VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Computer Architecture Lab (CO2008)

PROJECT REPORT

FILTERING AND PREDICTION SIGNAL WITH WIENER FILTER

Lecturer: Nguyễn Thành Lộc

Semester: 251 - Class ID: CC03

No.	Student Name	ID
1	Nguyễn Việt Khoa	2452549
2	Phạm Võ Tiến Kha	2352461
3	Lương Gia Khánh	2452497

Ho Chi Minh City, November 2025

TEAM WORKLOAD

Student Name	Task Assignment	Evaluation
Phạm Võ Tiến Kha	• Implemented full C++ code for Wiener Filter algorithm. • Wrote algorithm explanation and pseudocode for report. • Compiled final report and proofread all sections. • Prepared result screenshots and diagrams for report.	100%
Nguyễn Việt Khoa	• Implemented signal processing core functions: `computeAutocorrelation()`, `computeCrosscorrelation()`, and `createToeplitzMatrix()`. • Designed and wrote `writeOutputToFile()` function. • Tested intermediate results and verified mathematical correctness.	100%
Lương Gia Khánh	• Implemented linear system solving (`solveLinearSystem()`) and main filtering routine (`applyWienerFilter()`). • Developed MIPS simulation version based on the C++ algorithm.	100%

TABLE OF CONTENTS

Chapter 1 Introduction

In real-world systems, signals are often corrupted by noise during transmission or processing. To estimate the desired signal from noisy observations, the Wiener filter is widely used because it provides the optimal linear filter that minimizes the Mean Square Error (MSE) between the output and the target signal.

This project aims to implement the Wiener filter in MIPS assembly using the MARS simulator. The program reads the input and desired signals from text files, computes the filtered output, evaluates the Minimum Mean Square Error (MMSE), and writes the results to both the console and an output file.

Through this assignment, students learn how to translate digital signal processing concepts into low-level assembly operations, understand the connection between theoretical models and computer architecture, and gain practical experience with numerical computation and file handling in MIPS.

The **Wiener filter** is a statistical approach used to estimate an unknown signal that has been corrupted by additive noise. It provides the *optimal linear filter* in the sense of minimizing the *Mean Square Error (MSE)* between the estimated output $y(n)$ and the desired signal $d(n)$.

The filter output can be expressed as:

$$y(n) = \sum_{k=0}^{M-1} h(k)\, x(n-k) \tag{1}$$

where $h(k)$ are the filter coefficients and $x(n)$ is the input signal.

The objective is to find the set of coefficients $h(k)$ that minimize:

$$E[e^2(n)] = E\left[(d(n) - y(n))^2\right] \tag{2}$$

This minimization leads to the **Wiener–Hopf equations** in matrix form:

$$R_x\, h = r_{dx} \tag{3}$$

where R_x is the autocorrelation matrix of the input signal, and r_{dx} is the cross-correlation vector between the desired and input signals.

Solving these equations yields the optimal coefficients h, which produce the filtered output $y(n)$. The performance of the filter is evaluated using the **Minimum Mean Square Error (MMSE)** value—the smaller the MMSE, the better the estimation accuracy.

Chapter 2 Problem Statement and Requirements

This project aims to implement a simplified Wiener filter using MIPS assembly on the MARS simulator. The program reads two sequences: the input (noisy) signal and the desired (clean) signal, both stored in text files named `input.txt` and `desired.txt`.

The main requirements are as follows:

- Read the input and desired sequences from external files.
- Verify that both signals have the same length; if not, display an error message.
- Compute the filtered output sequence based on the Wiener filtering process.
- Evaluate the performance by calculating the Minimum Mean Square Error (MMSE).
- Display and write the results to the file `output.txt`.

The expected output consists of:

1. A sequence of filtered output values.
2. The MMSE value printed with one decimal places.

Chapter 3 Program Design and Implementation

3.1 Overall Program Flowchart

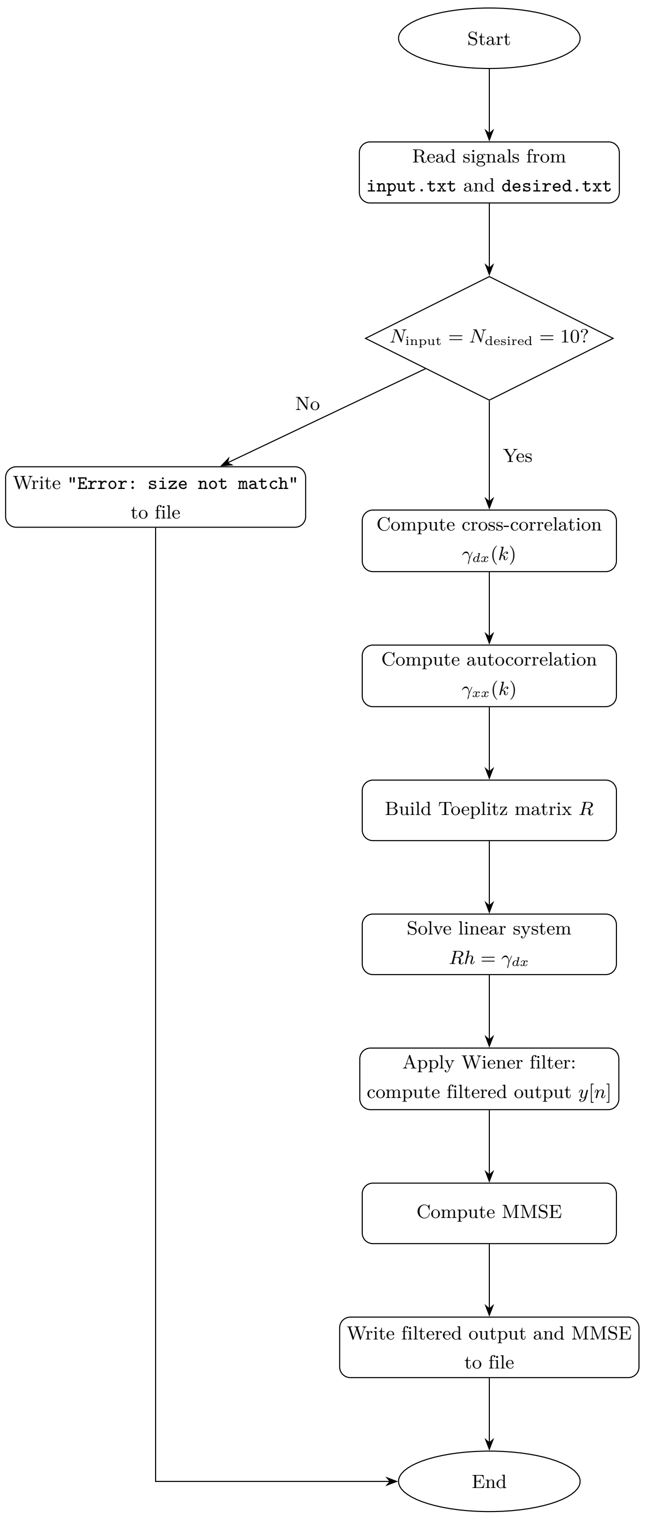

Start
Read signals from input.txt and desired.txt
$N_{\text{input}} = N_{\text{desired}} = 10?$
No
Yes
Write "Error: size not match" to file
Compute cross-correlation $\gamma_{dx}(k)$
Compute autocorrelation $\gamma_{xx}(k)$
Build Toeplitz matrix R
Solve linear system $Rh = \gamma_{dx}$
Apply Wiener filter: compute filtered output $y[n]$
Compute MMSE
Write filtered output and MMSE to file
End

Figure 1: Algorithm Design Flowchart

3.2 Data Section Design

The MIPS program relies heavily on a carefully structured `.data` segment. All signals, intermediate results, and constant values are pre-allocated here so that the assembly code can access them efficiently using simple base + offset addressing. The data section can be grouped into three main categories.

3.2.1 Buffers, sizes, and control parameters

At the top of the data segment, a large character buffer and several scalar control values are allocated:

- **`buf_size`** – integer constant specifying the maximum number of bytes that can be read from an input file (32 768 bytes).
- **`buffer`** – character array of size `buf_size` that temporarily holds the raw contents of `input.txt` or `desired.txt` before parsing.
- **`NUM_SAMPLES`** – integer constant equal to 10, defining the expected length of each signal.

These objects are defined after an `.align 2` directive to ensure that word and float data are 4-byte aligned in memory. This alignment is important for correct and efficient use of `lw`/`sw` and floating-point load/store instructions.

3.2.2 Signal arrays and intermediate variables

The core numerical data of the Wiener filter is stored in a set of single-precision float arrays:

- **`input`** – stores the noisy input signal $x[n]$ after parsing `input.txt`.
- **`desired`** – stores the desired (clean) signal $d[n]$ from `desired.txt`.
- **`crosscorr`** – holds the cross-correlation sequence $\gamma_{dx}(k)$.
- **`autocorr`** – holds the autocorrelation sequence $\gamma_{xx}(k)$.
- **`R`** – contiguous block of 400 bytes representing the 10×10 Toeplitz autocorrelation matrix in row-major order.
- **`coeff`** – stores the Wiener filter coefficients $h[k]$ obtained from solving the linear system.
- **`ouput`** – stores the filtered signal $y[n]$ (the symbol name preserves the original spelling `ouput` from the code).

Each 1-D signal array is allocated as `.space 40`, which corresponds to 10 single-precision floats $\times$ 4 bytes. The Toeplitz matrix `R` is allocated as `.space 400` ($10 \times 10 \times 4$ bytes). This fixed-size layout simplifies index calculation in the assembly code:

- an element at index i is always at address `base + 4 * i`,
- a matrix element $R[i][j]$ is at `R_base + 4 * (i * NUM_SAMPLES + j)`.

In addition to these arrays, the data section includes several auxiliary buffers and scalar floats:

- **`str_buf`** and **`temp_str`** – temporary character buffers used by the custom `float_to_str` and integer-to-string routines when formatting numbers for output.

- `mmse` – single-precision float that stores the final MMSE value computed by `computeMMSE`.
- **Floating-point constants** such as `zero_f`, `one_f`, `ten`, `hundred`, `half`, `minus_half`, `zero_thresh`, and `rounding_val`:
  - `ten` and `hundred` are used for scaling during decimal rounding and parsing.
  - `half` (0.5) is added before truncation to implement round-to-nearest behaviour.
  - `zero_thresh` (0.05) and `rounding_val` (0.05) are used in some printing routines to clamp very small values to 0.0 and to control rounding.

By storing these constants in memory instead of hard-coding them as immediates, the program can reuse them across many instructions via `l.s`, keeping the code shorter and clearer.

3.2.3 String literals for file names and messages

Finally, all human-readable text is placed at the end of the `.data` section using `.asciiz` directives.

File names

- `input_file`: `"input.txt"`
- `desired_file`: `"desired.txt"`
- `output_file`: `"output.txt"`

Output headers and separators

- `header_filtered`: `"Filtered output: "`
- `header_mmse`: `"\nMMSE: "`
- `space_str`: `" "`

Error messages

- `error_open`: `"Error: Can not open file"`
- `error_size`: `"Error: size not match"`

These null-terminated strings are passed directly to the MIPS system calls for file I/O and console printing. Grouping them in the data section keeps all textual information centralized and makes it easy to change file names or messages without touching the core algorithmic code.

3.3 Input Handling and Validation

This section explains how the program loads the two signals from text files and verifies that their lengths satisfy the assignment requirement. The logic is split into two procedures, `read_file_to_array` and `parse_floats_from_string`, and a simple size–checking step in `main`.

3.3.1 Reading files into memory

For both `input.txt` and `desired.txt`, the program calls:

```
read_file_to_array(filename, array, NUM_SAMPLES)
```

This procedure encapsulates all low-level file I/O:

1. **Open the file**
   - Uses syscall 13 to open `filename` in read-only mode.
   - If the returned file descriptor is negative, the program prints `"Error: Can not open file"` and terminates.
2. **Read the entire content**
   - Uses syscall 14 to read up to `buf_size` bytes into the global character buffer `buffer`.
   - The number of bytes actually read is returned by the syscall.
3. **Close the file**
   - Uses syscall 16 to release the file descriptor.
4. **Parse the buffer**
   - Calls `parse_floats_from_string(buffer, array, NUM_SAMPLES)` to convert the ASCII text into floating-point samples stored in `array`.
   - The return value (number of samples successfully parsed) is propagated back to the caller.

In `main`, this procedure is called twice:

```
count_input   = read_file_to_array("input.txt",   input,   NUM_SAMPLES)
count_desired = read_file_to_array("desired.txt", desired, NUM_SAMPLES)
```

so that both signals are loaded in a uniform manner.

3.3.2 Float parsing

The helper routine `parse_floats_from_string` performs the actual conversion from characters to floating-point numbers:

- It scans the buffer from left to right, skipping whitespace characters (`' '`, `'\n'`, `'\r'`, `'\t'`).
- For each number, it optionally reads a minus sign `'-'`, then accumulates the integer part digit by digit.
- If a decimal point `'.'` is present, it continues to read the fractional part, maintaining a power-of-ten divisor (10.0, 100.0, . . .) to correctly place the digits after the decimal point.
- The integer and fractional parts are combined, the sign is applied, and the resulting value is stored into the target float array.
- The process repeats until either `N` numbers have been parsed or the end of the string is reached.

In pseudocode:

```
procedure parse_floats_from_string(buf, arr, N):

    i ← 0
    p ← 0

    while i < N and buf[p] != '\0' do

        # Skip whitespace
        skip whitespace at buf[p]
        if buf[p] = '\0' then
            break
        end if

        # Optional sign
        sign ← +1
        if buf[p] = '-' then
            sign ← -1
            p ← p + 1
        end if

        # Integer part
        int_part ← 0
        while buf[p] is digit do
            int_part ← int_part * 10 + (buf[p] - '0')
            p ← p + 1
        end while

        # Fractional part (optional)
        frac_part ← 0
        divisor   ← 1
        if buf[p] = '.' then
            p ← p + 1
            while buf[p] is digit do
                frac_part ← frac_part * 10 + (buf[p] - '0')
                divisor   ← divisor * 10
                p ← p + 1
            end while
        end if

        # Combine and store
        value  ← sign * (int_part + frac_part / divisor)
        arr[i] ← value
        i      ← i + 1

    end while

    return i
```

```
end procedure
```

3.3.3 Signal size checking and error handling

After loading both files, the `main` program validates the signal lengths:

```
if (count_input   != NUM_SAMPLES) or
   (count_desired != NUM_SAMPLES) then
    go to size_error
else
    proceed with Wiener filter computations
end if
```

where:

- `count_input` is the number of samples parsed from `input.txt`.
- `count_desired` is the number of samples parsed from `desired.txt`.
- `NUM_SAMPLES` is fixed at 10.

If either count is different from 10, the program does not attempt to compute correlations or filter coefficients. Instead, it jumps to the `size_error` branch, where it:

1. Opens `output.txt` in write-only mode.
2. Writes the exact error message:

   ```
   Error: size not match
   ```

3. Closes the file and terminates.

This guarantees that any malformed input (wrong number of samples in one or both files) produces a deterministic, easy-to-check output, exactly matching the specification.

3.4 Cross-Correlation Computation

The cross-correlation between the desired signal $d[n]$ and the noisy input signal $x[n]$ is a key component of the Wiener–Hopf equations. In the implemented system, this quantity is stored in the array `crosscorr` and corresponds to the right-hand side vector γ_{dx} of the linear system

$$R \cdot h = \gamma_{dx}.$$

3.4.1 Mathematical definition

For a finite sequence of length N, the biased estimate of the cross-correlation between $d[n]$ and $x[n]$ at lag k is defined as:

$$\gamma_{dx}(k) = \frac{1}{N} \sum_{n=k}^{N-1} d[n]\, x[n-k], \qquad k = 0, 1, \ldots, N-1.$$

Intuitively, $\gamma_{dx}(k)$ measures how similar the desired signal is to a shifted version of the input signal. When we stack these values into a vector

$$\gamma_{dx} = \left[\gamma_{dx}(0), \gamma_{dx}(1), \ldots, \gamma_{dx}(N-1)\right]^T,$$

this vector becomes the target term in the Wiener–Hopf equation used to compute the optimal filter coefficients $h[k]$.

3.4.2 Algorithm and pseudocode

The computation of `crosscorr` follows directly from the definition above. For each lag k, the program accumulates the product $d[n] \cdot x[n-k]$ over all valid indices and then normalizes by N.

High-level pseudocode:

```
procedure computeCrosscorrelation(desired, input, crosscorr, N):

    for k ← 0 to N - 1 do
        sum ← 0.0

        for n ← k to N - 1 do
            sum ← sum + desired[n] · input[n - k]
        end for

        crosscorr[k] ← sum / N
    end for

end procedure
```

Key points:

- The outer loop iterates over the lag k.
- For each k, the inner loop runs from $n = k$ up to $N-1$, ensuring that the index $n-k$ is always non-negative and within bounds of the input array.
- After finishing the inner loop, the accumulated sum is divided by N (not by $N-k$), which implements the biased estimator.
- The result for each lag is stored in `crosscorr[k]`.

3.4.3 Mapping to the MIPS implementation

In the MIPS version, the procedure `computeCrosscorrelation` receives:

- the base address of the desired signal `desired` in one argument register,

- the base address of the input signal `input` in another,
- the base address of the output array `crosscorr`,
- and the length N (here, 10 samples).

The two nested loops are implemented with integer counters:

- the outer loop variable corresponds to the lag k,
- the inner loop variable corresponds to the index n.

For each pair (n, k):

- The address of `desired[n]` is computed as `desired_base + 4 * n`.
- The address of `input[n - k]` is computed as `input_base + 4 * (n - k)`.
- Both values are loaded into floating-point registers, multiplied, and accumulated in a floating-point accumulator register that holds the partial sum for the current lag k.

After the inner loop finishes for a given k, the accumulator is divided by N (which is first converted from integer to float) and the resulting value is stored at `crosscorr[k]`. The outer loop then continues with the next lag until all N lags have been processed.

3.5 Autocorrelation Computation

The autocorrelation of the input signal $x[n]$ is another fundamental component of the Wiener filter. It is used to build the Toeplitz matrix R that appears on the left-hand side of the Wiener–Hopf equation

$$R \cdot h = \gamma_{dx}.$$

In the implementation, the autocorrelation values are stored in the array `autocorr`.

3.5.1 Mathematical definition

For a finite sequence $x[n]$ of length N, the biased estimate of the autocorrelation at lag k is defined as

$$\gamma_{xx}(k) = \frac{1}{N} \sum_{n=k}^{N-1} x[n]\, x[n-k], \qquad k = 0, 1, \ldots, N-1.$$

- For $k = 0$, $\gamma_{xx}(0)$ is simply the average signal power.
- For $k > 0$, $\gamma_{xx}(k)$ measures the similarity between the signal and a shifted version of itself.

Just like in the cross-correlation computation, the implementation uses the *biased* estimator (division by N for all lags).

The vector of autocorrelation values is

$$\gamma_{xx} = \left[\gamma_{xx}(0), \gamma_{xx}(1), \ldots, \gamma_{xx}(N-1)\right]^T,$$

and these entries are later reused to construct the Toeplitz matrix R.

3.5.2 Algorithm and pseudocode

The algorithm is similar in structure to the cross-correlation, but uses only the input signal `input`:

```
procedure computeAutocorrelation(input, autocorr, N):

    for k ← 0 to N - 1 do
        sum ← 0.0

        for n ← k to N - 1 do
            sum ← sum + input[n] · input[n - k]
        end for

        autocorr[k] ← sum / N
    end for

end procedure
```

Key aspects:

- The outer loop iterates over the lag k.
- The inner loop runs from $n = k$ to $N - 1$, so that the index $n - k$ is always valid (non-negative and less than N).
- Each term in the sum is a product of two samples of the same signal at different time indices.
- After the inner loop, the sum is normalized by dividing by N, and the result is stored into `autocorr[k]`.

3.5.3 Mapping to the MIPS implementation

The MIPS procedure `computeAutocorrelation` receives:

- the base address of the input signal array `input`,
- the base address of the output array `autocorr`,
- and the number of samples N.

The nested loops are implemented with two integer counters:

- k is stored in one general-purpose register (outer loop),
- n is stored in another (inner loop).

For each pair (n, k):

- The address of `input[n]` is computed as `input_base + 4 · n` and loaded into a floating-point register.
- The index $n - k$ is computed in integer registers, multiplied by 4, and added to the base address to obtain the address of `input[n - k]`. This value is also loaded into a floating-point register.

- The product `input[n] * input[n - k]` is computed in the FPU and added to a floating-point accumulator register that holds the running sum for the current lag k.

When the inner loop finishes:

- The integer N is converted to a floating-point value,
- the accumulator is divided by N, and
- the result is stored at `autocorr[k]` (again using base+offset addressing with `k * 4`).

The outer loop then increments k and continues until all lags from 0 to $N-1$ have been processed.

The MIPS `computeAutocorrelation` routine produces a consistent autocorrelation vector γ_{xx}, which is then reused in the next design step: constructing the Toeplitz matrix in Section 3.6.

3.6 Toeplitz Matrix Construction

The Toeplitz autocorrelation matrix R is the core structure on the left-hand side of the Wiener-Hopf equation. It encodes how the input signal $x[n]$ correlates with shifted versions of itself and is built directly from the previously computed autocorrelation sequence $\gamma_{xx}(k)$, stored in the array `autocorr`.

3.6.1 Role in the Wiener filter

For an FIR Wiener filter of length N, the Wiener-Hopf equation can be written as:

$$R \cdot h = \gamma_{dx},$$

where

- $h = [h(0), h(1), \ldots, h(N-1)]^T$ is the vector of optimal filter coefficients,
- γ_{dx} is the cross-correlation vector between the desired signal and the input signal,
- R is the $N \times N$ autocorrelation matrix of the input signal.

The entries of R are defined in terms of the autocorrelation sequence $\gamma_{xx}(k)$ as:

$$R[i,j] = \gamma_{xx}(|i-j|), \qquad i,j = 0,1,\ldots,N-1.$$

This structure implies that:

- Each diagonal of R is constant: elements with the same $|i-j|$ share the same value.
- The matrix is symmetric ($R[i,j] = R[j,i]$) and Toeplitz (constant along diagonals).

By constructing R from the `autocorr` array, the implementation ensures that the subsequent linear system solver operates on a matrix that is consistent with the statistical properties of the input signal.

3.6.2 Algorithm and pseudocode

The construction of R is straightforward once the autocorrelation values $\gamma_{xx}(k)$ are known. For each pair of indices (i,j), the absolute difference $|i-j|$ selects the corresponding element from the `autocorr` array:

```
procedure createToeplitzMatrix(autocorr, R, N):

    for i ← 0 to N - 1 do
        for j ← 0 to N - 1 do
            index   ← abs(i - j)
            R[i][j] ← autocorr[index]
        end for
    end for

end procedure
```

Key points:

- The outer loop iterates over the matrix rows i.
- The inner loop iterates over the columns j for each row.
- The index into `autocorr` is the absolute difference `abs(i - j)`, which guarantees that all values on the same diagonal of the matrix share the same autocorrelation coefficient.
- No additional normalization is needed, because `autocorr[k]` already contains the biased estimate $\gamma_{xx}(k)$.

3.6.3 Mapping to the MIPS implementation

In the MIPS procedure `createToeplitzMatrix`, the following arguments are passed:

- `autocorr` - base address of the autocorrelation array (input),
- `R` - base address of the matrix storage (output),
- `N` - dimension of the matrix (here, `NUM_SAMPLES = 10`).

The two nested loops are implemented with integer registers:

- One register holds the current row index i.
- Another holds the column index j.

For each pair (i, j), the code performs:

1. **Compute absolute difference**
    - Compute `t2 = i - j`.
    - If `t2` is negative, it is negated to obtain $|i - j|$.
2. **Load autocorrelation value**
    - The offset into `autocorr` is `4 * |i - j|` (each entry is a 4-byte float).
    - The address `autocorr_base + 4 * |i - j|` is formed and loaded into a floating-point register using `l.s`.
3. **Compute matrix index and store**

- The 2D index (i, j) is converted into a 1D offset using row-major order:

$$\text{index} = i \cdot N + j.$$

- This index is multiplied by 4 to obtain the byte offset: `4 * (i * N + j)`.
- The value loaded from `autocorr` is then stored at `R_base + 4 * (i * N + j)` using `s.s`.

The outer and inner loops continue until all pairs (i, j) from $(0, 0)$ to $(N-1, N-1)$ have been processed, at which point the matrix R is fully populated.

3.7 Linear System Solver (solveLinearSystem)

Once the autocorrelation matrix R and the cross-correlation vector γ_{dx} have been computed, the next step is to solve the Wiener-Hopf equation in order to obtain the optimal filter coefficients:

$$R \cdot h = \gamma_{dx}.$$

In the implementation, the matrix R is stored in the array `R`, the right-hand side vector γ_{dx} is stored in `crosscorr`, and the solution vector of coefficients is stored in `coeff`. The procedure `solveLinearSystem` is responsible for computing `coeff` from `R` and `crosscorr`.

3.7.1 Problem statement

For a filter length N (here $N = 10$), the system has the form

$$\begin{bmatrix} R[0,0] & R[0,1] & \cdots & R[0,N-1] \\ R[1,0] & R[1,1] & \cdots & R[1,N-1] \\ \vdots & \vdots & \ddots & \vdots \\ R[N-1,0] & R[N-1,1] & \cdots & R[N-1,N-1] \end{bmatrix} \begin{bmatrix} h(0) \\ h(1) \\ \vdots \\ h(N-1) \end{bmatrix} = \begin{bmatrix} \gamma_{dx}(0) \\ \gamma_{dx}(1) \\ \vdots \\ \gamma_{dx}(N-1) \end{bmatrix}$$

The goal is to solve for the coefficient vector h efficiently in MIPS assembly. The chosen method is *Gaussian elimination without pivoting*, which is sufficient for the small, well-conditioned 10×10 system in this assignment.

3.7.2 Algorithm overview

The procedure `solveLinearSystem` operates in two main phases:

1. **Forward elimination**
   - Transform the matrix R into an upper triangular form.
   - Apply the same row operations to the right-hand side vector `crosscorr`.
2. **Back substitution**
   - Starting from the last equation, solve for $h(N-1)$ and move upwards.
   - At each step, subtract already-known terms and divide by the diagonal element.

3.7.3 Pseudocode

A high-level pseudocode version of `solveLinearSystem(R, b, x, N)` (where `b` is `crosscorr` and `x` is `coeff`) is:

```
procedure solveLinearSystem(R, b, x, N):

    # Forward elimination
    for k ← 0 to N - 1 do
        pivot ← R[k][k]

        for i ← k + 1 to N - 1 do
            factor ← R[i][k] / pivot

            # Eliminate column k from row i
            for j ← k to N - 1 do
                R[i][j] ← R[i][j] - factor · R[k][j]
            end for

            # Apply the same operation to the RHS
            b[i] ← b[i] - factor · b[k]
        end for
    end for

    # Back substitution
    for i ← N - 1 down to 0 do
        sum ← b[i]
        for j ← i + 1 to N - 1 do
            sum ← sum - R[i][j] · x[j]
        end for

        x[i] ← sum / R[i][i]
    end for

end procedure
```

Key points:

- The forward elimination loop over k zeros out all entries below the diagonal in column k.
- The back substitution loop over i computes each `x[i]` using already computed `x[j]` for $j > i$.
- All divisions and multiplications are performed in floating point to maintain numerical accuracy.

For this assignment, pivoting (row swapping) is not implemented, which simplifies the code and is acceptable for small, non-pathological matrices.

3.7.4 Mapping to the MIPS implementation

In the MIPS program, the procedure `solveLinearSystem` receives:

- `a0` - base address of the matrix `R` (stored in row-major order),
- `a1` - base address of the right-hand side vector `crosscorr` (`b`),
- `a2` - base address of the output vector `coeff` (`x`),
- `a3` - the dimension N.

The mapping of the pseudocode to MIPS includes:

1. **Matrix indexing**
   - The element $R[i, j]$ is located at offset `4 * (i * N + j)` from the base address `R`. Integer registers compute $(i \cdot N + j)$, which is then multiplied by 4 for byte addressing.
2. **Forward elimination phase**
   - The loop indices k, i, and j are stored in general-purpose registers.
   - The pivot $R[k, k]$ is loaded into a floating-point register .
   - For each row $i > k$, the factor
     $$\text{factor} = \frac{R[i,k]}{R[k,k]}$$
     is computed in the FPU and stored in a floating register .
   - The inner loop over j updates each matrix element as
     $$R[i,j] \leftarrow R[i,j] - \text{factor} \cdot R[k,j],$$
     using `mul.s` and `sub.s` instructions.
   - After updating the matrix row, the corresponding entry of the RHS is updated using the same factor:
     $$b[i] \leftarrow b[i] - \text{factor} \cdot b[k].$$
3. **Back substitution phase**
   - The outer loop index i starts from $N - 1$ and decrements down to 0.
   - A floating accumulator (e.g. `f16`) is initialised with $b[i]$.
   - The inner loop over $j = i+1, \ldots, N-1$ subtracts the contributions of already known coefficients:
     $$\text{sum} \leftarrow \text{sum} - R[i,j] \cdot x[j].$$
   - After the inner loop, the diagonal element $R[i, i]$ is loaded, and the solution component is computed as
     $$x[i] = \frac{\text{sum}}{R[i,i]},$$
     and stored into the `coeff` array at offset `4 * i`.

Throughout this procedure, all matrix entries and vectors are updated *in place*: `R` and `crosscorr` are overwritten by their triangular and transformed forms.

3.7.5 Discussion and numerical considerations

For the small system size $N = 10$, Gaussian elimination without pivoting is a reasonable trade-off between simplicity and performance. The Toeplitz matrix R constructed from the biased autocorrelation of the input signal is typically well-behaved in the context of this assignment, so the absence of pivoting does not cause numerical instability in practice.

Moreover:

- All operations are carried out in single-precision floating point.
- Any differences between the two implementations are limited to minor rounding effects.

As a result, the computed `coeff` vector produces filtered outputs and MMSE values up to small floating-point rounding differences.

3.8 Wiener Filtering (applyWienerFilter)

Once the optimal Wiener filter coefficients $h[k]$ have been obtained by solving the Wiener-Hopf equations, the next step is to apply this FIR filter to the noisy input signal $x[n]$ in order to produce the filtered output signal $y[n]$. This operation is implemented in the procedure `applyWienerFilter` and corresponds to a causal convolution between the coefficient vector and the input signal.

3.8.1 Mathematical definition

For an FIR filter of length N, the output $y[n]$ of the Wiener filter is given by

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

with the understanding that terms with negative indices are ignored, i.e. $x[n-k]$ is used only if $n-k \geq 0$.

For each output sample $y[n]$:

- We look back at the current and previous input samples $x[n], x[n-1], \ldots,$
- We weight them by the corresponding filter coefficients $h[0], h[1], \ldots,$
- We sum all valid contributions where the index $n-k$ remains within the bounds of the input signal.

This is the standard causal FIR filtering structure.

3.8.2 Algorithm and pseudocode

Given

- `input` - array containing the noisy input signal $x[n]$,
- `coeff` - array of Wiener filter coefficients $h[k]$,
- `output` - array that will store the filtered signal $y[n]$,
- `N` - number of samples and filter length (here, $N = 10$),

the algorithm implemented by `applyWienerFilter` can be expressed as:

```
procedure applyWienerFilter(input, coeff, output, N):

    for n ← 0 to N - 1 do
        sum ← 0.0

        for k ← 0 to N - 1 do
            idx ← n - k
            if idx >= 0 then
                sum ← sum + coeff[k] · input[idx]
            end if
        end for

        output[n] ← sum
    end for

end procedure
```

Key points:

- The outer loop runs over each output index n.
- The inner loop runs over all filter taps k.
- The check `idx = n - k` and `idx >= 0` enforces causality and prevents reading outside the valid input range. For small n, only a subset of the coefficients contribute.
- After the inner loop, the accumulated sum becomes `output[n]`, i.e. the filtered sample $y[n]$.

3.8.3 Mapping to the MIPS implementation

In the MIPS program, the procedure `applyWienerFilter` receives:

- `$a0` - base address of the input array (noisy signal),
- `$a1` - base address of the `coeff` array (filter coefficients),
- `$a2` - base address of the `ouput` array (filtered signal),
- `$a3` - the length N.

The nested loops are implemented using integer registers:

- One register holds the outer index n (output sample index).
- Another holds the inner index k (coefficient index).

For each pair (n, k), the following operations are performed:

1. **Index check**
    - Compute `idx = n - k` in an integer register.
    - If `idx` is negative, the program skips directly to the next k (no contribution).

2. **Address computation and loading**
   - The coefficient address is `coeff_base + 4 * k` $\to$ load $h[k]$ into a floating-point register.
   - The input sample address is `input_base + 4 * idx` $\to$ load $x[\mathrm{idx}]$ into another floating-point register.
3. **Multiply–accumulate**
   - Compute the product $h[k] \cdot x[\mathrm{idx}]$ using `mul.s`.
   - Add the product to the running sum `sum` using `add.s`.

After the inner loop completes for a given n, the accumulated sum is stored to

$$\texttt{ouput\_base} + 4 \cdot n$$

using `s.s`, thereby setting `ouput[n]` $= y[n]$. The outer loop then increments n and repeats until all N output samples have been computed.

3.9 MMSE Computation

After the Wiener filter has been applied and the filtered signal $y[n]$ has been obtained, the quality of the reconstruction is measured using the Minimum Mean Square Error (MMSE) between the filtered output and the desired signal. This scalar value is stored in the global variable `mmse` and printed as part of the final program output.

3.9.1 Mathematical definition

Given

- the desired (clean) signal $d[n]$,
- the filtered output signal $y[n]$,
- and the number of samples N (here $N = 10$),

the MMSE is defined as

$$\mathrm{MMSE} = \frac{1}{N} \sum_{n=0}^{N-1} \big(d[n] - y[n]\big)^2.$$

This metric:

- is always non-negative,
- becomes zero only if the filtered output exactly matches the desired signal at every sample,
- decreases as the Wiener filter better approximates the desired signal.

In the implementation, `desired` corresponds to $d[n]$, `ouput` corresponds to $y[n]$, and the result of the above expression is stored in the single-precision float `mmse`.

3.9.2 Algorithm and pseudocode

The MMSE computation is straightforward once both `desired` and `ouput` have been fully computed. The procedure iterates over all samples, accumulates the squared error, and normalizes by N:

```
procedure computeMMSE(desired, output, N, &mmse):

    sum ← 0.0

    for i ← 0 to N - 1 do
        e   ← desired[i] - output[i]
        sum ← sum + e · e
    end for

    mmse ← sum / N

end procedure
```

Key points:

- The loop index i runs from 0 to $N-1$.
- For each i, the instantaneous error $e = d[i] - y[i]$ is computed.
- The squared error e^2 is accumulated into `sum`.
- After the loop, `sum` is divided by N to obtain the mean squared error.

3.9.3 Mapping to the MIPS implementation

In the MIPS program, the procedure `computeMMSE` receives:

- `$a0` - base address of the `desired` array,
- `$a1` - base address of the `ouput` array,
- `$a2` - the number of samples N.

Additionally, the address of the global variable `mmse` is passed indirectly via the stack, so the procedure can write the final result back to memory.

The implementation follows the pseudocode above:

1. **Initialization**
   - A floating-point accumulator register is set to 0.0 to represent `sum`.
   - An integer loop counter i is initialised to 0.
2. **Main loop** For each i from 0 to $N-1$:
   - Compute the byte offset `4 * i`.
   - Load `desired[i]` and `ouput[i]` from memory using `l.s`.
   - Compute the difference $e = d[i] - y[i]$ with `sub.s`.

- Square the error $e \cdot e$ using `mul.s`.
- Add the squared error to the accumulator using `add.s`.

3. **Normalization**

   - Convert the integer N to a floating-point value in the FPU.
   - Divide the accumulated sum by N using `div.s` to obtain the final MMSE value.

4. **Store result**

   - Store the resulting MMSE into the global variable `mmse` using `s.s`.

At the end of `computeMMSE`, the memory location associated with `mmse` holds the final error value, which is later read, rounded to one decimal place, converted to a string and written to `output.txt` in the format:

```
MMSE: m.m
```

3.10 Number Formatting and Output Generation

The final stage of the program is responsible for presenting the numerical results in a human-readable format and writing them to the output file `output.txt`. This involves two aspects:

- Defining a clear, fixed text format for the results.
- Converting internal floating-point values into ASCII strings with controlled rounding and sign behaviour.

Both are handled entirely in MIPS using custom formatting routines.

3.10.1 Output file format

When the input size is valid, the program produces an output file `output.txt` with two lines:

```
Filtered output: y0 y1 y2 ... y9
MMSE: m.m
```

- The first line begins with the header string `"Filtered output: "`, followed by the 10 filtered samples of the output signal.
- Each sample y_i is printed with one decimal digit, separated by a single space.
- The second line begins with `"\nMMSE: "`, followed by the MMSE value, also formatted with one decimal digit.

At a high level, the logic in `main` can be described as:

```
open("output.txt", write_only)

write("Filtered output: ")

for i ← 0 to NUM_SAMPLES - 1 do
    s ← float_to_str(output[i], decimals = 1)
    write(s)
    if i < NUM_SAMPLES - 1 then
        write(" ")
    end if
end for

write("\nMMSE: ")

mmse_rounded ← round_to_1dec(mmse)
s_mmse      ← float_to_str(mmse_rounded, decimals = 1)
write(s_mmse)

close file
```

If the input size is invalid (see Section 3), the program instead writes a single line

```
Error: size not match
```

and terminates. This fixed string is used to make automated checking easy and to strictly follow the assignment specification.

3.10.2 Rounding to one decimal place

Before printing the MMSE value, the program calls a helper routine `round_to_1dec`. This function performs the rounding purely in floating point:

- Multiply the original value by 10.
- Add or subtract 0.5 depending on the sign (to implement round-to-nearest):

$$\text{for positive } x: \quad x' = x \cdot 10 + 0.5, \qquad \text{for negative } x: \quad x' = x \cdot 10 - 0.5.$$

- Truncate x' to an integer (discarding extra digits).
- Convert the integer back to float and divide by 10.

Conceptually:

```
procedure round_to_1dec(x):

    temp ← x * 10.0
    if x < 0 then
        temp ← temp - 0.5
    else
        temp ← temp + 0.5
    end if

    n ← trunc(temp)      # integer
    return n / 10.0

end procedure
```

3.10.3 Float-to-string conversion

The core formatting work is done by the custom routine

`float_to_str(buffer, value, decimals)`

which converts a single-precision float into an ASCII representation stored in `buffer`. The parameter `decimals` controls whether 0, 1, or 2 digits are printed after the decimal point.

The algorithm proceeds in several steps.

1. **Sign handling**
   - If the input value is negative, the character `'-'` is written to the buffer and the value is replaced by its absolute value.
2. **Scaling and rounding**
   - To support decimal places, the value is scaled by

     $$1 \text{ (for } \texttt{decimals} = 0), \quad 10 \text{ (for } \texttt{decimals} = 1), \quad 100 \text{ (for } \texttt{decimals} = 2).$$

   - Then 0.5 is added before truncation to implement rounding to the nearest integer.
   - The truncated scaled value is stored in an integer variable `N`.
3. **Splitting integer and fractional parts** For `decimals > 0`, the integer part and fractional digits are recovered by integer division and modulo:
   - With 1 decimal:
     $$\texttt{intPart} = N/10, \qquad \texttt{frac1} = N \bmod 10.$$
   - With 2 decimals:
     $$\texttt{intPart} = N/100,$$
     and the first and second fractional digits are extracted from the remainder.
   - For `decimals = 0`, only `intPart` is needed.
4. **Zero normalization**
   - There is a small fix to avoid printing `-0.0`: if both `intPart` and all fractional digits are zero, the routine resets the buffer so that the result is printed as `"0.0"` (or `"0"` for zero decimals), regardless of the original sign.
5. **Integer-to-string conversion**
   - The integer part is converted to text by repeatedly taking `mod 10` and `div 10` and storing characters `'0'`–`'9'` into a temporary buffer in reverse order, then copying them back in the correct order.
6. **Assembling the final string**
   - The integer part is written to `buffer`.
   - If `decimals > 0`, a dot `'.'` is appended, followed by one or two fractional digits converted to characters.

In pseudocode form:

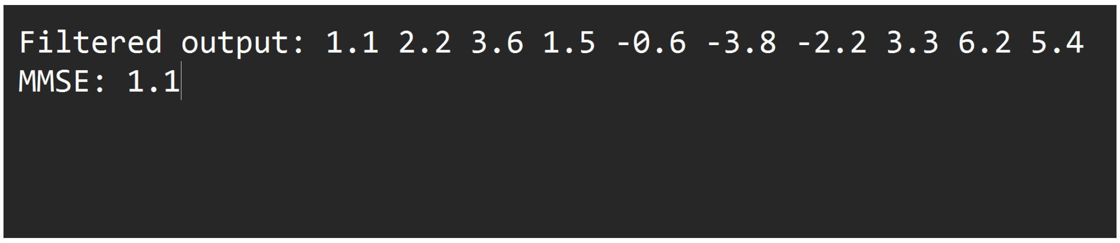
```
procedure float_to_str(buffer, value, decimals):

    # 1) sign
    if value < 0 then
        append '-' to buffer
        value ← -value
    end if

    # 2) scale and round
    if decimals = 1 then
        scale ← 10
    else if decimals = 2 then
        scale ← 100
    else
        scale ← 1
    end if

    temp ← value * scale + 0.5
    N    ← trunc(temp)

    # 3) split integer and fractional digits
    if decimals = 0 then
        intPart ← N
    else if decimals = 1 then
        intPart ← N / 10
        frac1   ← N % 10
    else
        intPart ← N / 100
        frac1   ← (N / 10) % 10
        frac2   ← N % 10
    end if

    # 4) normalize negative zero -> "0.0"
    if intPart = 0 and all frac digits = 0 then
        # ensure we end up printing "0" or "0.0"
    end if

    # 5) write integer part as decimal string
    write intPart into buffer (base 10)

    # 6) write fractional part if needed
    if decimals > 0 then
        append '.' to buffer
        append frac1 (and frac2 if needed) as digits
    end if

end procedure
```

This routine is used in `main` to generate all numeric fields written to `output.txt`, including the list of filtered samples and the MMSE value.

Chapter 4 Examples and Results

4.1 Example 1

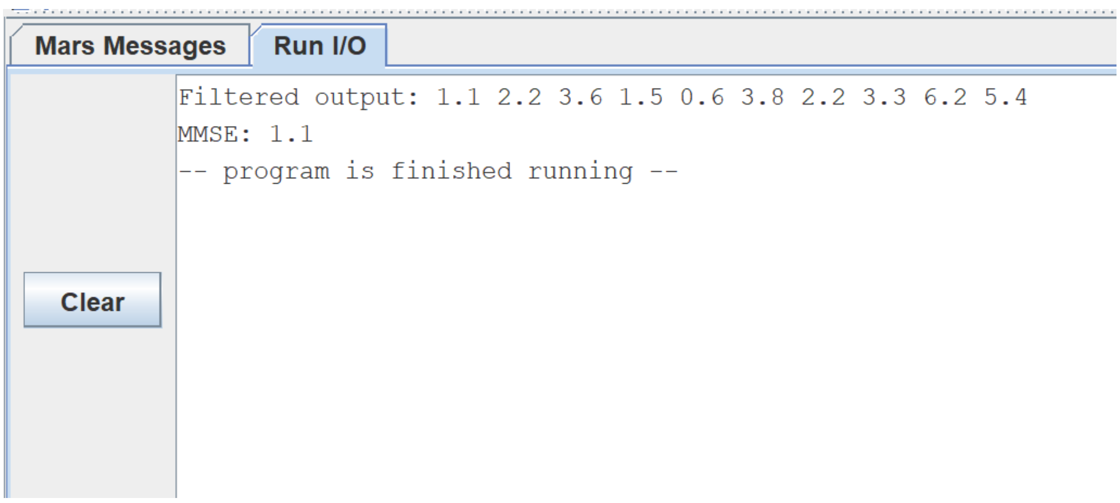
- **Input sequence:** 3.2 2.8 5.9 -2.3 -0.3 -8.3 1.0 9.1 4.6 5.6
- **Desired sequence:** 0.0 3.6 4.6 2.3 -1.0 -2.3 -0.3 3.5 6.3 6.0
- **Expected output (C++ implementation):**



```
Filtered output: 1.1 2.2 3.6 1.5 -0.6 -3.8 -2.2 3.3 6.2 5.4
MMSE: 1.1
```

Expected result for Example 1

- **MIPS output :**

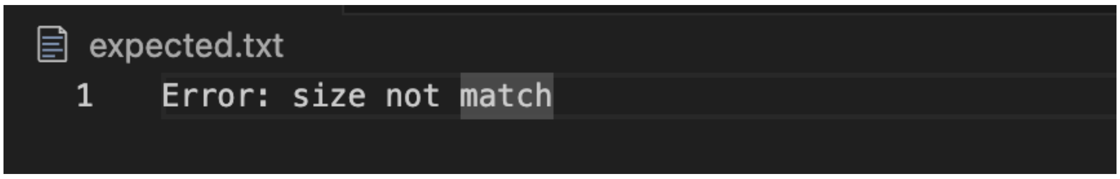


```
Filtered output: 1.1 2.2 3.6 1.5 0.6 3.8 2.2 3.3 6.2 5.4
MMSE: 1.1
-- program is finished running --
```

Result for Example 1

4.2 Example 2

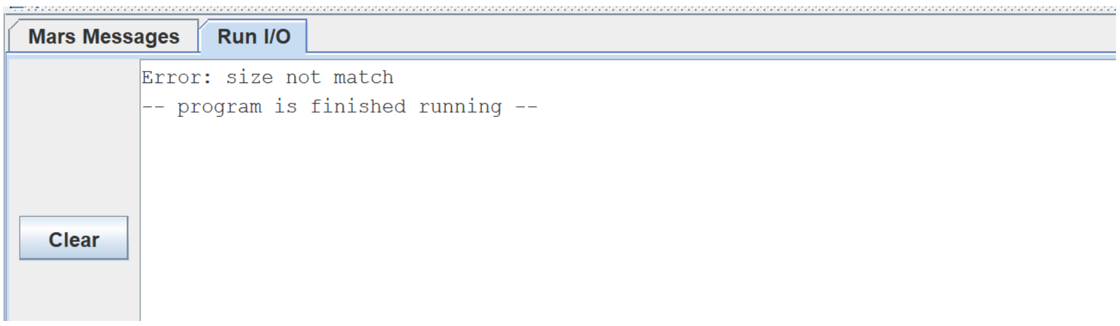
- **Input sequence:** 3.2 2.8 5.9 -2.3 -0.3 -8.3 1.0 9.1 4.6 5.6
- **Desired sequence:** 0.0 3.6 4.6 2.3 -1.0 -2.3 -0.3 3.5 6.3
- **Expected output (C++ implementation):**



```
Error: size not match
```

Expected Result for Example 2

- **MIPS output :**



```
Error: size not match
-- program is finished running --
```

Expected result for Example 2

REFERENCES

[1] Stanford University. *Wiener Filtering (Lecture 12)*, Course EE264: Digital Filtering.