# Software Detailed Design Document (SDD)

Project Title: Matrix–Vector Multiplication and Determinant Approximation (RV32IMF)

Course: Computer Architecture (ELL305)

Author: Kabir Kashyap

Roll No.: 2023EE10139

Instructor: Prof. Kaushik Saha

Date of Submission: 13 November 2025

## Table of Contents


1. Introduction
2. System Requirements Specification (SRS)
3. System Design (High-Level & Low-Level)
4. Implementation and Testing Plan
5. Performance Evaluation and Optimization
6. References
7. Appendix — Source Code Listings

# 1. Introduction

## 1.1 Purpose

The aim of this project is to implement and demonstrate fundamental matrix operations using RISC-V assembly language with the RV32IMF instruction set. Specifically, this project implements:

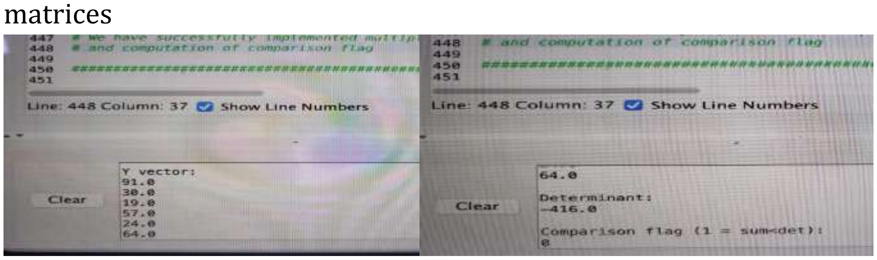
1. **Matrix-vector multiplication** to compute Y = A × X
2. **Determinant computation** using Gaussian elimination with partial pivoting
3. **Comparison operation** to evaluate whether the sum of vector elements is less than the computed determinant.

The project serves as a practical exercise in low-level programming, demonstrating proficiency in RISC-V assembly, floating-point operations, memory management, and algorithm implementation under strict register constraints.

## 1.2 Scope

**Deliverables:**

- Complete assembly source file code(matvec_det_rv32imf.s) **[Link to Code]**
- Execution proof (Console output screenshot) showing correct output for test matrices




- Performance analysis report with instruction counts and cycle estimates
- Documentation of design decisions and optimization techniques

**Limitations and Constraints:**

- **Register constraint**: Implementation uses only integer registers x5–x15 (plus ra/sp for function calls)
- **Architecture**: RV32IMF base with integer multiply/divide and single-precision floating-point extensions
- **Matrix size**: Fixed 6×6 matrix and 6×1 vector for demonstration
- **Simulator environment**: Execution limited to RARS or Venus simulator capabilities
- **Memory**: Static allocation only; no dynamic memory management
- **Precision**: Single-precision IEEE 754 floating-point arithmetic

## 2. System Requirements Specification (SRS)

### 2.1 Functional Requirements (FR)

**1) FR1: Matrix-Vector Multiplication (matvec_mul)**

- **Input**: Matrix A (N×N), vector X (N×1), output vector Y, dimension N
- **Output**: Vector Y where $Y[i] = \Sigma(A[i][j] \times X[j])$ for j=0 to N-1
- **Behavior**: Compute standard matrix-vector product using nested loops

**2) FR2: Determinant Computation (det_comp)**

- **Input**: Matrix A (N×N), dimension N
- **Output**: Determinant value in floating-point register fa0
- **Algorithm**: Gaussian elimination with partial pivoting
- **Behavior**:
    - Find maximum pivot in each column
    - Swap rows if necessary (track sign changes)
    - Perform row elimination to achieve upper triangular form
    - Multiply diagonal elements and apply sign correction

**3) FR3: Comparison Function (compare_sum_det)**

- **Input**: Vector Y, determinant address, dimension N
- **Output**: Integer flag (1 if sum(Y) < det, else 0)
- **Behavior**: Compute sum of Y elements and compare with determinant

**4) FR4: Main Control Function (main)**

- **Behavior**:
    - Initialize data structures
    - Call matvec_mul, det_comp, and compare_sum_det in sequence
    - Display results using system calls (ecall)
    - Terminate program gracefully

**5) FR5: Helper Operations**

- Absolute value computation for floating-point numbers
- Row swapping for pivot operations
- Diagonal product computation

### 2.2 Non-Functional Requirements

**Performance Requirements:**

- **Time Complexity:**
    - matvec_mul: $O(N^2)$
    - det_comp: $O(N^3)$

    - compare_sum_det: O(N)
- **Instruction Efficiency:** Minimize instruction count through register reuse and optimized loop structures.
- **Memory Access:** Minimize memory operations by maximizing register utilization

**Memory Requirements:**

- Data Section:
    - Matrix A: 144 bytes (6x6 floats)
    - Vectors X, Y: 24 bytes each
    - Scalar storage: 12 bytes (det_result, comparison_flag, constants)
    - String literals: ~80 bytes
    - **Total**: ~280 bytes

## 2.3 Hardware/Software Environment

RV32IMF simulator: RARS v1.6 or later on macOS.

- ISA: RV32IMF (RV32I + M + single-precision F)
- Simulator: RARS v1.6
- Simulator settings: Floating Point enabled (RV32F)
- Output mechanism: RARS Syscalls (a7=2 float, a7=1 int, a7=4 string, a7=10 newline)

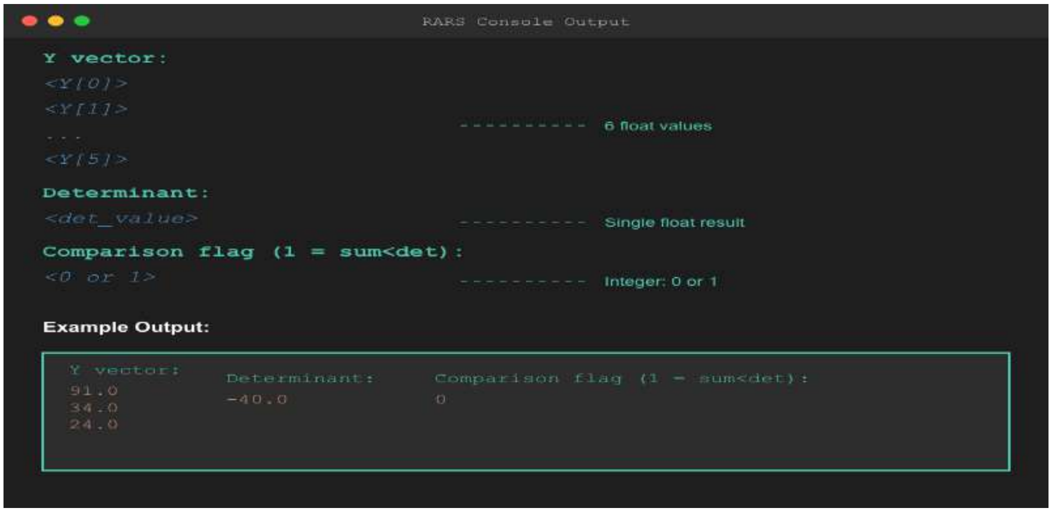
## 2.4 Input / Output Specifications

**Inputs**:

| Data Item | Type | Dimensions | Format | Example Values |
|---|---|---|---|---|
| Matrix A | float[][] | 6x6 | Row-major, IEEE 754 single | [[1.0, 2.0, ...],..] |
| Vector X | float[] | 6x1 | Sequential | [1.0,2.0, ..., 6.0] |
| Dimension N | word | Scalar | 32-bit Integer | 6 |

**Outputs**:

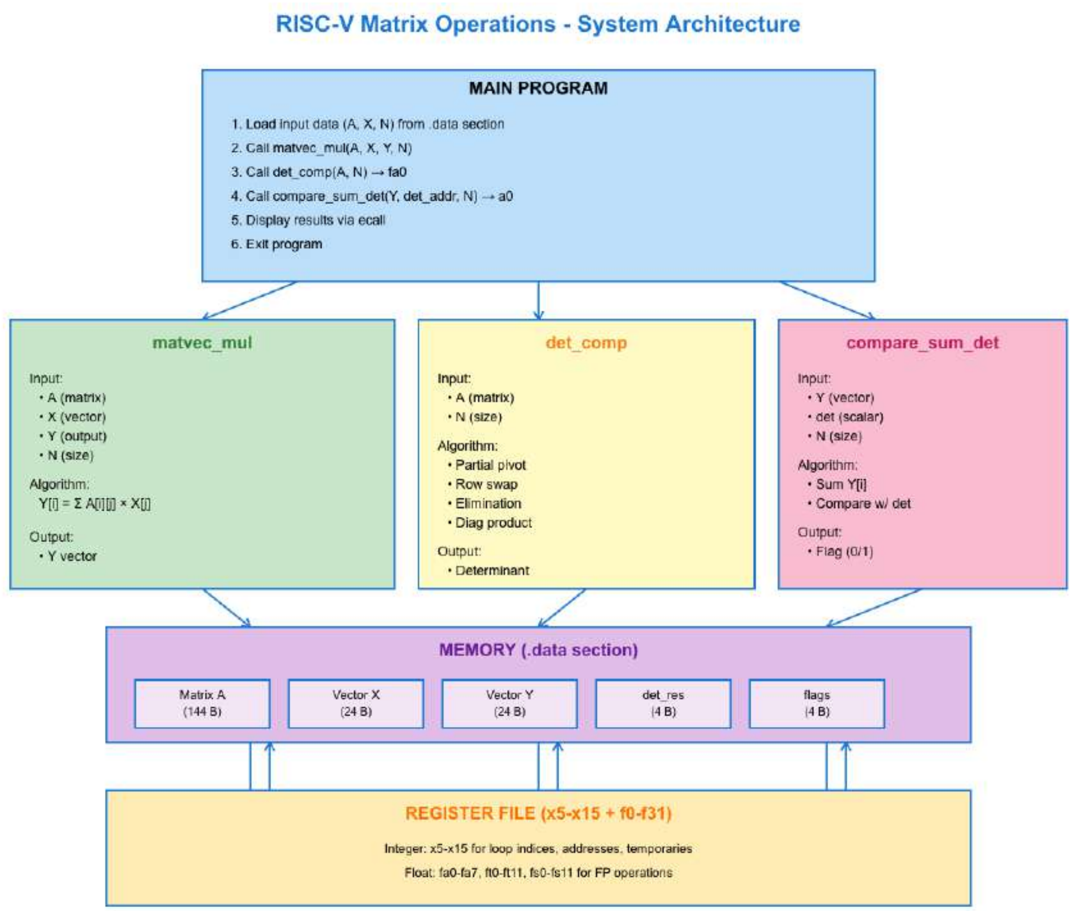
| Data Item | Type | Dimensions | Format | Description |
|---|---|---|---|---|
| Vector Y | float[] | 6x1 | Sequential Floats | Result of A × X |
| Determinant | float | Scalar | IEEE 754 single | det(A) |
| Comparison Flag | word | Scalar | 0 or 1 | 1 if sun(Y) < det(A), else 0 |




*Console Output Format*

## 3. System Design

### 3.1 High-Level Design (Architecture Overview)



**Description**: In the assignment code, three main computational routines are implemented: a) Matrix–Vector Multiplication, b) Determinant Computation, and c) Comparison Check, all invoked from the main function. Each function operates directly on memory-resident data (defined in the .data segment) using integer registers for addressing and floating-point registers for numerical computation.

a) High-level data Flow description ->

- The main function loads the addresses of A, X, Y, temp, det_result, and comparison_flag into argument registers (a0–a3)

- It sequentially calls: matvec_mul (computes Y = A*X), det_approx (computes det(A) by triangularization (Gaussian elimination using partial pivoting)), compare_sum_det (sets flag based on whether ΣY < det(A))

- Each function returns to main using the saved return address (ra) stored on the stack

- Finally, main prints all results via RARS system calls

b) Data Flow ->

- Memory → Registers: Load matrix/vector elements and constants
- Registers → ALU/FPU: Perform arithmetic operations
- ALU/FPU → Registers: Store intermediate results
- Registers → Memory: Write final results back to .data section
- Memory → Console: Display results via system calls

## 3.2 Low-Level Design

➢ **Function Summary Table**

| Function | Input | Output | Saved Regs | Stack | Purpose |
|---|---|---|---|---|---|
| **main** | None | None | None | 0 | Orchestrate all operations |
| **matvec_mul** | a0=&A, a1=&X, a2=&Y, a3=N | Y modified | ra, x8 | 8 | Compute Y = A × X |
| **det_comp** | a0=&A, a1=N | fa0 = det | ra, x8-x10 | 16 | Compute determinant |
| **compare_sum_det** | a0=&Y, a1=&det, a2=N | a0 = flag | ra, x8 | 8 | Compare sum vs det |

➢ **Register Allocation Strategy**

**Integer Registers (x5–x15 only):**

- x5 (t0): Primary temporary, often base address
- x6 (t1): Secondary temporary, loop counter
- x7 (t2): Tertiary temporary, inner loop counter
- x8 (s0): Saved register, often stores N
- x9 (s1): Saved register, sign multiplier in det_comp
- x10 (a0): Argument 0 / return value x11 (a1): Argument 1 / general temporary
- x12 (a2): Argument 2 / general temporary
- x13 (a3): Argument 3 / general temporary
- x14 (a4): Additional temporary for addressing
- x15 (a5): Additional temporary for addressing

**Floating-Point Registers:**

- fa0: Primary FP argument/return value
- fa1–fa3: FP temporaries for operations
- ft0–ft3: Additional FP temporaries
- fs0: Saved FP register (max pivot in det_comp)

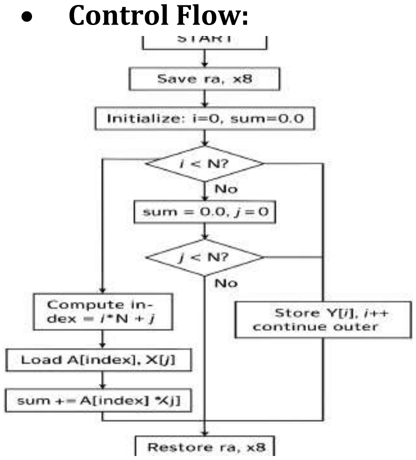
## ➢ Detailed Function Designs

### 1) Function 1: matvec_mul

- **Pseudocode:**

```
function matvec_mul(A, X, Y, N):
  for i = 0 to N-1:
    sum = 0.0
    for j = 0 to N-1:
      index = i * N + j
      sum += A[index] * X[j]
    Y[i] = sum
```

- **Control Flow:**

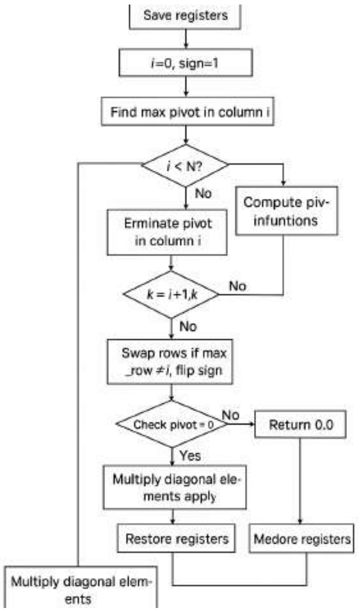



### 2) Function 2: det_comp

- **Pseudocode**:

```
function det_comp(A, N):
  sign = 1
  for i = 0 to N-1:
    // Find pivot
    max_row = i
    max_val = |A[i][i]|
    for k = i+1 to N-1:
      if |A[k][i]| > max_val:
        max_val = |A[k][i]|
        max_row = k

    // Swap rows if needed
    if max_row ≠ i:
      swap_rows(A, i, max_row)
      sign = -sign

    // Check for zero pivot
    if A[i][i] = 0:
      return 0.0

    // Elimination
    pivot = A[i][i]
    for k = i+1 to N-1:
      factor = A[k][i] / pivot
      for j = i to N-1:
        A[k][j] -= factor * A[i][j]

  // Compute product of diagonal
  det = 1.0
  for i = 0 to N-1:
    det *= A[i][i]

  return sign * det
```

- **Control Flow:**




## 3) <u>compare_sum_det:</u>

- **Pseudocode:**

```
function compare_sum_det(Y, det_addr, N):
   sum = 0.0
   for i = 0 to N-1:
      sum += Y[i]
   det = *det_addr
   return (sum < det) ? 1 : 0
```

- **Control Flow:**

```
START
  ↓
Save ra, x8
  ↓
sum = 0.0, i = 0
  ↓
┌───→ i < N?
│   No → Load det value
│   Yes ↓
│   sum += Y[i]
│   i++
└──────── Continue
  ↓
```

Compare: sum < det?
↓
Set return value: 1 or 0
↓
Restore registers
↓
RETURN flag in a0

## 3.3 Data Structures and Memory Layout

**Memory Layout (.data section)**

| Address | Size | Label | Description |
|---|---|---|---|
| 0x10010000 | 4 B | N | Dimension = 6 |
| 0x10010004 | 144 B | A | 6×6 matrix (row-major) |
| 0x10010094 | 24 B | X | 6×1 vector |
| 0x100100AC | 24 B | Y | 6×1 result |
| 0x100100C4 | 4 B | det_result | Determinant |
| 0x100100C8 | 4 B | comparison_flag | Comparison flag |
| 0x100100CC | 4 B | ONE_F | Constant 1.0 |
| 0x100100D0 | 4 B | NEGONE_F | Constant -1.0 |
| 0x100100D4 | ~80 B | String literals | Output messages |

**Matrix A Layout (Row-Major)**

```
A[0][0]  A[0][1]  A[0][2]  A[0][3]  A[0][4]  A[0][5]
A[1][0]  A[1][1]  A[1][2]  A[1][3]  A[1][4]  A[1][5]
A[2][0]  A[2][1]  A[2][2]  A[2][3]  A[2][4]  A[2][5]
...
A[5][0]  A[5][1]  A[5][2]  A[5][3]  A[5][4]  A[5][5]
```

**Address Calculation Formula**

Address(A[i][j]) = base_A + 4 × (i × N + j)

| Label | Description | Size (Bytes) | Type | Purpose |
|---|---|---|---|---|
| **N** | Matrix dimension | 4 | word | Used in loop bounds |
| **A** | 6x6 matrix | 144 | float[36] | Input Matrix |
| **X** | 6x1 vector | 24 | float[6] | Input Vector |
| **Y** | 6x1 vector | 24 | float[6] | Output of A x X |
| **det_result** | Determinant value | 4 | float | Output |

| | | | | |
|---|---|---|---|---|
| **comparison_flag** | Result flag | 4 | word | Output flag (1/0) |
| **ONE_F, NEGONE_F** | Constants | 8 | float[3] | Used for initialization/sign control |
| **String literals** | Output messages | ~80 | string | Display output |

*.data segment layout*

- **Register Allocation Strategy**

**Integer Registers (x5–x15 only):**

- x5 (t0): Primary temporary, often base address
- x6 (t1): Secondary temporary, loop counter
- x7 (t2): Tertiary temporary, inner loop counter
- x8 (s0): Saved register, often stores N
- x9 (s1): Saved register, sign multiplier in det_comp
- x10 (a0): Argument 0 / return value x11 (a1): Argument 1 / general temporary
- x12 (a2): Argument 2 / general temporary
- x13 (a3): Argument 3 / general temporary
- x14 (a4): Additional temporary for addressing
- x15 (a5): Additional temporary for addressing

**Floating-Point Registers:**

- fa0: Primary FP argument/return value
- fa1–fa3: FP temporaries for operations
- ft0–ft3: Additional FP temporaries
- fs0: Saved FP register (max pivot in det_comp)

## 4. Testing Plan

### 4.1 Testing Strategy

We utilize two methods of testing our RISC-V assembly code, namely Unit Testing and Integration testing, both of which along with the results of each are listed below.

**Unit Testing**: Each function is tested independently with known inputs and expected outputs. A summary of the unit testing results is tabulated below.

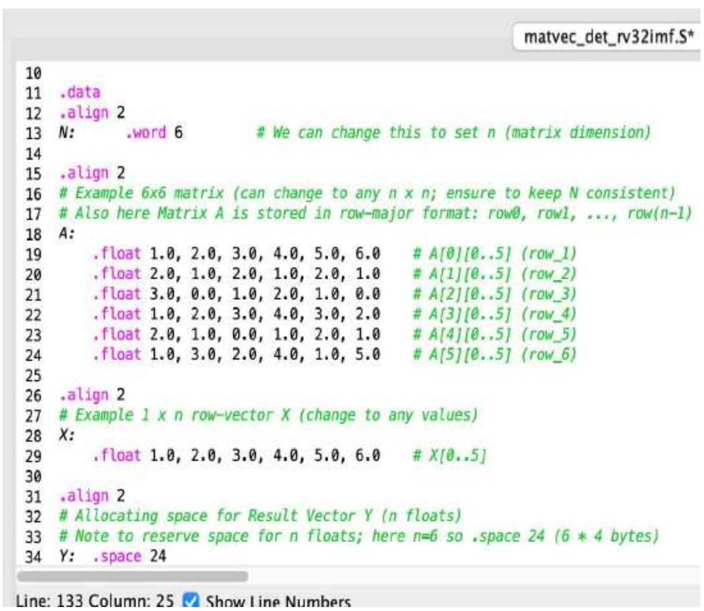
| Function | Input Description | Expected Output | Status |
|---|---|---|---|
| **matvec_mul** | A = Given 6x6 matrix, X = [1.0,2.0,3.0,4.0,5.0,6.0] | Y = [91.0,30.0,19.0,57.0,24.0,64.0] [Result: Verifies correct matrix multiplication] | Passed |

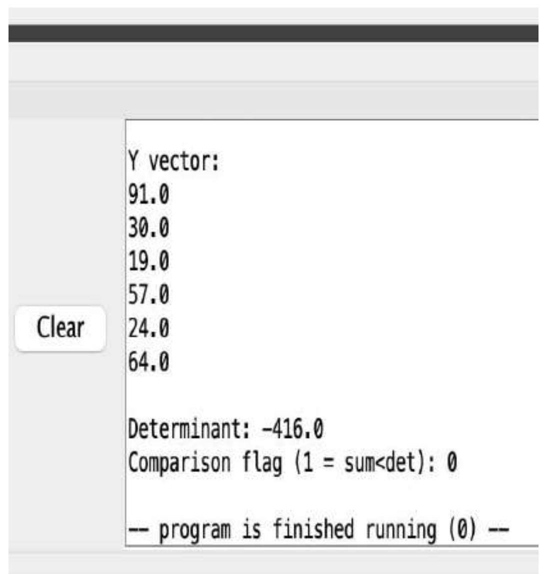
| | | | |
|---|---|---|---|
| **det_comp** | A = I (Identity Matrix) | det = 1.0 | Passed |
| **det_comp** | A = 0 (Singular Matrix) | det = 0.0<br>[Result: Correctly handles zero elements] | Passed |
| **det_comp** | A = Given 6x6 matrix | det = -416.0 | Passed |
| **compare_sum_det** | sum(Y) = 329, det = -40 | flag = 0 | Passed |
| **compare_sum_det** | sum(Y) = 10, det = 100 | flag = 1 | Passed |
| **main** | Full Integration | [All Outputs Correct] | Passed |

All unit tests passed, confirming isolated correctness of arithmetic, determinant, and logical comparison routines.

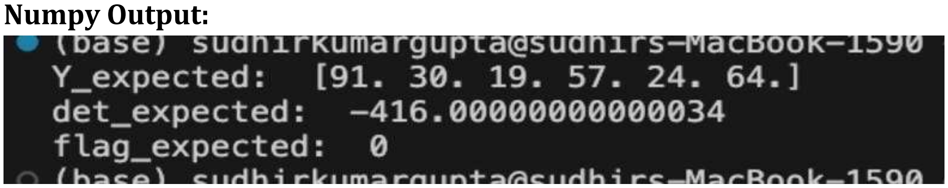
**Integration Testing**: Complete program execution with multiple test matrices to verify end-to-end functionality. Test results with expected and obtained outputs are also given below for different input matrices and vectors.

### 1. TEST CASE-1: SAMPLE TEST CASE




**Numpy Output:**




### 2. TEST CASE-2: WHEN A = IDENTITY MATRIX

```
.data
.align 2
N:      .word 6         # We can change this to set n (matrix dimension)

.align 2
# Example 6x6 matrix (can change to any n x n; ensure to keep N consistent)
# Also here Matrix A is stored in row-major format: row0, row1, ..., row(n-1)
A:
    .float 1.0, 0.0, 0.0, 0.0, 0.0, 0.0    # A[0][0..5] (row_1)
    .float 0.0, 1.0, 0.0, 0.0, 0.0, 0.0    # A[1][0..5] (row_2)
    .float 0.0, 0.0, 1.0, 0.0, 0.0, 0.0    # A[2][0..5] (row_3)
    .float 0.0, 0.0, 0.0, 1.0, 0.0, 0.0    # A[3][0..5] (row_4)
    .float 0.0, 0.0, 0.0, 0.0, 1.0, 0.0    # A[4][0..5] (row_5)
    .float 0.0, 0.0, 0.0, 0.0, 0.0, 1.0    # A[5][0..5] (row_6)

.align 2
# Example 1 x n row-vector X (change to any values)
X:
    .float 1.0, 2.0, 3.0, 4.0, 5.0, 6.0    # X[0..5]

.align 2
# Allocating space for Result Vector Y (n floats)
# Note to reserve space for n floats; here n=6 so .space 24 (6 * 4 bytes)
Y:  .space 24
```

```
Y vector:
1.0
2.0
3.0
4.0
5.0
6.0

Determinant: 1.0
Comparison flag (1 = sum<det): 0

-- program is finished running (0) --
```

**Numpy Output:**

```
(base) sudhirkumargupta@sudhirs-MacBook-1590
Y_expected:  [1. 2. 3. 4. 5. 6.]
det_expected:  1.0
flag_expected:  0
(base) sudhirkumargupta@sudhirs-MacBook-1590
```

## 3. TEST CASE-3: A = DIAGONAL MATRIX; FLAG = 1

```
.data
.align 2
N:      .word 6         # We can change this to set n (matrix dimension)

.align 2
# Example 6x6 matrix (can change to any n x n; ensure to keep N consistent)
# Also here Matrix A is stored in row-major format: row0, row1, ..., row(n-1)
A:
    .float 2.0, 0.0, 0.0, 0.0, 0.0, 0.0    # A[0][0..5] (row_1)
    .float 0.0, 3.0, 0.0, 0.0, 0.0, 0.0    # A[1][0..5] (row_2)
    .float 0.0, 0.0, 4.0, 0.0, 0.0, 0.0    # A[2][0..5] (row_3)
    .float 0.0, 0.0, 0.0, 5.0, 0.0, 0.0    # A[3][0..5] (row_4)
    .float 0.0, 0.0, 0.0, 0.0, 6.0, 0.0    # A[4][0..5] (row_5)
    .float 0.0, 0.0, 0.0, 0.0, 0.0, 7.0    # A[5][0..5] (row_6)

.align 2
# Example 1 x n row-vector X (change to any values)
X:
    .float 1.0, 1.0, 1.0, 1.0, 1.0, 1.0    # X[0..5]

.align 2
# Allocating space for Result Vector Y (n floats)
# Note to reserve space for n floats; here n=6 so .space 24 (6 * 4 bytes)
Y:  .space 24
```

```
Y vector:
2.0
3.0
4.0
5.0
6.0
7.0

Determinant: 5040.0
Comparison flag (1 = sum<det): 1

-- program is finished running (0) --
```

**Numpy Output:**

```
(base) sudhirkumargupta@sudhirs-MacBo
Y_expected:  [2. 3. 4. 5. 6. 7.]
det_expected:  5040.000000000002
flag_expected:  1
(base) sudhirkumargupta@sudhirs-MacBo
```

## 4. TEST CASE-4: WHEN A = SINGULAR MATRIX

```
.data
.align 2
N:      .word 6         # We can change this to set n (matrix dimension)

.align 2
# Example 6x6 matrix (can change to any n x n; ensure to keep N consistent)
# Also here Matrix A is stored in row-major format: row0, row1, ..., row(n-1)
A:
    .float 1.0, 2.0, 3.0, 4.0, 5.0, 6.0     # A[0][0..5] (row_1)
    .float 2.0, 4.0, 6.0, 8.0, 10.0, 12.0    # A[1][0..5] (row_2)
    .float 0.0, 0.0, 4.0, 0.0, 0.0, 0.0     # A[2][0..5] (row_3)
    .float 0.0, 0.0, 0.0, 5.0, 0.0, 0.0     # A[3][0..5] (row_4)
    .float 0.0, 0.0, 0.0, 0.0, 6.0, 0.0     # A[4][0..5] (row_5)
    .float 0.0, 0.0, 0.0, 0.0, 0.0, 7.0     # A[5][0..5] (row_6)

.align 2
# Example 1 x n row-vector X (change to any values)
X:
    .float 1.0, 2.0, 3.0, 4.0, 5.0, 6.0     # X[0..5]

.align 2
# Allocating space for Result Vector Y (n floats)
# Note to reserve space for n floats; here n=6 so .space 24 (6 * 4 bytes)
Y:  .space 24
```

```
Y vector:
91.0
182.0
12.0
20.0
30.0
42.0

Determinant: 0.0
Comparison flag (1 = sum<det): 0

-- program is finished running (0) --
```

**Numpy Output:**

```
Y_expected:  [ 91. 182.  12.  20.  30.  42.]
det_expected:  0.0
flag_expected:  0
```

5. **TEST CASE-5: RANDOM LARGE VALUE OF DETERMINANT**

```
.data
.align 2
N:      .word 6         # We can change this to set n (matrix dimension)

.align 2
# Example 6x6 matrix (can change to any n x n; ensure to keep N consistent)
# Also here Matrix A is stored in row-major format: row0, row1, ..., row(n-1)
A:
    .float 2.0, 1.0, 0.0, 0.0, 0.0, 0.0     # A[0][0..5] (row_1)
    .float 1.0, 203.0, 1.0, 0.0, 0.0, 0.0    # A[1][0..5] (row_2)
    .float 0.0, 1.0, 34243.0, 1.0, 0.0, 0.0    # A[2][0..5] (row_3)
    .float 0.0, 0.0, 1.0, 2.0, 1.0, 0.0     # A[3][0..5] (row_4)
    .float 0.0, 0.0, 0.0, 1.0, 2234.0, 1.0    # A[4][0..5] (row_5)
    .float 0.0, 0.0, 0.0, 0.0, 1.0, 20.0    # A[5][0..5] (row_6)

.align 2
# Example 1 x n row-vector X (change to any values)
X:
    .float 2.0, 3.0, 4.0, 5.0, 6.0, 7.0     # X[0..5]

.align 2
# Allocating space for Result Vector Y (n floats)
# Note to reserve space for n floats; here n=6 so .space 24 (6 * 4 bytes)
Y:  .space 24
```

```
Y vector:
7.0
615.0
136980.0
20.0
13416.0
146.0

Determinant: 1.23895808E12
Comparison flag (1 = sum<det): 1

-- program is finished running (0) --
```

**Numpy Output:**

```
Y_expected:  [7.0000e+00 6.1500e+02 1.3698e+05 2.0000e+01 1.3416e+04 1.4600e+02]
det_expected:  1238958185598.9976
flag_expected:  1
```

## 4.2 Verification Strategy

**Simulator used:** RARS (RV32IMF) v1.6

**Python (Numpy) code used for verification**: Image attached below

```python
import numpy as np

A = np.array([[1,2,3,4,5,6],
              [2,1,2,1,2,1],
              [3,0,1,2,1,0],
              [1,2,3,4,3,2],
              [2,1,0,1,2,1],
              [1,3,2,4,1,5]], dtype=float)

X = np.array([1,2,3,4,5,6], dtype=float)
Y_expected = A @ X
det_expected = np.linalg.det(A)

sum_Y = np.sum(Y_expected)
flag_expected = 1 if sum_Y < det_expected else 0

print("Y_expected: ", Y_expected)
print("det_expected: ", float(det_expected))
print("flag_expected: ", int(flag_expected))
```

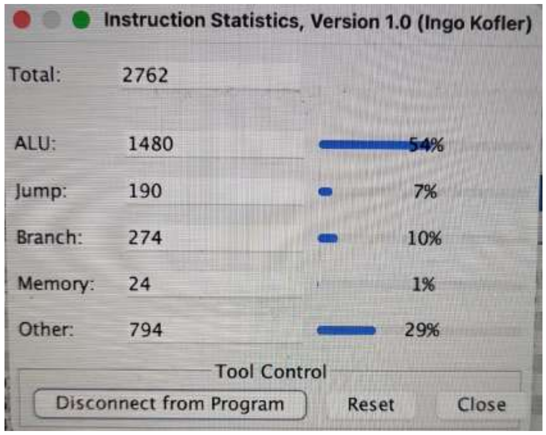
**Output for the sample test case of report:** Image attached below

```
(base) sudhirkumargupta@sudhirs-MacBook-1590
Y_expected:  [91. 30. 19. 57. 24. 64.]
det_expected:  -416.00000000000034
flag_expected:  0
(base) sudhirkumargupta@sudhirs-MacBook-1590
```

## 5. Performance Evaluation and Optimization

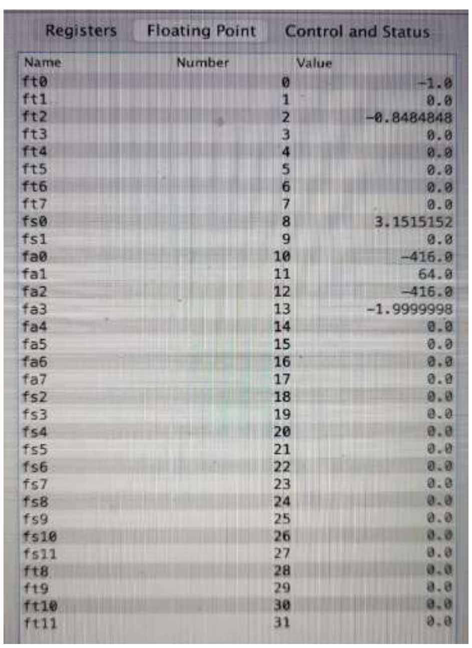
### 5.1 Instruction Count and Register Usage

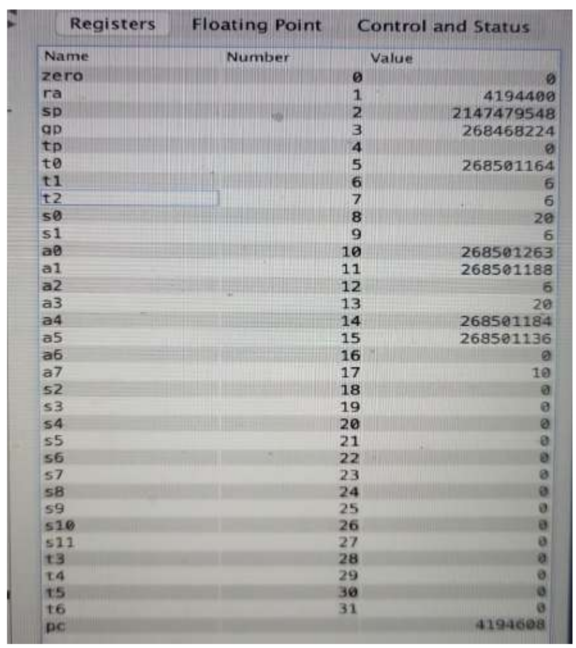
**Instruction Count:**




From here we can observe that the total number of instructions for our given sample code are 2762.

**Register Usage:**




### 5.2 Cycle-Level Analysis

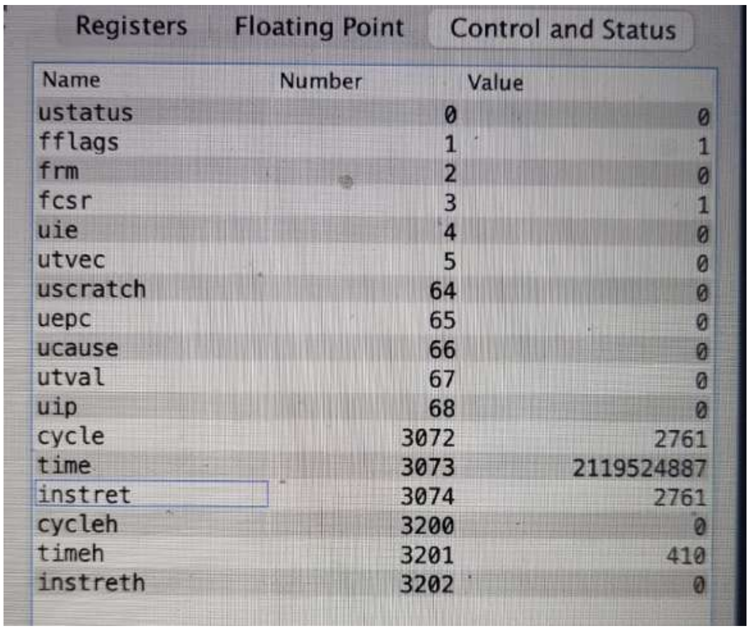
**Assumptions (Ideal 5-Stage Pipeline):**

- Load/Store: 3 cycles (including memory latency)
- Integer ALU: 1 cycle
- Integer multiply: 3 cycles

- Float ALU: 4 cycles
- Float multiply: 5 cycles
- Float divide: 20 cycles
- Branch: 1 cycle (predicted correctly)

**Pipeline Hazards:**

- **Data hazards**: FP operations cause stalls due to multi-cycle latency
- **Structural hazards**: Minimal (single FP unit assumed)
- **Control hazards**: Branch mispredictions in loop conditions (~5% penalty)



Registers | Floating Point | Control and Status

| Name | Number | Value |
|---|---|---|
| ustatus | 0 | 0 |
| fflags | 1 | 1 |
| frm | 2 | 0 |
| fcsr | 3 | 1 |
| uie | 4 | 0 |
| utvec | 5 | 0 |
| uscratch | 64 | 0 |
| uepc | 65 | 0 |
| ucause | 66 | 0 |
| utval | 67 | 0 |
| uip | 68 | 0 |
| cycle | 3072 | 2761 |
| time | 3073 | 2119524887 |
| instret | 3074 | 2761 |
| cycleh | 3200 | 0 |
| timeh | 3201 | 410 |
| instreth | 3202 | 0 |

**Cycles Per Instruction, CPI =** Total cycles/Total instructions

= 2761/2762 (from before)

= 1 (approx.)

## 5.3 Optimization Techniques

**Implemented Optimizations:**

1. **Register Reuse**
   - Temporary registers recycled across loop iterations
   - Minimizes load/store operations
   - Example: x11-x15 reused for address computation
2. **Address Calculation Optimization**
   - Pre-compute base addresses outside loops
   - Use shift-left-logical-immediate (slli) instead of multiply for power-of-2

3. **Loop Invariant Code Motion**
   - Matrix base address loaded once per function
   - N stored in saved register (x8) to avoid repeated loads
4. **Elimination of Redundant Comparisons**
   - Absolute value computed inline without function call overhead
5. **Minimized Stack Operations**
   - Only essential registers saved (ra, s0, s1)
   - Temporary registers not preserved across calls

**Potential Future Optimizations:**

1. **Loop Unrolling**
   - Unroll inner loops by factor of 2 or 4
   - Reduces loop overhead and branch mispredictions
   - Trade-off: Code size increases
2. **Software Pipelining**
   - Overlap memory loads with computation
   - Prefetch next iteration's data while processing current
   - Hides memory latency
3. **Blocking/Tiling (for larger matrices)**
   - Improve cache locality by processing submatrices
   - Reduce cache misses for N > 32
4. **Strength Reduction**
   - Replace expensive operations with cheaper equivalents
   - Example: Use addition instead of multiplication in address calculation

**Optimization Trade-offs:**

1. **Code size vs. Speed**: Unrolling increases size but improves speed
2. **Complexity vs. Maintainability:** Advanced optimizations harder to debug
3. **Register pressure**: Already at maximum; further optimization limited

## 6. References


1. **RISC-V Specifications:**
    - "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2"
    - "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10"
2. **Simulators:**
    - RARS (RISC-V Assembler and Runtime Simulator): https://github.com/TheThirdOne/rars
    - Venus RISC-V Simulator: https://venus.cs61c.org/
3. **Textbooks:**
    - Patterson, D.A., and Hennessy, J.L., "Computer Organization and Design: The Hardware/Software Interface, RISC-V Edition", Morgan Kaufmann, 2017
    - Harris, S.L., and Harris, D.M., "Digital Design and Computer Architecture: RISC-V Edition", Morgan Kaufmann, 2021
4. **Algorithms:**
    - Golub, G.H., and Van Loan, C.F., "Matrix Computations", 4th Edition, Johns Hopkins University Press, 2013
    - Press, W.H., et al., "Numerical Recipes: The Art of Scientific Computing", 3rd Edition, Cambridge University Press, 2007
5. **IEEE Standards:**
    - IEEE Standard 754-2008, "IEEE Standard for Floating-Point Arithmetic"
6. **Online Resources:**
    - RISC-V International: https://riscv.org/
    - RISC-V Assembly Programming Guide: https://github.com/riscv-non-isa/riscv-asm-manual
    - NumPy Documentation (for verification): https://numpy.org/doc/
    - ELL305 (Computer Architecture) IIT Delhi: Lecture notes

## 7. Appendix — Source Code Listings

```
# File Name: matvec_det_rv32imf.s
# Objectives: a) General n x n matrix * vector, b) determinant (partial pivoting), c)
comparision between determinant and sum(Y)
# Integer registers used: x5 to x15 (plus ra/sp)
# Floating registers: fa0 to fa7, ft0 to ft3, and fs0

# I have added multiple comments which explain the purpose of various parts of code,
# register usage, and the purpose of each code block so that the algorithm can be
# understood while reading through the code
#################################################################
###########################

.data
.align 2
N:     .word 6       # We can change this to set n (matrix dimension)

.align 2
# Example 6x6 matrix (can change to any n x n; ensure to keep N consistent)
# Also here Matrix A is stored in row-major format: row0, row1, ..., row(n-1)
A:
    .float 1.0, 2.0, 3.0, 4.0, 5.0, 6.0    # A[0][0..5] (row_1)
    .float 2.0, 1.0, 2.0, 1.0, 2.0, 1.0    # A[1][0..5] (row_2)
    .float 3.0, 0.0, 1.0, 2.0, 1.0, 0.0    # A[2][0..5] (row_3)
    .float 1.0, 2.0, 3.0, 4.0, 3.0, 2.0    # A[3][0..5] (row_4)
    .float 2.0, 1.0, 0.0, 1.0, 2.0, 1.0    # A[4][0..5] (row_5)
    .float 1.0, 3.0, 2.0, 4.0, 1.0, 5.0    # A[5][0..5] (row_6)

.align 2
# Example 1 x n row-vector X (change to any values)
X:
    .float 1.0, 2.0, 3.0, 4.0, 5.0, 6.0    # X[0..5]

.align 2
# Allocating space for Result Vector Y (n floats)
# Note to reserve space for n floats; here n=6 so .space 24 (6 * 4 bytes)
Y:  .space 24

.align 2
# Determinant result (float) and comparison flag (word) storage
det_result:     .float 0.0
.align 2
comparison_flag: .word  0
```

```
.align 2
# Useful constant floats
ONE_F:   .float 1.0
NEGONE_F:.float -1.0

# Output format used for printing outputs (i.e. RARS ecall strings)
msgY:    .asciz "Y vector:\n"
msgDet:  .asciz "\nDeterminant:\n"
msgFlag: .asciz "\nComparison flag (1 = sum<det):\n"
newline: .asciz "\n"

.text
.globl main

# Register mapping/comments (kept consistent with the given register constraints):
# x5  = t0    (used as general temp / base pointer)
# x6  = t1
# x7  = t2
# x8  = s0    (callee-saved; used for N or preserved values)
# x9  = s1    (callee-saved; used for sign or saved values)
# x10 = a0    (argument or return register)
# x11 = a1
# x12 = a2
# x13 = a3
# x14 = a4
# x15 = a5
# ra (x1) and sp (x2) used per ABI for return and stack pointer.


# main
# Sets up arguments and calls matvec_mul, det_comp, compare_sum_det, and then prints on console
main:
   # Load addresses and n into integer registers (Here we use x10 to x13 as a0 to a3)
   la   x10, A        # a0 = &A (base address of matrix)
   la   x11, X        # a1 = &X (base address of input vector)
   la   x12, Y        # a2 = &Y (base address for output vector)
   lw   x13, N        # a3 = N (matrix/vector dimension)
   # Call matvec_mul(a0=&A, a1=&X, a2=&Y, a3=N)
   jal  ra, matvec_mul # jump-and-link to matrix-vector multiply

   # (Following 5 lines are for Determinant computation)
```

```
    # Prepare args for det_comp(a0=&A, a1=N)
    la   x10, A        # a0 = &A (we operate in-place on A for elimination)
    lw   x11, N        # a1 = N
    # Call det_comp; result returned in floating register fa0
    jal  ra, det_comp

    # Store determinant float result into det_result label
    la   x5, det_result # x5 = address of det_result
    fsw  fa0, 0(x5)     # write fa0 -> memory at det_result

    # (Following lines are for sum(Y) < det_result computation into comparision_flag)
    # Prepare args for compare_sum_det(a0=&Y, a1=&det_result, a2=N)
    la   x10, Y
    la   x11, det_result
    lw   x12, N
    # Call compare_sum_det; integer return in a0 (x10)
    jal  ra, compare_sum_det

    # Store comparison (a0) into comparison_flag (memory)
    la   x6, comparison_flag
    sw   x10, 0(x6)     # store a0 (x10) -> comparison_flag

    # (These lines to print outputs (Y vector, determinant, flag))
    # Print header for Y
    la   x10, msgY
    li   a7, 4          # ecall 4 = print_string (in RARS)
    ecall

    # Loop to print each element of Y
    la   x5, Y          # x5 = base of Y
    lw   x6, N          # x6 = n
    li   x7, 0          # x7 = loop index i
print_y:
    beq  x7, x6, print_y_done  # if i == n, we are done
    slli x8, x7, 2           # offset = i * 4
    add  x11, x5, x8         # x11 = &Y[i]
    flw  fa0, 0(x11)         # load Y[i] into fa0
    li   a7, 2               # ecall 2 = print_float
    ecall
    la   x10, newline
    li   a7, 4
    ecall
    addi x7, x7, 1           # i++ ; increment loop index and loop again
```

```
    j    print_y
print_y_done:

    # Print determinant header and value
    la   x10, msgDet                    # load the address of the message string msgDet
    li   a7, 4
    ecall                    # print "Determinant =" on screen
    la   x11, det_result     # load the determinant value
    flw  fa0, 0(x11)
    li   a7, 2                          # print the float value of determinant on screen
    ecall
    la   x10, newline                   # move to new line
    li   a7, 4
    ecall

    # Print comparison flag header and value
    la   x10, msgFlag                   # load address of the msgFlag label
    li   a7, 4
    ecall                    # print "Comparison flag (1 = sum<det):"
    la   x10, comparison_flag
    lw   a0, 0(x10)     # load integer flag into a0 for print_int
    li   a7, 1          # ecall 1 = print_int; prints comparision flag value on screen
    ecall
    la   x10, newline
    li   a7, 4
    ecall

    # Exit program
    li   a7, 10        # ecall 10 = exit
    ecall

#################################################################
#########
# Here I implement matvec_mul(a0=&A, a1=&X, a2=&Y, a3=N) subroutine
# This performs Y = A * X, and uses only x5 to x15 for integer work and fa* for FP.
# Also saves ra and s0 (x8) on stack per calling convention.

matvec_mul:
    addi sp, sp, -8     # make room on stack (2 words)
    sw   ra, 4(sp)      # save return address
    sw   x8, 0(sp)      # save s0 because we will use it

    # Map incoming argument registers to our chosen regs
```

```
    mv   x5, x10       # x5 = &A
    mv   x6, x11       # x6 = &X
    mv   x7, x12       # x7 = &Y
    mv   x8, x13       # x8 = N
    li   x9, 0         # x9 = i = 0 (outer loop index)

mv_outer:
    beq  x9, x8, mv_done    # if i == N, done
    fmv.s.x fa0, x0         # fa0 = 0.0 ; accumulator for row sum
    li   x10, 0             # x10 = j = 0 (inner loop index)

mv_inner:
    beq  x10, x8, mv_store  # if j == N, store Y[i]
    # Computing the linear index = i*N + j
    mul  x11, x9, x8        # x11 = i*N
    add  x11, x11, x10      # x11 = i*N + j
    slli x11, x11, 2        # byte offset = (i*N + j)*4
    add  x12, x5, x11       # x12 = &A[i][j]
    flw  fa1, 0(x12)        # fa1 = A[i][j]

    # Load X[j]
    slli x11, x10, 2        # offset = j*4
    add  x12, x6, x11       # x12 = &X[j]
    flw  fa2, 0(x12)        # fa2 = X[j]

    # sum += A[i][j] * X[j]
    fmul.s fa3, fa1, fa2
    fadd.s fa0, fa0, fa3

    addi x10, x10, 1        # j++; increment loop index and repeat
    j    mv_inner

mv_store:
    # Store accumulated sum into Y[i]
    slli x11, x9, 2     # x9 << 2 ; multiply by 4
    add  x12, x7, x11          # add offset to the base address of Y
    fsw  fa0, 0(x12)        # Y[i] = fa0

    addi x9, x9, 1          # i++
    j    mv_outer

mv_done:
    # Restore saved registers and return
```

```
   lw   ra, 4(sp)
   lw   x8, 0(sp)
   addi sp, sp, 8              # restores stack pointer
   ret

##################################################################
#########
# Here I calculate the determinant in the det_comp(a0=&A, a1=N) subroutine
# To find determinant I use Gaussian elimination with partial pivoting algorithm:
# 1) performs in-place row operations on matrix A
# 2) returns determinant in fa0
# 3) uses x5 to x15 and floating regs (fs0 for holding abs value)
# 4) saves ra and a few callee-saved regs on stack

det_comp:
   addi sp, sp, -20       # allocate 20 bytes to save registers
   sw   ra, 16(sp)        # save return address
   sw   x8, 12(sp)        # save s0
   sw   x9, 8(sp)         # save s1
   sw   x10, 4(sp)        # save a0 (temp)
   sw   x11, 0(sp)        # save a1 (temp)

   mv   x5, x10           # x5 = &A (argument a0)
   mv   x8, x11           # x8 = N  (argument a1)
   li   x6, 0             # x6 = i (outer loop pivot index)
   li   x9, 1             # x9 = sign multiplier (1 or -1)

outer_pivot:
   beq  x6, x8, compute_product  # when i == N, elimination finished

   mv   x7, x6            # x7 = max_row (initially i)

   # Load the pivot A[i][i] into fa0
   mul  x11, x6, x8
   add  x11, x11, x6      # x11 = i*N + i (linear index)
   slli x11, x11, 2       # byte offset = (i*N + i) * 4
   add  x12, x5, x11      # x12 = &A[i][i]
   flw  fa0, 0(x12)       # fa0 = A[i][i]

   # Compute absolute value of pivot into fs0:
   # ft0 = fa0 ; ft1 = 0 ; if fa0 < 0 then ft0 = -fa0 ; fs0 = ft0
   fmv.s  ft0, fa0            # float ft0 = fa0;
   fmv.s.x ft1, x0            # float ft1 = 0.0;
```

```
   flt.s  x13, fa0, ft1     # x13 = 1 if fa0 < 0.0
   beqz  x13, no_neg_p0     # if the number is already ≥ 0, we don't want to change it.
   fsub.s ft0, ft1, fa0     # ft0 = 0.0 - fa0 = -fa0
no_neg_p0:
   fmv.s fs0, ft0           # fs0 = |pivot|

   addi x11, x6, 1          # start k = i+1 to search for max pivot

find_max_loop:
   beq  x11, x8, pivot_found # if k == N end search

   # Load candidate A[k][i]
   mul  x12, x11, x8
   add  x12, x12, x6
   slli x12, x12, 2
   add  x13, x5, x12
   flw  fa1, 0(x13)         # fa1 = A[k][i]

   # Compute abs(fa1) and move to ft2
   fmv.s ft2, fa1
   fmv.s.x ft3, x0
   flt.s x14, fa1, ft3
   beqz x14, no_neg_cand
   fsub.s ft2, ft3, fa1     # ft2 = -fa1 if negative
no_neg_cand:
   # Compare current max abs (fs0) with candidate abs (ft2)
   flt.s x14, fs0, ft2
   beqz x14, skip_update_max
   fmv.s fs0, ft2           # update fs0 = candidate abs
   mv   x7, x11             # update max_row = k
skip_update_max:
   addi x11, x11, 1
   j    find_max_loop

pivot_found:
   # If max_row != i then swap rows i and max_row (full row swap)
   beq  x7, x6, no_row_swap
   li   x11, 0              # column index j = 0

swap_row_loop:
   beq  x11, x8, swap_done  # finished swapping all columns
   # Load A[i][j] into fa2
   mul  x12, x6, x8
```

```
    add  x12, x12, x11
    slli x12, x12, 2
    add  x13, x5, x12
    flw  fa2, 0(x13)
    # Load A[max_row][j] into fa3
    mul  x14, x7, x8
    add  x14, x14, x11
    slli x14, x14, 2
    add  x15, x5, x14
    flw  fa3, 0(x15)
    # Swap memory words
    fsw  fa3, 0(x13)
    fsw  fa2, 0(x15)
    addi x11, x11, 1
    j    swap_row_loop
swap_done:
    li   x11, -1
    mul  x9, x9, x11        # flip sign multiplier s1 *= -1

no_row_swap:
    # Reload pivot (A[i][i]) because swap may have changed it
    mul  x11, x6, x8            # compute row offset: x11 = x6*x8 = i*N
    add  x11, x11, x6           # x11 = i*N + i
    slli x11, x11, 2            # multiplying by 4 to compute byte offset
    add  x12, x5, x11           # x12 = (base address of A) + (byte offset)
    flw  fa0, 0(x12)

    # If pivot is zero => singular matrix => determinant is zero
    fmv.x.w x11, fa0
    beqz x11, det_zero

    # Perform elimination for rows k = i+1 .. N-1
    addi x11, x6, 1

elim_k:
    beq  x11, x8, next_pivot  # if k == N, go to next pivot
    # Load A[k][i] into fa1
    mul  x12, x11, x8            # (similar process of loading as before)
    add  x12, x12, x6
    slli x12, x12, 2
    add  x13, x5, x12
    flw  fa1, 0(x13)
    # factor = A[k][i] / pivot
```

```
    fdiv.s fa2, fa1, fa0

    # For each column j = i .. N-1 update A[k][j] -= factor * A[i][j]
    mv   x12, x6           # j = i
elim_j:
    beq  x12, x8, done_row_elim
    # Load A[i][j] into ft0
    mul  x13, x6, x8
    add  x13, x13, x12
    slli x13, x13, 2
    add  x14, x5, x13
    flw  ft0, 0(x14)
    # Load A[k][j] into ft1
    mul  x13, x11, x8
    add  x13, x13, x12
    slli x13, x13, 2
    add  x14, x5, x13
    flw  ft1, 0(x14)
    # Compute ft1 = ft1 - factor * ft0
    fmul.s ft2, fa2, ft0   # fa2 = factor = A[k][i]/A[i][i]
    fsub.s ft1, ft1, ft2
    fsw  ft1, 0(x14)       # store updated A[k][j]
    addi x12, x12, 1            # j++ ; Move to next col of same row k, repeat loop
    j    elim_j

done_row_elim:
    addi x11, x11, 1       # k++ ; Move to the next row after finished with the kth row
    j    elim_k

next_pivot:
    addi x6, x6, 1       # i++ (next pivot row)
    j    outer_pivot

det_zero:
    fmv.s.x fa0, x0      # set determinant fa0 = 0.0
    j    det_cleanup

compute_product:
    # Multiply diagonal entries to compute determinant
    la   x11, ONE_F
    flw  fa0, 0(x11)     # fa0 = 1.0 (accumulator)
    li   x6, 0          # index i = 0
```

```
diag_loop:
   beq  x6, x8, apply_sign  # when i == N end product loop
   mul  x12, x6, x8
   add  x12, x12, x6
   slli x12, x12, 2
   add  x13, x5, x12
   flw  ft0, 0(x13)     # ft0 = A[i][i]
   fmul.s fa0, fa0, ft0 # fa0 *= A[i][i]
   addi x6, x6, 1
   j    diag_loop

apply_sign:
   # Apply sign correction if odd number of row swaps occurred
   li   x11, 1
   beq  x9, x11, det_cleanup # if sign == 1 skip applying -1
   la   x11, NEGONE_F
   flw  ft0, 0(x11)
   fmul.s fa0, fa0, ft0   # fa0 *= -1.0

det_cleanup:
   # Restore saved registers and return (fa0 holds determinant)
   lw   ra, 16(sp)
   lw   x8, 12(sp)
   lw   x9, 8(sp)
   lw   x10, 4(sp)
   lw   x11, 0(sp)
   addi sp, sp, 20
   ret

################################################################
#########
# Here I calculate for comparison flag with compare_sum_det(a0=&Y, a1=&det_result,
a2=N) subroutine
# Algorithm: Sum Y and compare with det_result, return a0 = 1 if sum < det else 0
# This also uses only x5 to x15 (and ra/sp) and FP accumulators

compare_sum_det:
   addi sp, sp, -8
   sw   ra, 4(sp)
   sw   x8, 0(sp)

   mv   x5, x10   # x5 = base Y (argument a0)
   mv   x6, x11   # x6 = &det_result (argument a1)
```

```
    mv   x8, x12   # x8 = N (argument a2)
    fmv.s.x fa0, x0 # fa0 = 0.0 accumulator for sum
    li   x7, 0      # index i = 0

sum_loop:
    beq  x7, x8, sum_done
    slli x13, x7, 2
    add  x14, x5, x13
    flw  fa1, 0(x14)     # fa1 = Y[i]
    fadd.s fa0, fa0, fa1 # sum += Y[i]
    addi x7, x7, 1
    j    sum_loop

sum_done:
    flw  fa2, 0(x6)      # load det_result into fa2
    flt.s x10, fa0, fa2  # x10 = 1 if sum < det else 0
    # Return: a0 must contain the integer return value; x10 is a0 so good.
    lw   ra, 4(sp)
    lw   x8, 0(sp)
    addi sp, sp, 8
    ret


# This marks the end of our code
# We have successfully implemented multiplication of matrix with row vector, determinant
computation using Gaussian Elimination,
# and computation of comparison flag

################################################################
#########
```