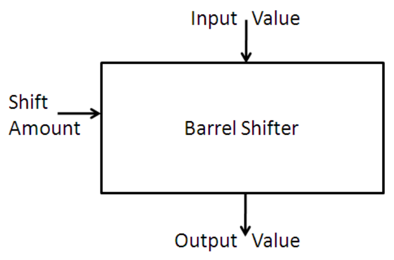
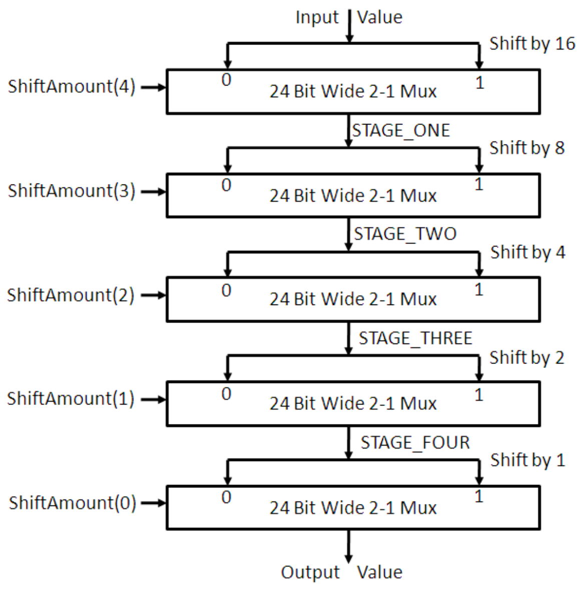
Use of Barrel Shifter for Alignment Network and Post Normalization

For the first assignment you were asked to implement an IEEE Floating Point Adder for normalized numbers that was combinational in nature – that is, no clocks are utilized for any of the functions in the system. In particular, there are two different functions that are serial in nature, or that we sometimes think of in serial terms. The first, and easiest, is the alignment network that aligns the smaller mantissa with respect to the larger one. The second is the post normalization system, which has a similar function. In both of these cases, a value – represented as bits – needs to be shifted by a specific number of bit positions. Consider each of these applications in turn.

The first situation under consideration is the alignment of the mantissas for the operation to be performed. This is accomplished by shifting the mantissa of the smaller number to the right with respect to the bits of the larger number. The number of bit positions is determined by the difference in the two exponents. This shift is easily accomplished by a series of two-to-one muxes that are arranged in what is called a barrel shifter. First, the block diagram of the barrel shifter itself:



The input value is the 24 bit mantissa. The Shift Amount is the difference between exponents, which is the number of bits to shift the Input Amount to the right (with leading zeros on the left) when it is presented to the Output Value. The barrel shift approach to this problem is to utilize a series of two-to-one muxes connected as follows:



The circuit can be described as follows. The input value is presented to the first mux, and ShiftAmount(4) determines whether a shift is applied or not. If ShiftAmount(4) is zero, then the Input Value is passed to STAGE_ONE without any shift. If, however, ShiftAmount(4) is one, then Input Value is shifted to the right by 16 bit positions (the vacant positions are filled with zeros) and then presented to STAGE_ONE. The cost of this operation is 2 gate delays. Then the same method is implanted between STAGE_ONE and STAGE_TWO. That is, ShiftAmount(3) is used to determine whether an eight-bit shift is applied. If ShiftAmount(3) is zero, no shift is utilized. If ShiftAmount(3) is one, then the bits of STAGE_ONE are shifted to the right by 8 bit positions (vacant positions filled with zeros) and presented to STAGE_TWO. Again, the cost of this stage is 2 gate delays. The remaining three stages operate in exactly the same fashion. The net result is that it takes 10 gate delays to shift Input Value any number of bit positions (0 to 23; note that if the value was represented by 31 bits, this would also be possible).

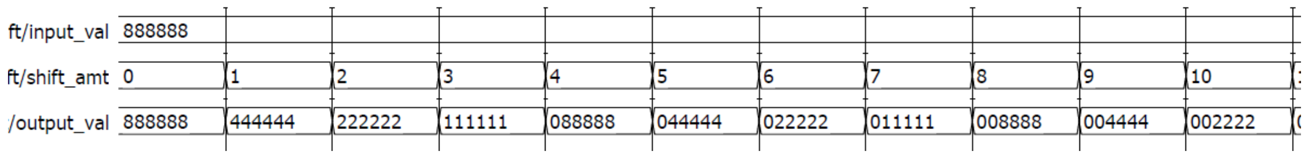
VHDL implementation of the alignment barrel shifter is very straightforward. Consider a system created for this purpose, with the names of buses as shown in the diagram above. The value to align is provided in a bus called INPUT_VAL, and how many places to shift is in SHIFT_AMT. In the declaration area of the architecture we declare the buses which form each stage, and also the final value. So, the declaration area contains:

```
    signal INPUT_VAL  : STD_LOGIC_VECTOR ( 23 downto 0 );
    signal OUTPUT_VAL   : STD_LOGIC_VECTOR ( 23 downto 0 );
    signal SHIFT_AMT    : STD_LOGIC_VECTOR (  4 downto 0 );

    signal STAGE_ONE    : STD_LOGIC_VECTOR ( 23 downto 0 );
    signal STAGE_TWO    : STD_LOGIC_VECTOR ( 23 downto 0 );
    signal STAGE_THREE  : STD_LOGIC_VECTOR ( 23 downto 0 );
    signal STAGE_FOUR   : STD_LOGIC_VECTOR ( 23 downto 0 );
```

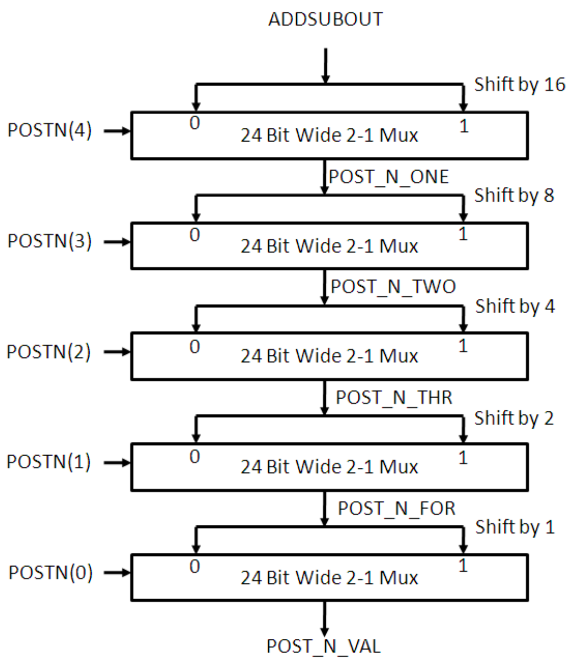
The work of the alignment system is then provided in five 2-to-1 mux statements:

```
    STAGE_ONE <= INPUT_VAL when SHIFT_AMT(4) = '0' else
                 X"0000" & INPUT_VAL ( 23 downto 16);
    STAGE_TWO <= STAGE_ONE when SHIFT_AMT(3) = '0' else
                 X"00" & STAGE_ONE ( 23 downto 8 );
    STAGE_THREE <= STAGE_TWO when SHIFT_AMT(2) = '0' else
                 X"0" & STAGE_TWO ( 23 downto 4 );
    STAGE_FOUR <= STAGE_THREE when SHIFT_AMT(1) = '0' else
                 "00" & STAGE_THREE ( 23 downto 2 );
    OUTPUT_VAL <= STAGE_FOUR when SHIFT_AMT(0) = '0' else
                 '0' & STAGE_FOUR ( 23 downto 1 );
```

Simulation of the system results in the waveform shown below (only input, output, and shift amount values are shown). Note that the input value is always 0x888888, and shift amounts from zero to ten are shown. The output value is correctly shifted to the right for each different value of the shift amount. If the simulation were continued to twenty four, nothing would be left in the output value, since all bits would have been shifted out.



The post normalization activity is exactly the same – but in reverse. That is, we know that a shift may be required, but the shift is to the left, not to the right. The other problem with post normalization is that the information about how far to shift is determined by the data, not by an external value as was the case with the alignment network. However, it turns out that the determination of how much to shift can be incorporated with each of the five stages of the post-normalizer in a straightforward fashion. Consider the problem of the first stage – shifting by 16 bit positions or providing no shift. When the data presents itself, the following statement describes the function of the first stage: if all of the 16 MSBs of the input value are zero, the input value should be shifted left by 16 bits. On the other hand, if there is a non-zero value in the 16 MSBs of the input value, no shift should be applied, and the value used for the second stage is just a copy of the value used for the first stage. The block diagram of the post-normalizer is shown below:



The block diagram for the post-normalizer looks identical to that of the alignment network. But, as noted above, there are two distinct differences. First, the alignment network shifts input values to the right, while the post-normalizer shifts values to the left. And second, the number of positions to shift is presented externally for the alignment network, while the post-normalizer generates its own number of positions to shift. In the diagram above, POSTN(4) is derived from the MSBs of ADDSUBOUT. POSTN(3) is derived from the MSBs of POST_N_ONE. This pattern continues throughout the system, so that at the end of the process the bits of POSTN form a binary number that is an indication of the number of bit positions that ADDSUBOUT has been shifted to arrive at POST_N_VAL. This number can then be utilized in the adjustment of the exponent presented as part of the answer.

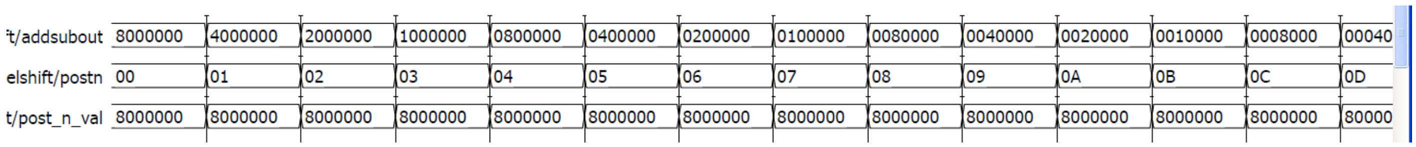
The VHDL for the post-normalizer is similar to the alignment network. In the declaration area of the architecture, signals involved in the post-normalizer are declared:

```
signal ADDSUBOUT  : STD_LOGIC_VECTOR ( 27 downto 0 );
signal POST_N_ONE : STD_LOGIC_VECTOR ( 27 downto 0 );
signal POST_N_TWO : STD_LOGIC_VECTOR ( 27 downto 0 );
signal POST_N_THR : STD_LOGIC_VECTOR ( 27 downto 0 );
signal POST_N_FOR : STD_LOGIC_VECTOR ( 27 downto 0 );
signal POST_N_VAL : STD_LOGIC_VECTOR ( 27 downto 0 );
signal POSTN      : STD_LOGIC_VECTOR (  4 downto 0 );
constant ZEROS    : STD_LOGIC_VECTOR ( 31 downto 0 ) := X"0000_0000";
```

The constant ZEROS is used to provide '0' at the appropriate bit positions. The work statements are fairly simple:

```
POSTN(4) <= '1' when ADDSUBOUT(27 downto 12) = X"0000" else '0';
POST_N_ONE <= ADDSUBOUT when POSTN(4) = '0' else
              ADDSUBOUT ( 11 downto 0 ) & ZEROS(15 downto 0 );

POSTN(3) <= '1' when POST_N_ONE ( 27 downto 20 ) = X"00" else '0';
POST_N_TWO <= POST_N_ONE when POSTN(3) = '0' else
              POST_N_ONE ( 19 downto 0 ) & ZEROS ( 7 downto 0 );

POSTN(2) <= '1' when POST_N_TWO ( 27 downto 24 ) = "0000" else '0';
POST_N_THR <= POST_N_TWO when POSTN(2) = '0' else
              POST_N_TWO ( 23 downto 0 ) & ZEROS ( 3 downto 0 );

POSTN(1) <= '1' when POST_N_THR ( 27 downto 26 ) = "00" else '0';
POST_N_FOR <= POST_N_THR when POSTN(1) = '0' else
              POST_N_THR ( 25 downto 0 ) & ZEROS ( 1 downto 0 );

POSTN(0) <= '1' when POST_N_FOR ( 27 ) = '0' else '0';
POST_N_VAL <= POST_N_FOR when POSTN(0) = '0' else
              POST_N_FOR ( 26 downto 0 ) & ZEROS ( 0 );
```

The simulation of this system shows the correct behavior of the system:



```
t/addsubout  8000000 4000000 2000000 1000000 0800000 0400000 0200000 0100000 0080000 0040000 0020000 0010000 0008000 00040
elshift/postn 00     01      02      03      04      05      06      07      08      09      0A      0B      0C      0D
t/post_n_val 8000000 8000000 8000000 8000000 8000000 8000000 8000000 8000000 8000000 8000000 8000000 8000000 8000000 80000
```

In this case, the output value (POST_N_VAL) is always 0x8000000, while the input value (ADDSUBOUT) has a single bit that moves from bit position to bit position. And the amount shifted is correctly reported in POSTN. This is the behavior needed for the post normalization system, since the single bit reported in POST_N_VAL will be the hidden bit of the answer, and the other elements of the floating point addition will work together to get the right answer.