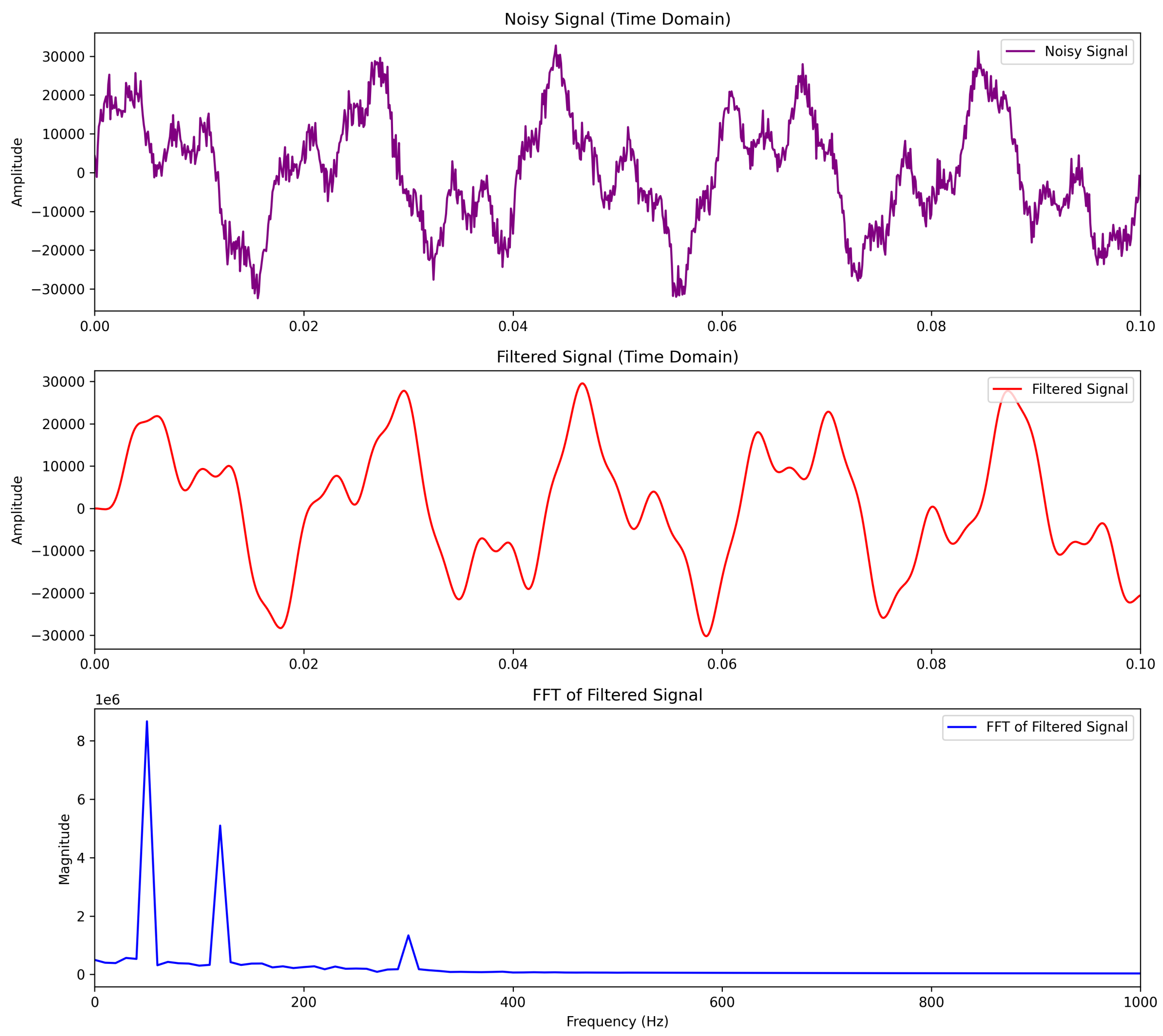
FIR Filter Assignment - Documentation

Sergio Brito
UALg ISE DEE

March 6, 2025

Figure 2: Noisy vs. Filtered Signal & FFT of Filtered Signal



Contents

1 Introduction & Overview

Digital FIR filters are integral components in modern digital signal processing systems. This document provides a comprehensive guide for implementing a digital FIR filter in VHDL, emphasizing both theoretical concepts and practical design techniques.

1.1 Purpose of the Assignment

- Develop an understanding of digital filtering principles and their applications.
- Gain practical experience in designing and simulating FIR filters using VHDL.
- Reinforce concepts in binary arithmetic, including operations with both positive and negative numbers.

1.2 Key Concepts Addressed

- **Signals:** An overview of signals, their characteristics, and their role in digital systems.
- **Digital Signal Processing (DSP):** A concise introduction to DSP, highlighting relevant concepts such as noise and signal manipulation.
- **Binary Arithmetic:** A review of binary number representations and arithmetic operations, with emphasis on multiplication involving signed numbers.

The sections that follow will systematically cover each topic, offering theoretical background alongside practical examples to ensure a robust understanding of FIR filter implementation in VHDL.

2 Digital vs. Analog Filtering

Filtering is a fundamental process in signal processing, employed in both analog and digital domains. This section provides an overview of analog and digital filters, emphasizing their operational principles, advantages, and applications.

2.1 Analog Filtering

Analog filters process continuous-time signals using electronic components. Key aspects include:

- **Operation:** Manipulation of the amplitude and phase of continuous signals through circuits comprised of resistors, capacitors, and inductors.
- **Design Considerations:** The performance of analog filters can be affected by component tolerances, temperature variations, and other environmental factors.
- **Applications:** Typically used in audio processing, communications, and control systems where real-time, continuous signal manipulation is required.

2.2 Digital Filtering

Digital filters operate on discrete-time signals obtained via sampling and quantization. This section examines the underlying concepts and advantages of digital filtering in greater detail.

2.2.1 Fundamental Concepts

- **Sampling and Quantization:** Continuous-time signals must be converted into a digital form for processing. This conversion is achieved through two key steps:
  - **Sampling:** The process involves measuring the continuous signal at uniform time intervals, known as the sampling period (T). The reciprocal of the sampling period, the sampling frequency (f_s), is critical—according to the Nyquist-Shannon theorem, f_s must be at least twice the highest frequency present in the signal to avoid aliasing. Aliasing occurs when higher frequency components are misrepresented as lower frequencies, leading to distortion.
  - **Quantization:** After sampling, each measured amplitude is approximated to the nearest value from a finite set of levels determined by the resolution (typically expressed in bits) of the analog-to-digital converter (ADC). This mapping introduces a quantization error, which is essentially noise added to the signal. The precision of quantization affects the signal-to-noise ratio and overall fidelity of the digital representation.
- **Discrete-Time Signals:** Once a signal is sampled and quantized, it is represented as a sequence of numbers, each corresponding to the amplitude at a specific sampling instant. These sequences, usually denoted as $x[n]$, where n is an integer index, are the basis of digital signal processing. The discrete-time representation allows for:
  - **Numerical Manipulation:** Digital algorithms can process these sequences using operations like addition, multiplication, and convolution.
  - **Analysis of Signal Properties:** Properties such as periodicity, stability, and causality are defined for discrete-time signals. For example, understanding the frequency content of $x[n]$ through the discrete Fourier transform (DFT) is essential in filter design.

- **Convolution:** Convolution is a mathematical operation that combines the input signal with the filter's impulse response to produce the output signal. In the discrete-time domain, convolution is defined as:

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]\, h[n-k]$$

where $x[n]$ is the input signal and $h[n]$ is the impulse response of the filter. In practical FIR filter implementations, the sum is finite, which simplifies computations and guarantees stability.

2.2.2 Types of Digital Filters

Digital filters are generally classified into two main types:

- **Finite Impulse Response (FIR) Filters:**
  - **Definition:** FIR filters have a finite-duration impulse response, meaning the output depends only on a limited number of input samples.
  - **Advantages:** They are inherently stable and can be designed to have linear phase characteristics.
  - **Implementation:** Typically implemented using a convolution sum, making them straightforward to simulate in hardware description languages like VHDL.
- **Infinite Impulse Response (IIR) Filters:**
  - **Definition:** IIR filters have an impulse response that lasts indefinitely due to feedback in their structure.
  - **Advantages and Drawbacks:** They can achieve a desired filtering effect with fewer coefficients than FIR filters but may suffer from stability issues and non-linear phase response.

2.2.3 Advantages of Digital Filtering

Digital filtering offers several benefits over its analog counterpart:

- **Flexibility:** Digital filters can be easily reprogrammed or adjusted through software without requiring physical modifications.
- **Precision:** Higher precision can be achieved through the use of digital arithmetic, and errors can be systematically managed.
- **Robustness:** Digital filters are less sensitive to component tolerances and environmental influences, leading to consistent performance.
- **Complexity:** Advanced filtering techniques that are difficult or impractical in analog systems can be implemented digitally.

2.2.4 Applications in Modern Systems

Digital filters are widely used in various applications, including:

- **Audio Processing:** Equalization, noise reduction, and effects processing in both consumer electronics and professional audio equipment.
- **Communications:** Signal conditioning in digital communication systems to improve data integrity and reduce interference.

- **Image Processing:** Enhancing images and reducing noise in digital cameras and medical imaging devices.
- **Control Systems:** Precise digital control in industrial applications where adaptive filtering techniques can optimize system performance.

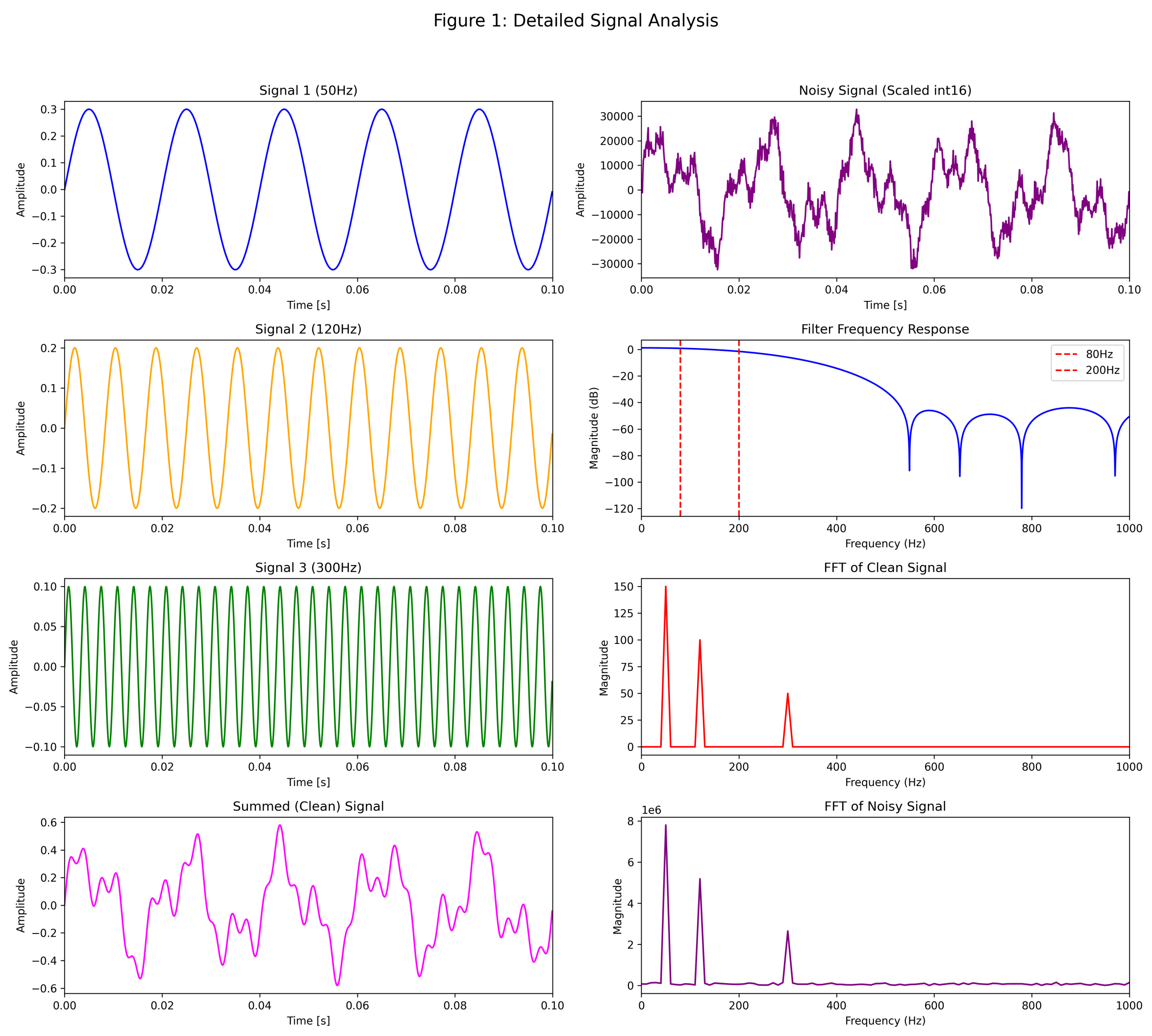


Figure 1: Signal building, Noise Introduction, Filter Impulse Response and FFTs

2.2.5 Implementation Considerations

Implementing digital filters in hardware, particularly using VHDL, involves several practical considerations:

- **Resource Utilization:** Efficient use of hardware resources such as memory and arithmetic units.
- **Latency and Throughput:** Balancing the speed of computation with the required filtering performance.

- **Numerical Precision:** Managing finite word lengths and quantization errors, particularly in fixed-point arithmetic implementations.

In this assignment, the emphasis is on digital filtering using an FIR filter design. The subsequent sections will delve into the detailed implementation aspects, covering convolution, sliding windows, binary arithmetic, and the practical application of these concepts in VHDL.

3 Convolution & Sliding Windows

3.1 Convolution in Digital Filtering

Convolution is the core operation in digital filtering, where the output signal is produced by combining the input signal with the filter's impulse response. In discrete time, the convolution of an input signal $x[n]$ and an impulse response $h[n]$ is defined by:

$$y[n] = \sum_{k=0}^{N-1} x[k]\, h[n-k]$$

For FIR filters, the sum is finite since the filter is defined by a limited number of coefficients, which simplifies computations and ensures stability.

3.2 Sliding Windows in FIR Filters

A sliding window is used to manage the most recent N input samples required for the convolution operation. As each new sample arrives, the window is updated by discarding the oldest sample and including the new one. Two common implementation approaches are described below.

3.3 Example 1: Parallel Processing Implementation

Step-by-Step Operation:

1. **Initialization:** Allocate an array (or shift register) to hold N samples and initialize it to zeros.
2. **Simultaneous Shifting:** On each clock cycle, all elements of the sliding window are shifted concurrently; that is, each sample is moved one position to the left.
3. **Insertion of New Sample:** The new sample is loaded into the rightmost position of the window.
4. **Convolution Calculation:** The updated window is used to compute the convolution sum with the predetermined filter coefficients.

Pseudocode Example:

```
process(clk)
begin
    if rising_edge(clk) then
        if reset = '1' then
            window <= (others => (others => '0'));
        else
            -- Parallel shifting: update all elements concurrently
            window(0) <= window(1);
            window(1) <= window(2);
            ...
            window(N-2) <= window(N-1);
            window(N-1) <= new_sample;
        end if;
    end if;
end process;
```

Explanation: In the parallel approach, the entire sliding window is updated in one clock cycle. Although this requires more hardware resources to shift all values simultaneously, it minimizes latency and ensures that the convolution operation always uses the most current data.

3.4 Example 2: Sequential Processing Implementation

Step-by-Step Operation:

1. **Initialization:** Similar to the parallel case, initialize an array of N samples to zeros.
2. **Sequential Shifting:** Instead of shifting all samples simultaneously, use a counter to update one element of the window per clock cycle. This spreads the shifting process over multiple cycles.
3. **Insertion of New Sample:** After the entire window has been shifted sequentially, insert the new sample at the final position.
4. **Convolution Calculation:** Once the window update is complete, compute the convolution sum using the new set of samples.

Pseudocode Example:

```
process(clk)
begin
    if rising_edge(clk) then
        if reset = '1' then
            window <= (others => (others => '0'));
            shift_index <= 0;
        else
            if shift_index < N-1 then
                window(shift_index) <= window(shift_index+1);
                shift_index <= shift_index + 1;
            else
                window(N-1) <= new_sample;
                shift_index <= 0;  -- Reset index for next update cycle
            end if;
        end if;
    end if;
end process;
```

Explanation: In the sequential approach, the sliding window is updated one element per clock cycle. This method reduces instantaneous hardware resource usage, as fewer operations are executed concurrently. However, it introduces additional latency since the complete update spans multiple cycles. This trade-off is acceptable in designs where resource constraints outweigh the need for ultra-fast processing.

3.5 VHDL Implementation Considerations

When implementing convolution and sliding window mechanisms in VHDL, several practical factors should be considered:

- **Memory Management:** Use arrays or shift registers effectively to store the sliding window data. Ensure that resource allocation is optimized for the target hardware.
- **Parallelism vs. Sequential Operation:** Choose between parallel and sequential shifting based on available hardware resources and the desired processing speed.
- **Timing and Control:** Carefully design control logic to manage data shifting and insertion. Ensure synchronization with the clock signal and proper handling of reset conditions.

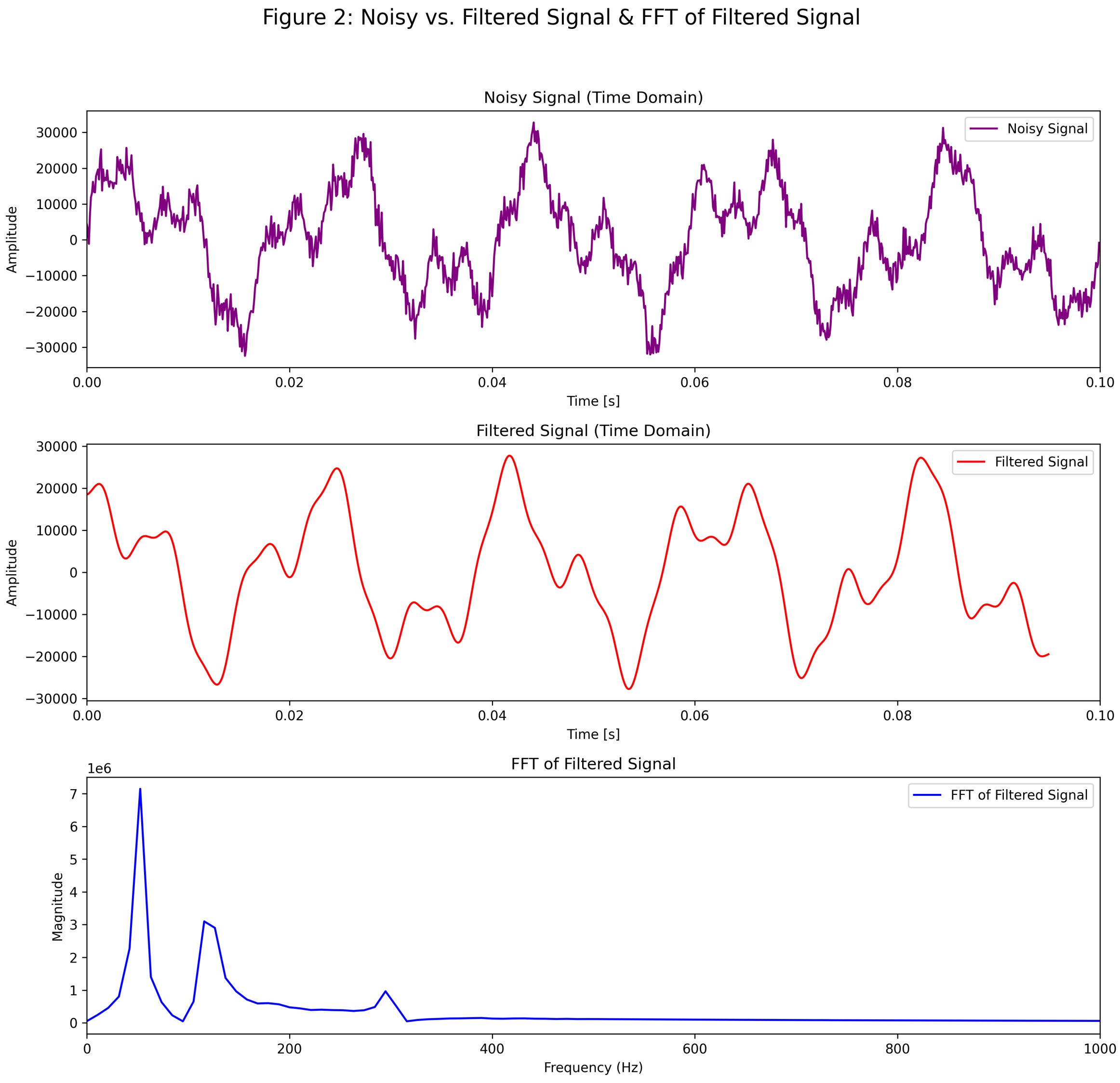


Figure 2: Result of Digital Filtering Applied in VHDL and Ploted in Python

- **Resource Optimization:** Balance the design between performance and hardware complexity. For instance, parallel processing minimizes latency at the cost of increased resource usage, while sequential processing conserves resources with added latency.
- **Testing and Simulation:** Verify the design using simulation tools (e.g., Modelsim) and validate the behavior with testbenches that emulate real-world signal conditions.

This section provides both theoretical background and practical examples to facilitate a robust understanding of convolution and sliding window techniques in digital filtering, setting the stage for detailed VHDL design in subsequent sections.

4 Multiplication in Binary

Binary multiplication is a fundamental arithmetic operation in digital systems. In this section, we discuss the principles of binary multiplication for both positive and negative numbers, with a focus on two's complement representation, which is widely used in digital systems and VHDL implementations.

4.1 Basic Binary Multiplication

Binary multiplication is analogous to decimal multiplication, but the only possible digits are 0 and 1. The operation involves:

- **Bitwise Multiplication:** Each bit of the multiplier multiplies the entire multiplicand, generating partial products.
- **Shifting:** Each partial product is shifted left by a number of positions corresponding to the bit's index (starting at 0 for the least-significant bit).
- **Accumulation:** The shifted partial products are added together to yield the final result.

Example: Multiply the two positive binary numbers:

$$\begin{aligned} \text{Multiplicand} &= 1011_2 \quad (11_{10}) \\ \text{Multiplier} &= 1101_2 \quad (13_{10}) \end{aligned}$$

Step-by-Step Process:

1. **Write Down the Numbers:**

$$\begin{array}{r} 1011 \quad \text{(multiplicand)} \\ \times\ 1101 \quad \text{(multiplier)} \end{array}$$

2. **Generate Partial Products:** Multiply the multiplicand by each bit of the multiplier (starting from the least-significant bit) and shift accordingly:

   (a) **LSB (bit = 1):** Partial product is

$$1011 \quad \text{(no shift)}$$

   (b) **Second Bit (bit = 0):** Partial product is

$$0000 \quad \text{(shifted left by 1)}$$

   (c) **Third Bit (bit = 1):** Partial product is

$$1011 \text{ shifted left by 2 } \rightarrow 101100$$

   (d) **MSB (bit = 1):** Partial product is

$$1011 \text{ shifted left by 3 } \rightarrow 1011000$$

3. **Sum the Partial Products:** Align and add the partial products:

$$\begin{array}{r} 1011 \\ +\quad 0000 \\ +\quad 101100 \\ +\ 1011000 \\ \hline 10001111 \end{array}$$

The final product is 10001111_2, which equals 143_{10} (since $11 \times 13 = 143$).

4.2 Multiplication of Negative Numbers

Digital systems typically represent negative numbers using two's complement. A common approach for signed multiplication is to multiply the absolute values and then assign the appropriate sign to the result.

Two's Complement Representation: For a 4-bit system, a positive number is represented normally, while a negative number is represented by inverting all bits of its positive counterpart and adding 1. For instance, to represent -3:

- Represent 3 in 4 bits: 0011_2.
- Invert to obtain 1100_2 and add 1 to get 1101_2.

4.3 Step-by-Step Example: Multiplying a Positive and a Negative Number

Consider multiplying $A = 6$ and $B = -3$ using 4-bit numbers. Multiply the absolute values and then apply the sign:

1. **Represent the Numbers:**
   - $A = 6$ is represented as 0110_2.
   - The magnitude of $B = 3$ is represented as 0011_2.
2. **Multiply the Absolute Values:** Multiply 0110_2 (6) by 0011_2 (3):
   (a) **LSB of 0011_2 (bit = 1):** Partial product is
   $$0110 \quad \text{(no shift)}$$
   (b) **Second Bit (bit = 1):** Partial product is
   $$0110 \text{ shifted left by } 1 \ \rightarrow 01100$$
   (c) **Remaining Bits (both 0):** Contribute partial products of 0000.
3. **Sum the Partial Products:**
   $$\begin{array}{r} 0110 \\ + \quad 01100 \\ \hline 10010 \end{array}$$
   The unsigned product is 10010_2, which equals 18_{10}.
4. **Apply the Sign:** Since B is negative, the final product must be negative. Represent 18 in 6 bits:
   $$18_{10} = 010010_2.$$
   Then, compute the two's complement:
   (a) Invert 010010_2 to obtain 101101_2.
   (b) Add 1 to 101101_2 to get 101110_2.

   The final result is 101110_2 (6-bit two's complement), representing -18_{10}.

4.4 VHDL Implementation Considerations

When implementing binary multiplication in VHDL, consider the following:

- **Data Representation:** Use the `signed` data type from the IEEE `numeric_std` package to handle both positive and negative numbers.
- **Arithmetic Operations:** Utilize built-in functions such as `resize` and `shift_right` to manage word lengths and sign adjustments.
- **Multiplication Algorithms:** For performance and resource optimization, consider algorithms like Booth's algorithm for signed multiplication, especially when targeting FPGA implementations with limited hardware multipliers.
- **Edge Cases and Overflow:** Account for potential overflow and ensure that the multiplication result is appropriately scaled or truncated to fit the desired bit-width.

5 VHDL Specifics

This section discusses key aspects of VHDL that are crucial for implementing digital filters. Topics include memory components, file I/O for simulation, bitwise manipulation, and useful functions that streamline design.

5.1 RAM and ROM in VHDL

Memory elements in VHDL are used to store data and parameters essential to digital designs. Two commonly used types are:

- **ROM (Read-Only Memory):** ROM is typically used to store constant data such as filter coefficients. It can be implemented as constant arrays or using dedicated memory components. For example, a ROM containing filter coefficients may be defined as:

  ```
  constant coeffs : array(0 to 50) of signed(15 downto 0) := (
      0, 1, -2, 3, ... -- Replace with actual coefficient values
  );
  ```

  Since the data is static, ROM designs are straightforward and resource-efficient.

- **RAM (Random Access Memory):** RAM is used for storing data that can be both read and modified during operation, such as input sample buffers or intermediate computation results. RAM is essential when handling sliding window techniques or temporary data storage. A simple RAM implementation might be:

  ```
  type ram_type is array(0 to 255) of signed(15 downto 0);
  signal ram : ram_type := (others => (others => '0'));
  ```

  A process can then be used to read from or write to this memory based on control signals.

5.2 File I/O in VHDL

File input and output operations are vital for simulation and verification. Using the `textio` package, VHDL can read from and write to text files, allowing designers to feed test vectors into a design and capture simulation results.

Example: Below is a simple example of file I/O operations in VHDL:

```
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;

entity file_io_example is
end file_io_example;

architecture behavioral of file_io_example is
    file infile  : text open read_mode is "input.txt";
    file outfile : text open write_mode is "output.txt";
    variable line_in  : line;
    variable line_out : line;
begin
```

```
    process
        variable data : integer;
    begin
        while not endfile(infile) loop
            readline(infile, line_in);
            read(line_in, data);
            -- Process data here
            write(line_out, data);
            writeline(outfile, line_out);
        end loop;
        wait;
    end process;
end behavioral;
```

5.3 Bitwise Manipulation in VHDL

Bitwise operations are indispensable in digital design, allowing for manipulation at the individual bit level. This includes:

- **Bit Shifting:** Functions such as `shift_left` and `shift_right` are used to multiply or divide by powers of two and to align data during arithmetic operations. **Example:** Shifting a 16-bit signed signal to the right by 2 positions:

  ```
  signal input_data : signed(15 downto 0);
  signal result     : signed(15 downto 0);
  ...
  result <= shift_right(input_data, 2);
  ```

  For left shifting:

  ```
  result <= shift_left(input_data, 3); -- Effectively multiplies the value by 8
  ```

- **Bit Masking:** Logical operators (AND, OR, XOR) are used to isolate or modify specific bits. **Example:** Masking the lower 4 bits of an 8-bit vector:

  ```
  signal data_in : std_logic_vector(7 downto 0);
  signal masked  : std_logic_vector(7 downto 0);
  ...
  masked <= data_in and "00001111";
  ```

- **Concatenation:** The concatenation operator (`&`) is used to combine bit vectors, enabling flexible data manipulation and reordering. **Example:** Combining two 4-bit vectors into an 8-bit vector:

  ```
  signal upper : std_logic_vector(3 downto 0);
  signal lower : std_logic_vector(3 downto 0);
  signal combined : std_logic_vector(7 downto 0);
  ...
  combined <= upper & lower;
  ```

5.4 Useful VHDL Functions

VHDL provides several built-in functions that facilitate efficient arithmetic and bit-level operations:

- **resize:** Adjusts the bit-width of a signal while preserving its numerical value. **Example:**

  ```
  signal a : signed(7 downto 0);
  signal b : signed(15 downto 0);
  ...
  b <= resize(a, 16); -- Converts an 8-bit signal to a 16-bit signal
  ```

- **shift_left/shift_right:** These functions perform bit shifts. **Example:**

  ```
  signal value : signed(15 downto 0);
  signal shifted_left : signed(15 downto 0);
  signal shifted_right: signed(15 downto 0);
  ...
  shifted_left  <= shift_left(value, 2);
  shifted_right <= shift_right(value, 2);
  ```

  Here, the value is multiplied by 4 when shifted left by 2, and divided by 4 when shifted right by 2.

- **to_integer and to_signed:** These conversion functions allow smooth transitions between different data types. **Example:**

  ```
  signal s_val : signed(7 downto 0);
  variable int_val : integer;
  ...
  int_val := to_integer(s_val);

  signal new_s_val : signed(7 downto 0);
  new_s_val <= to_signed(int_val, 8);
  ```

  This converts a signed vector to an integer and then back to an 8-bit signed vector.

5.5 Why Use Two's Complement

Two's complement is the preferred method for representing signed numbers in digital systems because it simplifies arithmetic operations. With two's complement:

- Addition, subtraction, and multiplication can be performed uniformly without requiring separate handling for positive and negative numbers.
- The representation naturally incorporates the sign of the number, reducing circuit complexity.
- It enables straightforward hardware implementations, as the same arithmetic circuits can process both signed and unsigned numbers.

This section covers key VHDL concepts that underpin efficient digital filter implementations, providing practical guidance on memory usage, file I/O, bitwise operations, and arithmetic functions.

6 Practical Tips & Common Pitfalls

Implementing digital filters in VHDL can be challenging. This section presents practical tips for effective debugging and design, highlights common pitfalls, and offers recommendations for optimization and further study.

6.1 Debugging Tips

- **Simulation and Verification:** Use simulation tools such as Modelsim to verify the behavior of your design before implementation. Construct comprehensive testbenches that include edge cases and real-world data scenarios.
- **File I/O for Debugging:** Employ file input/output operations to log intermediate results. Writing simulation data to text files can help in tracing issues related to memory access, arithmetic operations, or timing.
- **Signal Monitoring:** Monitor key signals, especially those related to control logic, memory read/write operations, and arithmetic outputs. Use waveform viewers to observe signal transitions and timing relationships.
- **Modular Testing:** Develop and test individual modules (e.g., the convolution block, memory management, and bitwise operations) separately before integrating them into the larger design.

6.2 Optimization & Efficiency

- **Resource Utilization:** Optimize your design by balancing hardware resource consumption and processing speed. For example, decide between parallel and sequential processing based on the available logic and speed requirements.
- **Code Readability:** Write clean, modular, and well-commented code. This practice not only aids debugging but also makes future modifications and optimizations easier.
- **Algorithm Selection:** Use efficient algorithms where possible. For instance, consider Booth's algorithm for signed multiplication if resource usage is a critical concern.
- **Fixed-Point Considerations:** When using fixed-point arithmetic, carefully manage word lengths and scaling factors to minimize quantization errors and prevent overflow.

6.3 Common Pitfalls and How to Avoid Them

- **Timing Issues:** Ensure that all processes are properly synchronized with the clock. Incorrect timing can lead to metastability or improper data shifting.
- **Memory Initialization:** Failing to properly initialize RAM or ROM can result in undefined behavior. Always include a reset routine to initialize memory elements.
- **Incorrect Bitwise Operations:** Misuse of bitwise operators can lead to erroneous data manipulation. Validate your logic with small test cases to ensure each operation performs as expected.
- **Overflow and Underflow:** Monitor arithmetic operations for overflow or underflow, especially when dealing with fixed-width data types. Use proper scaling and saturation techniques where necessary.

6.4 Suggestions for Further Exploration

- **Advanced VHDL Techniques:** Explore topics such as pipelining, parallel processing architectures, and the use of dedicated DSP blocks in FPGAs.
- **Deep Dive into DSP:** Further study digital signal processing theory to better understand filter design, spectral analysis, and adaptive filtering techniques.
- **Resource Material:** Consult textbooks, research articles, and online tutorials to expand your knowledge on VHDL design practices and digital filter implementations.

7 Conclusion & Summary

7.1 FIR Filter VHDL Template

This section provides two basic VHDL templates that you can use as a starting point for your FIR filter implementation. The first example demonstrates a parallel approach, while the second illustrates a sequential approach. Both examples are syntactically correct and include comments to guide your modifications.

7.1.1 Example 1: Parallel Implementation

This example updates the entire sliding window in one clock cycle and performs the convolution (multiply-accumulate) concurrently for all taps. It minimizes latency at the expense of higher resource usage.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity fir_filter is
    Port (
        clk          : in  std_logic;
        reset        : in  std_logic;
        start        : in  std_logic;
        sample_in    : in  signed(15 downto 0);
        sample_req   : out std_logic;
        filtered_out : out signed(15 downto 0)
    );
end fir_filter;

architecture Behavioral of fir_filter is
    constant NUM_TAPS : integer := 51;

    -- Define arrays for samples and filter coefficients
    type sample_array is array (0 to NUM_TAPS-1) of signed(15 downto 0);
    signal samples      : sample_array := (others => (others => '0'));
    signal coefs_array  : sample_array := (others => (others => '0'));

    -- Intermediate signal for multiply-accumulate result
    signal mac_result   : signed(31 downto 0) := (others => '0');

begin

    -- Process for sample loading and sliding window update
    process(clk)
    begin
        if rising_edge(clk) then
            if reset = '1' then
                samples <= (others => (others => '0'));
            elsif start = '1' then
                -- Load initial samples and filter coefficients (initialization code)
            else
                -- Update sliding window: shift all samples concurrently
                for i in 0 to NUM_TAPS-2 loop
                    samples(i) <= samples(i+1);
                end loop;
                samples(NUM_TAPS-1) <= sample_in;
            end if;
        end if;
    end process;

    -- Process for computing the convolution (multiply-accumulate)
    process(samples, coefs_array)
        variable sum : signed(31 downto 0);
    begin
        sum := (others => '0');
        for i in 0 to NUM_TAPS-1 loop
            sum := sum + (samples(i) * coefs_array(i));
        end loop;
        mac_result <= sum;
    end process;

    -- Output assignment with scaling (adjust as needed)
    filtered_out <= resize(shift_right(mac_result, 15), 16);
    sample_req   <= '1';  -- Control signal (adjust as necessary)

end Behavioral;
```

7.1.2 Example 2: Sequential Implementation

This example uses a sequential approach, processing one tap per clock cycle. A finite state machine (FSM) controls the operation: on a new sample arrival, the filter performs 8 cycles of multiplication–accumulation, then outputs the result, resets the coefficient counter, and updates the sample buffer. This reduces instantaneous resource usage at the cost of increased latency.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity fir_filter_seq is
    Port (
        clk          : in  std_logic;
        reset        : in  std_logic;
        start        : in  std_logic;  -- Indicates a new sample is available
        sample_in    : in  signed(15 downto 0);
        filtered_out : out signed(15 downto 0);
        done         : out std_logic   -- Indicates that the filtered output is ready
    );
end fir_filter_seq;

architecture Behavioral of fir_filter_seq is
    constant NUM_TAPS : integer := 8;

    -- Define a type for the sample buffer and coefficient array.
    type sample_array is array(0 to NUM_TAPS-1) of signed(15 downto 0);
    -- Example filter coefficients for demonstration
    signal coeffs : sample_array := (
        to_signed(1, 16), to_signed(2, 16), to_signed(3, 16), to_signed(4, 16),
        to_signed(4, 16), to_signed(3, 16), to_signed(2, 16), to_signed(1, 16)
    );

    -- Sample buffer for storing input samples
    signal sample_buffer : sample_array := (others => (others => '0'));

    -- Accumulator for multiply-accumulate operation
    signal acc : signed(31 downto 0) := (others => '0');

    -- Counter to iterate through filter taps
    signal tap_index : integer range 0 to NUM_TAPS := 0;

    -- FSM states for sequential processing
    type state_type is (IDLE, COMPUTE, OUTPUT);
    signal state : state_type := IDLE;

begin

    process(clk)
    begin
        if rising_edge(clk) then
            if reset = '1' then
                state         <= IDLE;
                tap_index     <= 0;
                acc           <= (others => '0');
                sample_buffer <= (others => (others => '0'));
                filtered_out  <= (others => '0');
                done          <= '0';
            else
                case state is
                    when IDLE =>
                        done <= '0';
                        if start = '1' then
                            -- New sample received; load into the buffer
                            sample_buffer(NUM_TAPS-1) <= sample_in;
                            tap_index <= 0;
                            acc <= (others => '0');
                            state <= COMPUTE;
                        end if;

                    when COMPUTE =>
                        if tap_index < NUM_TAPS then
                            -- Accumulate product of current sample and coefficient
                            acc <= acc + (sample_buffer(tap_index) * coeffs(tap_index));
                            tap_index <= tap_index + 1;
                        else
                            state <= OUTPUT;
                        end if;

                    when OUTPUT =>
                        -- Output the filtered result with scaling
                        filtered_out <= resize(shift_right(acc, 15), 16);
                        done <= '1';
                        -- Shift the sample buffer for the next sample
                        for i in 0 to NUM_TAPS-2 loop
                            sample_buffer(i) <= sample_buffer(i+1);
                        end loop;
                        state <= IDLE;

                    when others =>
                        state <= IDLE;
                end case;
            end if;
        end if;
    end process;

end Behavioral;
```

7.1.3 Comparison of the Two Approaches

- **Parallel Implementation:**
  - **Pros:**
    * Minimal latency; the entire sliding window and convolution operation is updated in one clock cycle.
    * Conceptually simpler since it uses a straightforward for-loop for shifting and accumulation.
  - **Cons:**
    * Higher hardware resource usage, as all operations occur concurrently.
    * May be impractical for systems with limited resources.
- **Sequential Implementation:**
  - **Pros:**
    * Lower resource utilization; processes one tap per cycle.
    * Suitable for devices with limited hardware resources.
  - **Cons:**
    * Increased latency, as the convolution operation is spread over multiple clock cycles.
    * Slightly more complex design due to the need for FSM control.

Choose the implementation approach that best fits your performance requirements and hardware constraints.

7.2 Summary of Key Concepts

In this assignment, you have explored:

- **Digital Filtering:** The principles behind digital filters, including convolution and sliding window techniques.

- **Binary Arithmetic:** The process of binary multiplication and handling of signed numbers using two's complement.
- **VHDL Essentials:** Key VHDL constructs such as memory elements (RAM and ROM), file I/O operations, bitwise manipulation, and useful arithmetic functions.
- **Practical Implementation:** Common pitfalls, optimization strategies, and debugging tips for effective VHDL design.

This template is intended to be a framework for your FIR filter project. Use it to build your design step by step, ensuring you understand the functionality of each component before integrating them into your final solution.