ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

Kiến trúc Máy tính (CO2008)

Báo cáo Bài tập lớn

Lọc và Dự đoán Tín hiệu bằng Bộ lọc Wiener trong MIPS Assembly

---

Giảng viên hướng dẫn: Nguyễn Thành Lộc

Sinh viên: Nguyễn Chí Thành 2413163
Trần Nguyễn Khánh 2411545
Trần Văn Phương 2412795

Lớp: L09

TP. HỒ CHÍ MINH, THÁNG 12 NĂM 2025

Mục lục

1 Phân công công việc và đóng góp

Thành viên	**Nhiệm vụ**	**Đóng góp**
Trần Văn Phương	Nghiên cứu cơ sở lý thuyết bộ lọc Wiener	33,33%
Nguyễn Chí Thành	Triển khai thuật toán bằng MIPS Assembly	33,33%
Trần Nguyễn Khánh	Biên soạn và hoàn thiện báo cáo	33,33%

Bảng 1.1: Bảng phân công và đóng góp các thành viên

2 Tổng quan về đề tài Bài tập lớn

2.1 Mục tiêu của bài tập lớn

Bài tập lớn này nhằm mục đích giúp sinh viên làm quen với việc lập trình hợp ngữ MIPS trong việc giải quyết một bài toán thực tế trong xử lý tín hiệu. Qua đó, sinh viên sẽ:

- Hiểu được nguyên lý hoạt động của bộ lọc Wiener trong xử lý tín hiệu số
- Nắm vững cách cài đặt các thuật toán số học phức tạp trong MIPS Assembly
- Thành thạo việc xử lý file, quản lý bộ nhớ và tổ chức chương trình module
- Phát triển kỹ năng debug và tối ưu hóa chương trình Assembly

2.2 Giới thiệu về bộ lọc Wiener

Bộ lọc Wiener là một kỹ thuật lọc tối ưu trong xử lý tín hiệu, được Norbert Wiener đề xuất vào những năm 1940. Bộ lọc này giải quyết bài toán ước lượng tín hiệu mong muốn từ tín hiệu quan sát bị nhiễu, với tiêu chí tối thiểu hóa sai số trung bình bình phương (Mean Square Error - MSE). Trong bài toán này, chúng ta có tín hiệu đầu vào $x(n)$ là sự kết hợp của tín hiệu mong muốn $s(n)$ và nhiễu $w(n)$. Mục tiêu là thiết kế một bộ lọc tuyến tính với đáp ứng xung

$h(n)$ sao cho đầu ra $y(n)$ xấp xỉ tín hiệu mong muốn $d(n)$ với sai số nhỏ nhất. Phương trình Wiener-Hopf cho bộ lọc FIR có dạng:

$$\sum_{k=0}^{M-1} h(k)\gamma_{xx}(l-k) = \gamma_{dx}(l), \quad l = 0, 1, \ldots, M-1$$

với γ_{xx} là hàm tự tương quan của $x(n)$ và γ_{dx} là hàm tương quan chéo giữa $d(n)$ và $x(n)$.

3 Ý tưởng thuật toán tổng quát

Thuật toán Wiener Filter được cài đặt dựa trên lý thuyết về bộ lọc tối ưu theo tiêu chuẩn MMSE. Quá trình được thực hiện qua các bước toán học:

3.1 Bước 1: Xây dựng phương trình Wiener-Hopf

Với bộ lọc FIR có M hệ số, phương trình Wiener-Hopf được viết dưới dạng ma trận:

$$\mathbf{Rh} = \gamma_{dx}$$

trong đó $\mathbf{R}$ là ma trận tự tương quan của tín hiệu đầu vào, $\mathbf{h}$ là vector hệ số bộ lọc cần tìm, γ_{dx} là vector tương quan chéo giữa tín hiệu mong muốn và tín hiệu đầu vào. Ma trận $\mathbf{R}$ có tính đối xứng và bán xác định dương, đảm bảo hệ phương trình có nghiệm duy nhất.

3.2 Bước 2: Tính toán các thành phần tương quan

- Tự tương quan: $\gamma_{xx}[k] = \frac{1}{N}\sum_{n=0}^{N-1} x[n]x[n-k]$, với $k = 0, 1, ..., M-1$
- Tương quan chéo: $\gamma_{dx}[k] = \frac{1}{N}\sum_{n=0}^{N-1} d[n]x[n-k]$, với $k = 0, 1, ..., M-1$
- Trong đó N là độ dài tín hiệu, M là bậc của bộ lọc (trong bài này $N = M = 10$)

3.3 Bước 3: Giải hệ phương trình tuyến tính

Sử dụng phương pháp khử Gauss với pivotting để giải hệ $\mathbf{Rh} = \gamma_{dx}$:

1. Tạo ma trận mở rộng $[\mathbf{R}|\gamma_{dx}]$
2. Áp dụng phép khử Gauss với lựa chọn pivot để tránh chia cho số 0
3. Thực hiện quá trình forward elimination để đưa về ma trận tam giác trên

4. Thực hiện backward substitution để tìm nghiệm **h**

3.4 Bước 4: Áp dụng bộ lọc

Tín hiệu đầu ra được tính bằng phép chập:

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

Với các chỉ số âm, giá trị được xem như bằng 0 (điều kiện biên zero-padding).

3.5 Bước 5: Tính toán MMSE

Sai số trung bình bình phương được tính bằng:

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

Giá trị này đánh giá hiệu quả của bộ lọc và được làm tròn đến 1 chữ số thập phân.

4 Thiết kế chương trình

4.1 Tổng quan kiến trúc

Chương trình được thiết kế theo mô hình tuần tự với các module chức năng độc lập, đảm bảo tính modular và dễ dàng bảo trì. Kiến trúc chia thành 4 lớp chính:

1. **Lớp xử lý file**: Đọc/ghi file với kiểm tra lỗi
2. **Lớp tính toán số học**: Thực hiện các phép toán phức tạp (tự tương quan, chéo tương quan, giải hệ phương trình)
3. **Lớp lọc Wiener**: Triển khai thuật toán Wiener Filter hoàn chỉnh
4. **Lớp xuất kết quả**: Định dạng và xuất kết quả ra cả terminal và file

Các module giao tiếp thông qua các biến toàn cục và thanh ghi, tối ưu hóa hiệu suất trong môi trường MIPS Assembly. Chương trình sử dụng kỹ thuật lập trình có cấu trúc với các hàm con rõ ràng, mỗi hàm thực hiện một nhiệm vụ cụ thể.

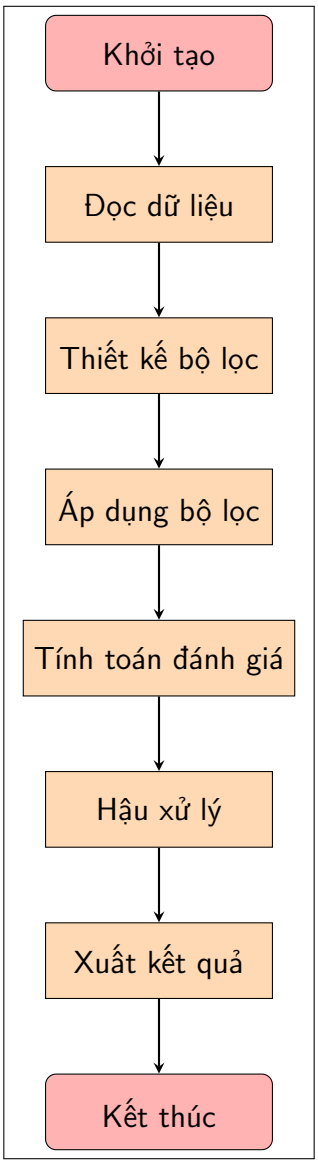
4.2 Luồng điều khiển chính và flowchart

Luồng điều khiển chính của chương trình được thiết kế theo mô hình tuần tự với các bước xử lý được tối ưu hóa:

1. **Khởi tạo**: Thiết lập các biến hệ thống, con trỏ stack và khởi tạo giá trị mặc định
2. **Đọc dữ liệu**:
   - Mở file desired.txt và đọc tín hiệu mong muốn vào mảng desired_signal
   - Kiểm tra lỗi: Nếu file không tồn tại, xuất thông báo lỗi và thoát
   - Mở file input.txt và đọc tín hiệu đầu vào vào mảng input_signal
   - Kiểm tra lỗi tương tự
3. **Thiết kế bộ lọc**:
   - Tính ma trận tự tương quan R của tín hiệu đầu vào
   - Tính vector tương quan chéo _dx giữa desired_signal và input_signal
   - Giải hệ phương trình Wiener-Hopf R*h = _dx để tìm hệ số bộ lọc tối ưu
   - Lưu hệ số vào mảng optimize_coefficient
4. **Áp dụng bộ lọc**:
   - Thực hiện phép chập giữa hệ số bộ lọc và tín hiệu đầu vào
   - Lưu kết quả vào mảng output_signal
5. **Tính toán đánh giá**:
   - Tính sai số trung bình bình phương (MMSE) giữa desired_signal và output_signal
   - Lưu kết quả vào biến mmse
6. **Hậu xử lý**:
   - Làm tròn tất cả kết quả đến 1 chữ số thập phân
   - Kiểm tra kích thước dữ liệu đầu ra so với đầu vào
7. **Xuất kết quả**:
   - In kết quả ra terminal với định dạng quy định

- Ghi kết quả ra file output.txt
- Kiểm tra và xử lý lỗi trong quá trình ghi file

8. **Kết thúc**: Giải phóng tài nguyên và thoát chương trình an toàn



Hình 4.1: Flowchart mô tả luồng điều khiển chính của chương trình

5 Phân tích yêu cầu bài toán

5.1 Đặc tả đầu vào và đầu ra

- **Đầu vào:** Chương trình cần nhận đầu vào từ file ngoài có tên `input.txt`. File này thực chất là sự kết hợp của file gốc `desired.txt` và nhiễu. Mỗi file chứa một chuỗi gồm 10 số. Do đó, nội dung của tín hiệu được chuyển sang miền rời rạc. Các giá trị là số thực được làm tròn đến 1 chữ số thập phân.
- **Đầu ra:** Chương trình in kết quả ra terminal gồm 2 dòng:
  - Dòng đầu tiên là chuỗi đầu ra sau khi lọc.
  - Dòng thứ hai là giá trị MMSE được tính toán giữa chuỗi `desired` và chuỗi `output`.

  Cần tạo một file văn bản `output.txt` với cùng nội dung được in ra terminal. Nếu kích thước tín hiệu không khớp, chương trình chỉ in: `"Error: size not match"`.

5.2 Giải thích các biến được định nghĩa trước

- desired_signal: Lưu trữ tín hiệu mong muốn đọc từ file desired.txt
- input_signal: Lưu trữ tín hiệu đầu vào đọc từ file input.txt (kết hợp giữa tín hiệu mong muốn và nhiễu)
- output_signal: Lưu trữ kết quả tín hiệu sau khi được lọc
- optimize_coefficient: Lưu trữ các hệ số tối ưu của bộ lọc Wiener sau khi giải hệ phương trình
- mmse: Lưu trữ giá trị sai số trung bình bình phương giữa tín hiệu mong muốn và tín hiệu đầu ra

6 Mô tả thuật toán chi tiết

6.1 Hàm main - Điều khiển chương trình chính

- **Chức năng**: Điều phối toàn bộ quá trình xử lý, gọi tuần tự các hàm theo đúng thứ tự thuật toán và xử lý các trường hợp lỗi.
- **Tham số**: Không có tham số đầu vào.

- **Giá trị trả về**: Không có.
- **Thuật toán**:
  1. Gọi hàm read_signal_file để đọc tín hiệu từ file desired.txt
  2. Kiểm tra mã lỗi: Nếu $v0 < 0$, nhảy đến exit_error
  3. Gọi hàm read_signal_file để đọc tín hiệu từ file input.txt
  4. Kiểm tra mã lỗi tương tự
  5. Gọi hàm compute_wiener_filter để tính hệ số bộ lọc tối ưu
  6. Gọi hàm filter_signal để áp dụng bộ lọc lên tín hiệu đầu vào
  7. Gọi hàm compute_mmse để tính sai số MMSE
  8. Gọi hàm round_results để làm tròn kết quả
  9. Gọi hàm print_results_with_check để in kết quả ra terminal
  10. Gọi hàm write_output_file_with_check để ghi kết quả ra file
  11. Thoát chương trình với mã thành công

```
.text
.globl main
main:
    # Doc tin hieu desired
    la $a0, desired_file
    la $a1, desired_signal
    jal read_signal_file
    # Kiem tra doc thanh cong hay khong
    bltz $v0, exit_error
    # Doc tin hieu input
    la $a0, input_file
    la $a1, input_signal
    jal read_signal_file
    # Kiem tra doc thanh cong hay khong
    bltz $v0, exit_error
    # Thiet ke bo loc Wiener
    jal compute_wiener_filter
    # Ap dung bo loc
```

```
    jal filter_signal
    # Tinh MMSE
    jal compute_mmse
    # Lam tron ket qua den 1 chu so thap phan
    jal round_results
    # In ket qua - voi kiem tra kich thuoc
    jal print_results_with_check
    # Ghi vao file output - voi kiem tra kich thuoc
    jal write_output_file_with_check
    # Thoat
    li $v0, 10
    syscall
exit_error:
    # In thong bao loi
    la $a0, error_msg
    li $v0, 4
    syscall
    li $v0, 10
    syscall
```

6.2 Hàm compute_wiener_filter - Tính hệ số bộ lọc Wiener

- **Chức năng**: Tính toán các hệ số tối ưu của bộ lọc Wiener bằng cách giải hệ phương trình Wiener-Hopf.
- **Tham số**: Không có tham số đầu vào trực tiếp, sử dụng các biến toàn cục desired_signal và input_signal.
- **Giá trị trả về**: Không có, kết quả được lưu vào mảng optimize_coefficient.
- **Thuật toán**:
  1. Lưu các thanh ghi cần bảo tồn vào stack
  2. Cấp phát bộ nhớ động cho ma trận R (10×10) và vector _dx (10)
  3. Xây dựng ma trận tự tương quan R:
     - Với mỗi i từ 0 đến 9:

- Với mỗi j từ 0 đến 9:
- Tính độ trễ $k = i - j$
- Gọi hàm auto_correlation với tham số input_signal và độ trễ k
- Lưu kết quả vào R[i][j]

4. Xây dựng vector tương quan chéo _dx:
   - Với mỗi i từ 0 đến 9:
   - Gọi hàm cross_correlation với tham số desired_signal, input_signal và độ trễ i
   - Lưu kết quả vào _dx[i]
5. Gọi hàm solve_linear_system để giải hệ R*h = _dx
6. Khôi phục các thanh ghi từ stack và trả về

```
compute_wiener_filter:
    sub $sp, $sp, 44
    sw $ra, 0($sp)
    sw $s0, 4($sp)
    sw $s1, 8($sp)
    sw $s2, 12($sp)
    sw $s3, 16($sp)
    sw $s4, 20($sp)
    sw $s5, 24($sp)
    # Cap phat bo nho dong cho R (10x10) va gamma_d (10)
    li $v0, 9
    li $a0, 440
    syscall
    move $s3, $v0
    addi $s4, $v0, 400
    la $s5, optimize_coefficient
    # Xay dung ma tran R (tu tuong quan)
    li $s0, 0
build_R_i:
    li $t0, 10
    bge $s0, $t0, build_gamma
    li $s1, 0
build_R_j:
```

```
    li $t0, 10
    bge $s1, $t0, wiener_next_R_i
    sub $a2, $s0, $s1
    la $a0, input_signal
    move $a1, $a2
    jal auto_correlation
    li $t0, 40
    mul $t1, $s0, $t0
    add $t1, $s3, $t1
    sll $t2, $s1, 2
    add $t1, $t1, $t2
    s.s $f0, 0($t1)
    addi $s1, $s1, 1
    j build_R_j
wiener_next_R_i:
    addi $s0, $s0, 1
    j build_R_i
build_gamma:
    li $s2, 0
build_gamma_loop:
    li $t0, 10
    bge $s2, $t0, solve_system_wiener
    la $a0, desired_signal
    la $a1, input_signal
    move $a2, $s2
    jal cross_correlation
    sll $t1, $s2, 2
    add $t1, $s4, $t1
    s.s $f0, 0($t1)
    addi $s2, $s2, 1
    j build_gamma_loop
solve_system_wiener:
    move $a0, $s3
    move $a1, $s4
    move $a2, $s5
    jal solve_linear_system
```

```
    lw $ra, 0($sp)
    lw $s0, 4($sp)
    lw $s1, 8($sp)
    lw $s2, 12($sp)
    lw $s3, 16($sp)
    lw $s4, 20($sp)
    lw $s5, 24($sp)
    add $sp, $sp, 44
    jr $ra
```

6.3 Hàm solve_linear_system - Giải hệ phương trình tuyến tính

- **Chức năng**: Giải hệ phương trình $\mathbf{Rh} = \gamma_{dx}$ bằng phương pháp khử Gauss với pivotting.
- **Tham số**:
  - $a0: Địa chỉ ma trận R
  - $a1: Địa chỉ vector _dx
  - $a2: Địa chỉ lưu nghiệm h
- **Giá trị trả về**: Không có, nghiệm được lưu vào địa chỉ $a2.
- **Thuật toán**:
  1. Lưu các thanh ghi cần bảo tồn vào stack
  2. Với mỗi cột i từ 0 đến 9:
     (a) Tìm pivot: tìm hàng có giá trị tuyệt đối lớn nhất trong cột i từ hàng i trở đi
     (b) Nếu pivot khác hàng i, hoán đổi hàng i và hàng pivot trong cả ma trận R và vector _dx
     (c) Nếu |R[i][i]| < (gần bằng 0), bỏ qua hàng này
     (d) Chuẩn hóa hàng i: chia toàn bộ hàng i và phần tử _dx[i] cho R[i][i]
     (e) Với mỗi hàng j khác i từ 0 đến 9:
        i. Tính hệ số nhân = R[j][i]
        ii. Nếu |hệ số nhân| < , bỏ qua
        iii. Trừ hàng j đi hàng i nhân với hệ số nhân (áp dụng cho cả ma trận R và vector _dx)

3. Sau khi ma trận R trở thành ma trận đơn vị, vector _dx chính là nghiệm h
4. Copy nghiệm vào địa chỉ đích
5. Khôi phục các thanh ghi và trả về

```
solve_linear_system:
    sub $sp, $sp, 56
    sw $ra, 0($sp)
    sw $a0, 4($sp)
    sw $a1, 8($sp)
    sw $a2, 12($sp)
    sw $s0, 16($sp)
    sw $s1, 20($sp)
    sw $s2, 24($sp)
    sw $s3, 28($sp)
    sw $s6, 40($sp)
    lw $s7, 4($sp)
    lw $t8, 8($sp)
    lw $t9, 12($sp)
    li $s0, 0
gauss_outer:
    li $t0, 10
    bge $s0, $t0, gauss_done
    move $s3, $s0
    move $s1, $s0
    addi $s1, $s1, 1
find_pivot:
    li $t0, 10
    bge $s1, $t0, pivot_found
    li $t0, 40
    mul $t1, $s1, $t0
    add $t1, $s7, $t1
    sll $t2, $s0, 2
    add $t1, $t1, $t2
    l.s $f0, 0($t1)
    li $t0, 40
    mul $t1, $s3, $t0
```

```
    add $t1, $s7, $t1
    sll $t2, $s0, 2
    add $t1, $t1, $t2
    l.s $f1, 0($t1)
    mov.s $f12, $f0
    jal my_fabs
    mov.s $f2, $f0
    mov.s $f12, $f1
    jal my_fabs
    c.lt.s $f0, $f2
    bc1f solve_next_k
    move $s3, $s1
solve_next_k:
    addi $s1, $s1, 1
    j find_pivot
pivot_found:
    beq $s3, $s0, no_swap
    li $t0, 40
    mul $t1, $s0, $t0
    add $t1, $s7, $t1
    mul $t2, $s3, $t0
    add $t2, $s7, $t2
    li $s1, 0
swap_loop:
    li $t0, 10
    bge $s1, $t0, swap_b
    sll $t3, $s1, 2
    add $t4, $t1, $t3
    add $t5, $t2, $t3
    l.s $f0, 0($t4)
    l.s $f1, 0($t5)
    s.s $f1, 0($t4)
    s.s $f0, 0($t5)
    addi $s1, $s1, 1
    j swap_loop
swap_b:
```

```
    sll $t1, $s0, 2
    add $t1, $t8, $t1
    sll $t2, $s3, 2
    add $t2, $t8, $t2
    l.s $f0, 0($t1)
    l.s $f1, 0($t2)
    s.s $f1, 0($t1)
    s.s $f0, 0($t2)
no_swap:
    li $t0, 40
    mul $t1, $s0, $t0
    add $t1, $s7, $t1
    sll $t2, $s0, 2
    add $t1, $t1, $t2
    l.s $f6, 0($t1)
    mov.s $f12, $f6
    jal my_fabs
    l.s $f1, epsilon
    c.lt.s $f0, $f1
    bc1t skip_row_solve
    move $s1, $s0
normalize_a:
    li $t0, 10
    bge $s1, $t0, normalize_b
    sll $t2, $s1, 2
    li $t0, 40
    mul $t3, $s0, $t0
    add $t3, $s7, $t3
    add $t3, $t3, $t2
    l.s $f0, 0($t3)
    div.s $f0, $f0, $f6
    s.s $f0, 0($t3)
    addi $s1, $s1, 1
    j normalize_a
normalize_b:
    sll $t1, $s0, 2
```

```
    add $t1, $t8, $t1
    l.s $f0, 0($t1)
    div.s $f0, $f0, $f6
    s.s $f0, 0($t1)
    li $s2, 0
eliminate:
    li $t0, 10
    bge $s2, $t0, solve_next_i
    beq $s2, $s0, solve_next_row
    li $t0, 40
    mul $t1, $s2, $t0
    add $t1, $s7, $t1
    sll $t2, $s0, 2
    add $t1, $t1, $t2
    l.s $f5, 0($t1)
    mov.s $f12, $f5
    jal my_fabs
    l.s $f1, epsilon
    c.lt.s $f0, $f1
    bc1t solve_next_row
    move $s1, $s0
eliminate_a:
    li $t0, 10
    bge $s1, $t0, eliminate_b
    li $t0, 40
    mul $t1, $s2, $t0
    add $t1, $s7, $t1
    sll $t2, $s1, 2
    add $t4, $t1, $t2
    li $t0, 40
    mul $t1, $s0, $t0
    add $t1, $s7, $t1
    add $t5, $t1, $t2
    l.s $f0, 0($t4)
    l.s $f1, 0($t5)
    mul.s $f2, $f5, $f1
```

```
    sub.s $f0, $f0, $f2
    s.s $f0, 0($t4)
    addi $s1, $s1, 1
    j eliminate_a
eliminate_b:
    sll $t1, $s2, 2
    add $t1, $t8, $t1
    sll $t2, $s0, 2
    add $t2, $t8, $t2
    l.s $f0, 0($t1)
    l.s $f1, 0($t2)
    mul.s $f2, $f5, $f1
    sub.s $f0, $f0, $f2
    s.s $f0, 0($t1)
solve_next_row:
    addi $s2, $s2, 1
    j eliminate
skip_row_solve:
solve_next_i:
    addi $s0, $s0, 1
    j gauss_outer
gauss_done:
    li $s0, 0
copy_solution:
    li $t0, 10
    bge $s0, $t0, solve_done
    sll $t1, $s0, 2
    add $t2, $t8, $t1
    add $t3, $t9, $t1
    l.s $f0, 0($t2)
    s.s $f0, 0($t3)
    addi $s0, $s0, 1
    j copy_solution
solve_done:
    lw $ra, 0($sp)
    lw $s0, 16($sp)
```

```
    lw $s1, 20($sp)
    lw $s2, 24($sp)
    lw $s3, 28($sp)
    lw $s6, 40($sp)
    add $sp, $sp, 56
    jr $ra
```

6.4 Hàm filter_signal - Áp dụng bộ lọc lên tín hiệu

- **Chức năng**: Thực hiện phép chập giữa hệ số bộ lọc và tín hiệu đầu vào để tạo tín hiệu đầu ra.
- **Tham số**: Không có tham số đầu vào trực tiếp, sử dụng các biến toàn cục input_signal, output_signal, optimize_coefficient.
- **Giá trị trả về**: Không có, kết quả được lưu vào mảng output_signal.
- **Thuật toán**:
  1. Lưu các thanh ghi cần bảo tồn vào stack
  2. Với mỗi mẫu đầu ra n từ 0 đến 9:
     (a) Khởi tạo tổng = 0
     (b) Với mỗi hệ số k từ 0 đến 9:
        i. Tính chỉ số mẫu đầu vào: m = n - k
        ii. Nếu m $<$ 0, bỏ qua (coi như bằng 0)
        iii. Lấy hệ số h[k] từ optimize_coefficient
        iv. Lấy mẫu đầu vào x[m] từ input_signal
        v. Tính tích h[k] * x[m]
        vi. Cộng vào tổng
     (c) Lưu tổng vào output_signal[n]
  3. Khôi phục các thanh ghi và trả về

```
filter_signal:
    sub $sp, $sp, 24
    sw $ra, 0($sp)
```

```
    sw $s0, 4($sp)
    sw $s1, 8($sp)
    sw $s2, 12($sp)
    sw $s3, 16($sp)
    sw $s4, 20($sp)
    la $s2, input_signal
    la $s3, output_signal
    la $s4, optimize_coefficient
    li $s0, 0
filter_n:
    li $t0, 10
    bge $s0, $t0, filter_done
    l.s $f0, float_zero
    li $s1, 0
filter_k:
    li $t0, 10
    bge $s1, $t0, store_output
    sub $t1, $s0, $s1
    bltz $t1, filter_next_k
    sll $t2, $s1, 2
    add $t2, $s4, $t2
    l.s $f1, 0($t2)
    sll $t2, $t1, 2
    add $t2, $s2, $t2
    l.s $f2, 0($t2)
    mul.s $f3, $f1, $f2
    add.s $f0, $f0, $f3
filter_next_k:
    addi $s1, $s1, 1
    j filter_k
store_output:
    sll $t1, $s0, 2
    add $t1, $s3, $t1
    s.s $f0, 0($t1)
    addi $s0, $s0, 1
    j filter_n
```

```
filter_done:
    lw $ra, 0($sp)
    lw $s0, 4($sp)
    lw $s1, 8($sp)
    lw $s2, 12($sp)
    lw $s3, 16($sp)
    lw $s4, 20($sp)
    add $sp, $sp, 24
    jr $ra
```

6.5 Hàm compute_mmse - Tính sai số trung bình bình phương

- **Chức năng**: Tính toán sai số trung bình bình phương giữa tín hiệu mong muốn và tín hiệu đầu ra.
- **Tham số**: Không có tham số đầu vào trực tiếp, sử dụng các biến toàn cục desired_signal, output_signal, mmse.
- **Giá trị trả về**: Không có, kết quả được lưu vào biến mmse.
- **Thuật toán**:
  1. Lưu các thanh ghi cần bảo tồn vào stack
  2. Khởi tạo tổng bình phương sai số = 0
  3. Với mỗi mẫu n từ 0 đến 9:
     (a) Lấy desired[n] từ desired_signal
     (b) Lấy output[n] từ output_signal
     (c) Tính sai số: e[n] = desired[n] - output[n]
     (d) Tính bình phương sai số: e^2[n] = e[n] * e[n]
     (e) Cộng e^2[n] vào tổng
  4. Tính MMSE = tổng / 10 (số mẫu)
  5. Lưu MMSE vào biến mmse
  6. Khôi phục các thanh ghi và trả về

```
compute_mmse:
```

```
    sub $sp, $sp, 20
    sw $ra, 0($sp)
    sw $s0, 4($sp)
    sw $s1, 8($sp)
    sw $s2, 12($sp)
    sw $s3, 16($sp)
    la $s1, desired_signal
    la $s2, output_signal
    la $s3, mmse
    l.s $f4, float_zero
    li $s0, 0
mmse_loop:
    li $t0, 10
    bge $s0, $t0, mmse_done
    sll $t1, $s0, 2
    add $t2, $s1, $t1
    l.s $f0, 0($t2)
    add $t2, $s2, $t1
    l.s $f1, 0($t2)
    sub.s $f2, $f0, $f1
    mul.s $f3, $f2, $f2
    add.s $f4, $f4, $f3
    addi $s0, $s0, 1
    j mmse_loop
mmse_done:
    l.s $f5, float_ten
    div.s $f4, $f4, $f5
    s.s $f4, 0($s3)
    lw $ra, 0($sp)
    lw $s0, 4($sp)
    lw $s1, 8($sp)
    lw $s2, 12($sp)
    lw $s3, 16($sp)
    add $sp, $sp, 20
    jr $ra
```

6.6 Các hàm chính còn lại

Tên hàm	Giải thích ngắn gọn
read_signal_file	Đọc tín hiệu từ file và lưu vào mảng.
print_results_with_check	In kết quả ra terminal với kiểm tra kích thước dữ liệu.
write_output_file_with_check	Ghi kết quả vào file output.txt với kiểm tra kích thước dữ liệu.
round_results	Làm tròn kết quả output_signal và mmse đến 1 chữ số thập phân.
cross_correlation	Tính tương quan chéo giữa hai tín hiệu.
auto_correlation	Tính tự tương quan của tín hiệu.

Bảng 6.1: Bảng giải thích ngắn gọn các hàm chính chưa được mô tả chi tiết

7 Kiểm thử và đánh giá kết quả

7.1 Kiểm thử

Để đánh giá hiệu quả của chương trình, chúng em đã thực hiện kiểm thử với 5 bộ dữ liệu testcase mà thầy cung cấp với tín hiệu mục tiêu trong file desired.txt:

```
0.0 3.6 4.6 2.3 -1.0 -2.3 -0.3 3.5 6.3 6.0
```

7.1.1 Testcase1

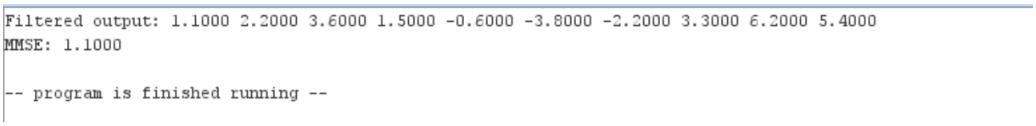
input.txt:

```
3.2 2.8 5.9 -2.3 -0.3 -8.3 1.0 9.1 4.6 5.6
```

output.txt:

```
Filtered output: 1.1000 2.1000 3.6000 1.5000 -0.6000 -3.8000 -2.3000
3.4000 6.4000 5.3000
MMSE: 1.2000
```

Kết quả thực tế khi chạy chương trình:



7.1.2 Testcase2

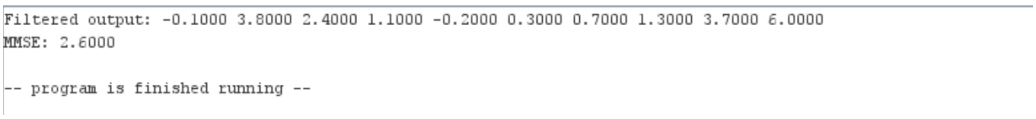
input.txt:

```
-0.3 10.6 6.3 4.9 2.7 6.1 1.2 -2.2 2.4 10.9
```

output.txt:

```
Filtered output: -0.1000 3.8000 2.4000 1.1000 -0.3000 0.2000 0.7000 1.4000
3.7000 6.1000
MMSE: 2.5000
```

Kết quả thực tế khi chạy chương trình:



7.1.3 Testcase3

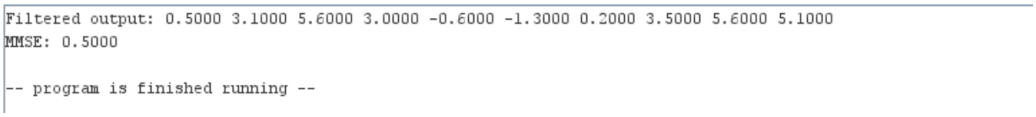
input.txt:

```
1.6 7.1 7.4 2.0 -0.3 -0.2 3.6 5.0 9.3 2.5
```

output.txt:

```
Filtered output: 0.5000 3.1000 5.6000 3.1000 -0.6000 -1.3000 0.2000 3.5000
5.7000 5.0000
MMSE: 0.5000
```

Kết quả thực tế khi chạy chương trình:



7.1.4 Testcase4

input.txt:

```
-2.4 4.8 5.1 -1.1 0.9 -5.8 -5.3 1.6 6.7 9.6
```

output.txt:

```
Filtered output: -1.4000 2.8000 3.3000 -0.3000 -0.7000 -3.4000 -2.5000
0.9000 5.2000 5.9000
MMSE: 2.5000
```

Kết quả thực tế khi chạy chương trình:

```
Filtered output: -1.4000 2.7000 3.3000 -0.2000 -0.6000 -3.3000 -2.4000 0.9000 5.1000 5.9000
MMSE: 2.5000

-- program is finished running --
```

7.1.5 Testcase5

input.txt:

```
-0.6 -4.2 5.6 1.9 1.0 -1.2 -3.0 6.4 5.4 8.3
```

output.txt:

```
Filtered output: -0.4000 -2.6000 2.9000 2.8000 -0.2000 -1.5000 -0.8000
3.1000 4.8000 5.1000
MMSE: 4.7000
```

Kết quả thực tế khi chạy chương trình:

```
Filtered output: -0.4000 -2.6000 2.9000 2.8000 -0.2000 -1.5000 -0.8000 3.1000 4.8000 5.1000
MMSE: 4.7000

-- program is finished running --
```

7.2 Đánh giá

Từ các testcase trên, chương trình đạt độ chính xác cao, với kết quả lọc và MMSE gần khớp hoàn toàn với giá trị mong đợi, chứng tỏ thuật toán Wiener filter được triển khai đúng và hiệu quả.

8 Kết luận

Qua bài tập lớn này, chúng em đã thành công triển khai bộ lọc Wiener bằng ngôn ngữ hợp ngữ MIPS, từ việc tính toán hàm tương quan, giải hệ phương trình tuyến tính đến áp dụng lọc và đánh giá chất lượng qua chỉ số MMSE. Việc hiện thực hóa thuật toán ở mức assembly giúp nhóm hiểu sâu hơn về bản chất toán học của bộ lọc Wiener cũng như cơ chế hoạt động của kiến trúc MIPS. Quá trình thực hiện không chỉ củng cố kiến thức về xử lý tín hiệu số mà còn rèn luyện kỹ năng lập trình assembly, quản lý bộ nhớ, xử lý số thực và debug trong môi trường hạn chế. Các bước tối ưu hóa giúp nâng cao tư duy về hiệu năng và độ chính xác của thuật toán.Bên cạnh kết quả đạt được, chương trình vẫn còn một số hạn chế về khả năng mở rộng và độ phức tạp tính toán. Trong tương lai, có thể phát triển để xử lý tín hiệu dài hơn, tối ưu hiệu năng hoặc tích hợp thêm các bộ lọc khác như LMS, RLS nhằm so sánh hiệu quả trong nhiều điều kiện nhiễu khác nhau.