry: required for two's complement
    );
END adder;

ARCHITECTURE Behavioral OF adder IS
    SIGNAL number_1 : std_logic_vector(32 DOWNTO 0);
    SIGNAL number_2 : std_logic_vector(32 DOWNTO 0);
    SIGNAL result : std_logic_vector(32 DOWNTO 0);

BEGIN
```

```
    number_1 <= '0' & A;
    number_2 <= '0' & B;

    result <= std_logic_vector(UNSIGNED(number_1) + UNSIGNED(number_2));

    S <= result(31 DOWNTO 0);
    carry <= result(32);

END Behavioral;
```