ELL 782: Computer Architecture
Assignment-1

Name-Suraj Natekar
Entry no.-2024EET2355

- **Introduction**

  In this assignment, you need to write the RISC-V assembly code to find the inverse of a matrix. This assignment needs to be done individually. Your implementation should consist of two modules: a compute module and a verification module. The compute module will calculate the inverse of the matrix, and the verification module will multiply the input matrix by its inverse (the output of the compute module) to verify the result. The result of this multiplication should be an identity matrix if the inverse is correct.

- **Implementation Methodology:**
  1. Data section
  2. Creating identity Matrix
  3. Creating Matrix copy
  4. Gaussian elimination
  5. Matrix verification
  6. Printing inverse Matrix
  7. Printing verification Matrix
  8. Test Cases

1) Data section

a) Giving input Matrix (order N) and also size of Matrix i.e. order of Matrix value N

```
N: .word 2                  #This will be our input of order of Matrix
```

The value of N will also be helpful in maintaining loop counter

```
matrix:
   .float 2, 1    # First row
   .float 1, 0,    # Second row
```

This way I have to hardcode Input Matrix of any order.

b) Some space allocation is done to store computations performed on matrix

i) Identity: .space 100

Allocates 100 bytes of memory (enough to store 25 floating-point numbers) for storing the result of matrix operations or computations.

ii) matrix_cpy: .space 100
Allocates 100 bytes of memory, used to store a copy of the original matrix, for operations of matrix multiplication with inverse of matrix to verify the result of inversion.

iii) result_id: .space 100
Allocates 100 bytes of memory for storing verified results, result_id matrix will contain an identity matrix on successful inversion of matrix.

iv) address: .float 0.0
Reserves space for a single floating-point value initialized to 0.0 used on program to compare floating point value to zero.

v) String in program used to print results in terminal

```
newline_string: .asciiz "\n"
space_string: .asciiz "   "
decimal_string: .asciiz "."
existance: .asciiz "inverse of matrix does not exist"
negate: .asciiz "-"
print_zero: .asciiz "0"
print_inverse: .asciiz "inverse matrix A"
print_verify: .asciiz "verification matrix B"
```

2) Creating identity Matrix:

i) Pseudo code for Matrix initialization:

```
Initialize i (t1) to 0
Initialize j (t2) to 0
    Calculate the effective address of the current matrix element using i and j
    If i == j (diagonal element):
        Make value in register 1.0 and store back to memory (identity matrix diagonal)
    Else:
        Make value in the register 0.0 and store back to memory(non-diagonal elements)
Increment j (move to the next column)
Loop (j < N):
Increment i (move to the next row)
```

```
Loop (i < N):
```

ii) The effective address for each matrix element is calculated using the row and column indices:

a) For outer loop i(t0=0) and for inner loop j(t1=0)
b) Register s9 is stored with value (N*4) to skip the whole row of addresses as one memory location is 4 bytes.
c) Register s10 is stored with value constant(4) to jump over one one memory location.
d) Register a7 is stored with base address of identity Matrix

```
mul t4, s9, t1        computes the offset for the row.
mul t5, s10, t2       computes the offset for the column.
add t4, t4, t5        sums these offsets to get the total offset.
add t5, t4, a7        adds the base address of the matrix to the offset,
resulting                                                in the effective
address.
```

3) Creating Matrix copy:

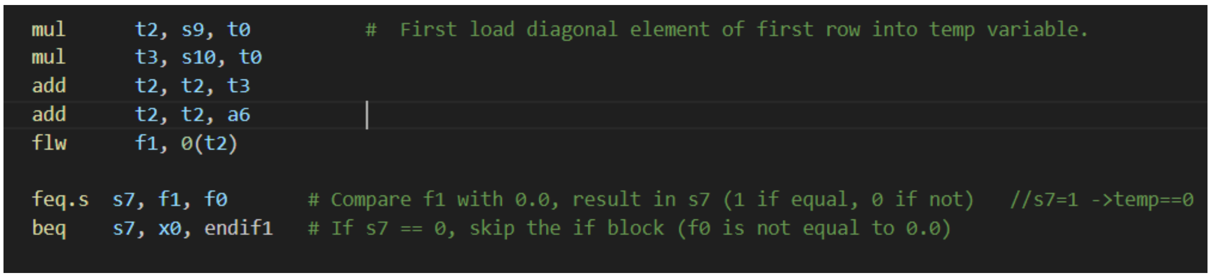
i) Matrix copy is required to multiply with inverse matrix to verify obtained result.
ii) Register t0 is stored with base address of original matrix and register t1 is stored with base address of matrix_copy
iii) Initialize loop counter t2=0 and store count of total number of elements in t3(t3=N*N)
iv) Pseudo code

```
.t0 < --  0(original Matrix)
.t1 < -- 0(matrix_copy)
Initialize the counter (t2==0)
Calculate the total number of elements in the matrix (t3==N * N)
    Loop (t2< t3):
        Load the floating-point element from the source matrix          (f1 <-- 0(t0))
        Store the floating-point element into the destination matrix    (0(t1) <-- f1)
        Move to the next element in the source matrix                   (t0=t0+4)
        Move to the next element in the destination matrix              (t1=t1+4)
        Increment the counter                                           (t2=t2+1)
    End loop
```

4) Gaussian Elimination:

Steps to be followed for both original matrix and identity matrix

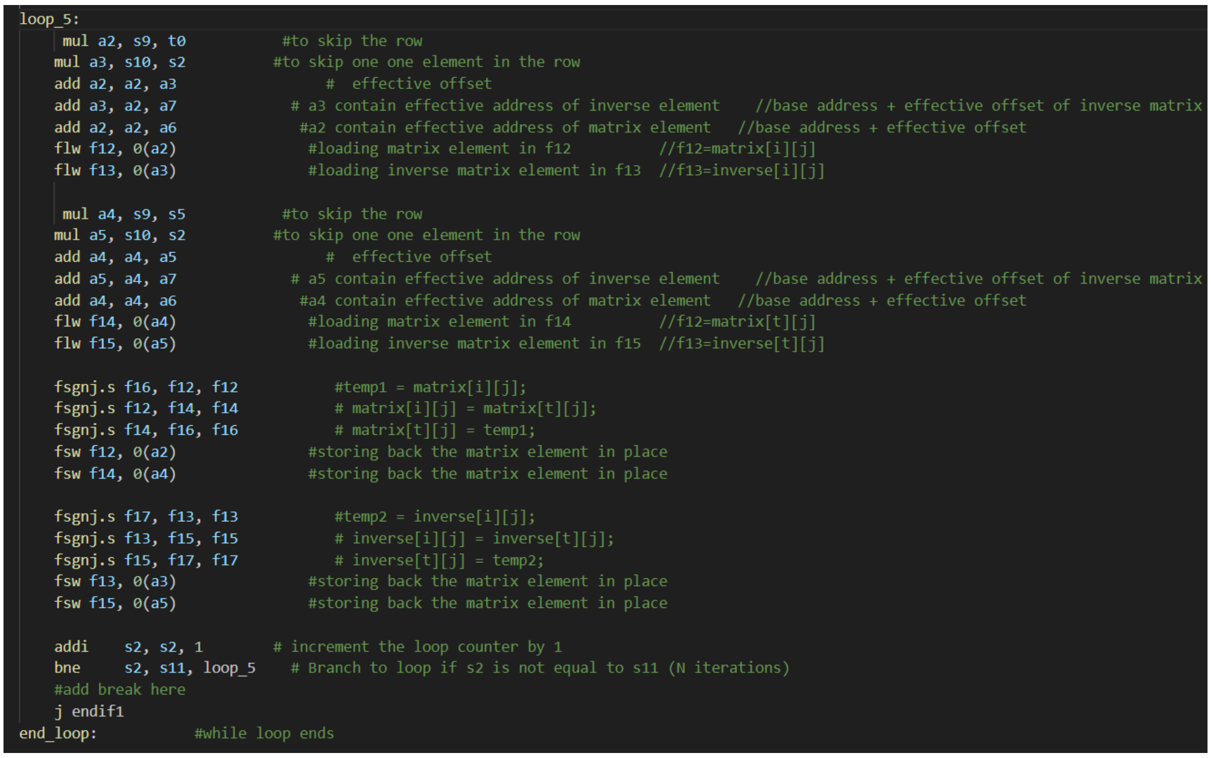
i) First load diagonal element of first row into temp variable.



```
mul      t2, s9, t0          #  First load diagonal element of first row into temp variable.
mul      t3, s10, t0
add      t2, t2, t3
add      t2, t2, a6
flw      f1, 0(t2)

feq.s  s7, f1, f0       # Compare f1 with 0.0, result in s7 (1 if equal, 0 if not)   //s7=1 ->temp==0
beq    s7, x0, endif1   # If s7 == 0, skip the if block (f0 is not equal to 0.0)
```

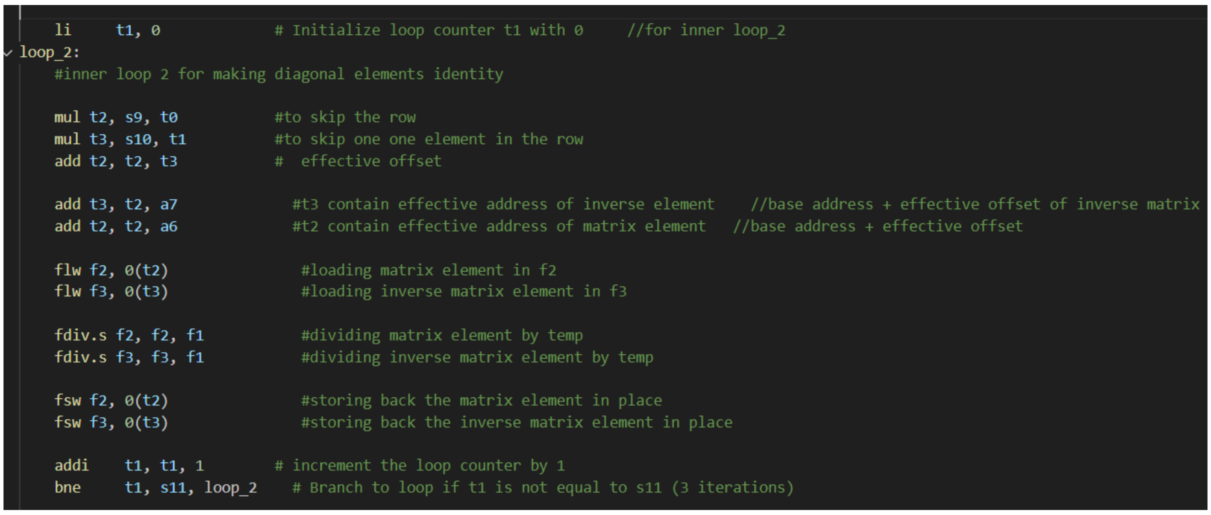
Loading diagonal element into temp variable and comparing it with zero.

ii) If diagonal element is zero then we need to swap row with bottom row not containing zero element below the diagonal element.
If non zero element is not found in a row below diagonal element then inverse of matrix does not exist.
Swap function:
Loop for swapping elements of matrices



```
loop_5:
     mul a2, s9, t0           #to skip the row
    mul a3, s10, s2          #to skip one one element in the row
    add a2, a2, a3                #  effective offset
    add a3, a2, a7             # a3 contain effective address of inverse element    //base address + effective offset of inverse matrix
    add a2, a2, a6              #a2 contain effective address of matrix element   //base address + effective offset
    flw f12, 0(a2)               #loading matrix element in f12          //f12=matrix[i][j]
    flw f13, 0(a3)               #loading inverse matrix element in f13  //f13=inverse[i][j]

     mul a4, s9, s5           #to skip the row
    mul a5, s10, s2          #to skip one one element in the row
    add a4, a4, a5                #  effective offset
    add a5, a4, a7             # a5 contain effective address of inverse element    //base address + effective offset of inverse matrix
    add a4, a4, a6              #a4 contain effective address of matrix element   //base address + effective offset
    flw f14, 0(a4)               #loading matrix element in f14          //f12=matrix[t][j]
    flw f15, 0(a5)               #loading inverse matrix element in f15  //f13=inverse[t][j]

    fsgnj.s f16, f12, f12           #temp1 = matrix[i][j];
    fsgnj.s f12, f14, f14           # matrix[i][j] = matrix[t][j];
    fsgnj.s f14, f16, f16           # matrix[t][j] = temp1;
    fsw f12, 0(a2)               #storing back the matrix element in place
    fsw f14, 0(a4)               #storing back the matrix element in place

    fsgnj.s f17, f13, f13           #temp2 = inverse[i][j];
    fsgnj.s f13, f15, f15           # inverse[i][j] = inverse[t][j];
    fsgnj.s f15, f17, f17           # inverse[t][j] = temp2;
    fsw f13, 0(a3)               #storing back the matrix element in place
    fsw f15, 0(a5)               #storing back the matrix element in place

    addi    s2, s2, 1        # increment the loop counter by 1
    bne     s2, s11, loop_5    # Branch to loop if s2 is not equal to s11 (N iterations)
    #add break here
    j endif1
end_loop:           #while loop ends
```

iii) If our diagonal element is not zero or we found element which is not zero below the diagonal element in a row then we divide the row with diagonal element.
Such that every diagonal element becomes identity element.



```
    li     t1, 0            # Initialize loop counter t1 with 0     //for inner loop_2
loop_2:
    #inner loop 2 for making diagonal elements identity

    mul t2, s9, t0          #to skip the row
    mul t3, s10, t1         #to skip one one element in the row
    add t2, t2, t3          #  effective offset

    add t3, t2, a7            #t3 contain effective address of inverse element    //base address + effective offset of inverse matrix
    add t2, t2, a6            #t2 contain effective address of matrix element   //base address + effective offset

    flw f2, 0(t2)              #loading matrix element in f2
    flw f3, 0(t3)              #loading inverse matrix element in f3

    fdiv.s f2, f2, f1          #dividing matrix element by temp
    fdiv.s f3, f3, f1          #dividing inverse matrix element by temp

    fsw f2, 0(t2)              #storing back the matrix element in place
    fsw f3, 0(t3)              #storing back the inverse matrix element in place

    addi    t1, t1, 1       # increment the loop counter by 1
    bne     t1, s11, loop_2   # Branch to loop if t1 is not equal to s11 (3 iterations)
```

iv) Make the remaining elements zero.

a. Initialize Outer Loop (k):
   Set k (loop counter t1) to 0.

b. Outer Loop (k loop):
   Check if k == i (Diagonal Element):
   If k equals i (t0), skip the row and go to the end of the outer loop.

c. Calculate Effective Address for matrix[k][i]:
   - Compute row offset for k using s9 * k.
   - Compute column offset for i using s10 * i.
   - Add the base address (a6) to the sum of these offsets to get the effective address.
   - Load matrix[k][i] into a temporary floating-point register f1.

d. Initialize Inner Loop (j):
   Set j (loop counter t2) to 0.

e. Inner Loop (j loop):
   - Calculate Effective Address for matrix[k][j]:
     Compute row offset for k using s9 * k.
     Compute column offset for j using s10 * j.
     Add the base address (a6) to the sum of these offsets to get the effective address.
     Load matrix[k][j] into floating-point register f2.

   - Calculate Effective Address for inverse[k][j]:
     Compute similarly as above but using the base address for the inverse matrix (a7).
     Load inverse[k][j] into floating-point register f3.

- Calculate Effective Address for matrix[i][j]:
  Compute row offset for i using s9 * i.
  Compute column offset for j using s10 * j.
  Add the base address (a6) to the sum of these offsets to get the effective address.
  Load matrix[i][j] into floating-point register f4.

- Calculate Effective Address for inverse[i][j]:
  Compute similarly as above but using the base address for the inverse matrix (a7).
  Load inverse[i][j] into floating-point register f5.

- Perform Multiplication:
  Multiply matrix[i][j] (f4) with temp (f1) and store the result in f4.
  Multiply inverse[i][j] (f5) with temp (f1) and store the result in f5.

- Perform Subtraction:
  Subtract f4 from matrix[k][j] (f2) and store the result in f2.
  Subtract f5 from inverse[k][j] (f3) and store the result in f3.
- Store Updated Values:
  Store the updated value in matrix[k][j] from f2.
  Store the updated value in inverse[k][j] from f3.

- Increment Inner Loop Counter (j):
  Increment j by 1 (t2).
  Repeat the inner loop until j equals the matrix size (s11).

f. End Inner Loop:
   Complete the inner loop for all columns j.

g. Increment Outer Loop Counter (k):
   Increment k by 1 (t1).
   Repeat the outer loop until k equals the matrix size (s11).
h. End Outer Loop:
   Complete the outer loop for all rows k.

5) Matrix Verification:

```
• la s2, result_id
• la s3, matrix_cpy
•
• li     t1, 0              # Initialize loop counter t1 with
  0     //for inner loop_3    i=t1
```

```
loop_10:

li      t2, 0              # Initialize loop counter t2 with
0     //for inner loop_4    j=t2
loop_11:

li      t3, 0
loop_12:

     mul t4, s9, t1
    mul t5, s10, t3
    add t4, t4, t5
    add t4, t4, a7
    flw f1, 0(t4)           #loading element from identity

      mul t5, s9, t3
    mul t6, s10, t2
    add t5, t5, t6
    add t5, t5, s3
    flw f2, 0(t5)           #loading element from original
matrix

     mul t6, s9, t1
    mul s1, s10, t2
    add t6, t6, s1
    add t6, t6, s2
    flw f3, 0(t6)

    fmul.s f1, f1, f2
    fadd.s f3, f3, f1
    fsw f3, 0(t6)           #storing element in result matrix

    addi    t3, t3, 1         # increment the loop counter by 1
    bne     t3, s11, loop_12    # Branch to loop if t3 is not
equal to s11 (N iterations)

    addi    t2, t2, 1         # increment the loop counter by 1
    bne     t2, s11, loop_11    # Branch to loop if t2 is not
equal to s11 (N iterations)

    addi    t1, t1, 1         # increment the loop counter by 1
    bne     t1, s11, loop_10    # Branch to loop if t1 is not
equal to s11 (N iterations)
```

6) Printing inverse Matrix:

a) Outer and Inner Loops:

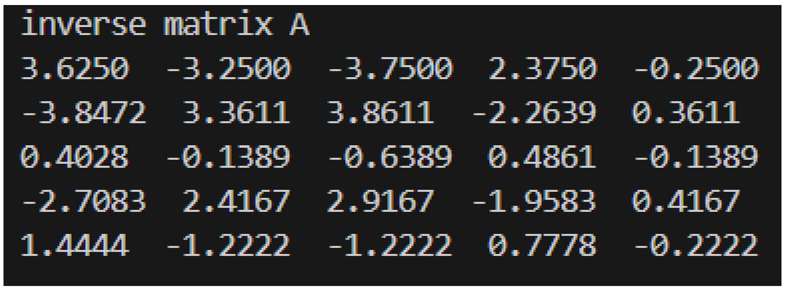
Outer Loop (loop_6): Iterates over each row of the matrix.

Inner Loop (loop_7): Iterates over each column in the current row.

b) Element Access and Check:
   Calculate the effective address of the current matrix element.
   Load the matrix element into a floating-point register f3.
   Check if the number is negative or positive.
c) If the number is positive
   Convert the floating-point number to an integer (fcvt.w.s).
   Print the integer number, then print decimal point using string and then
   Check if the number has a fractional part (not an exact integer).
   If the number has fractional part then multiply it with 10000 to get precision upto 4 decimal places.
   Load fractional part into integer register and print after the decimal point.
d) If the number is negative then there is just one additional step
   Print '-' sign to start of the number and multiply it by '-1' to print after '-' string and all process is same as just we did for positive number.
   For printing fractional number if number encounters to be negative then multiply it with '-1'.
e) All special cases such as
   i.e. if fractional part contain 1.0034 then it was printing 1.34 extra zeros are padded by appropriate if conditions.
f) All other requirements for printing all handled by printing strings

   i.e. printing newline , printing space etc.

   example of printing inverse Matrix:

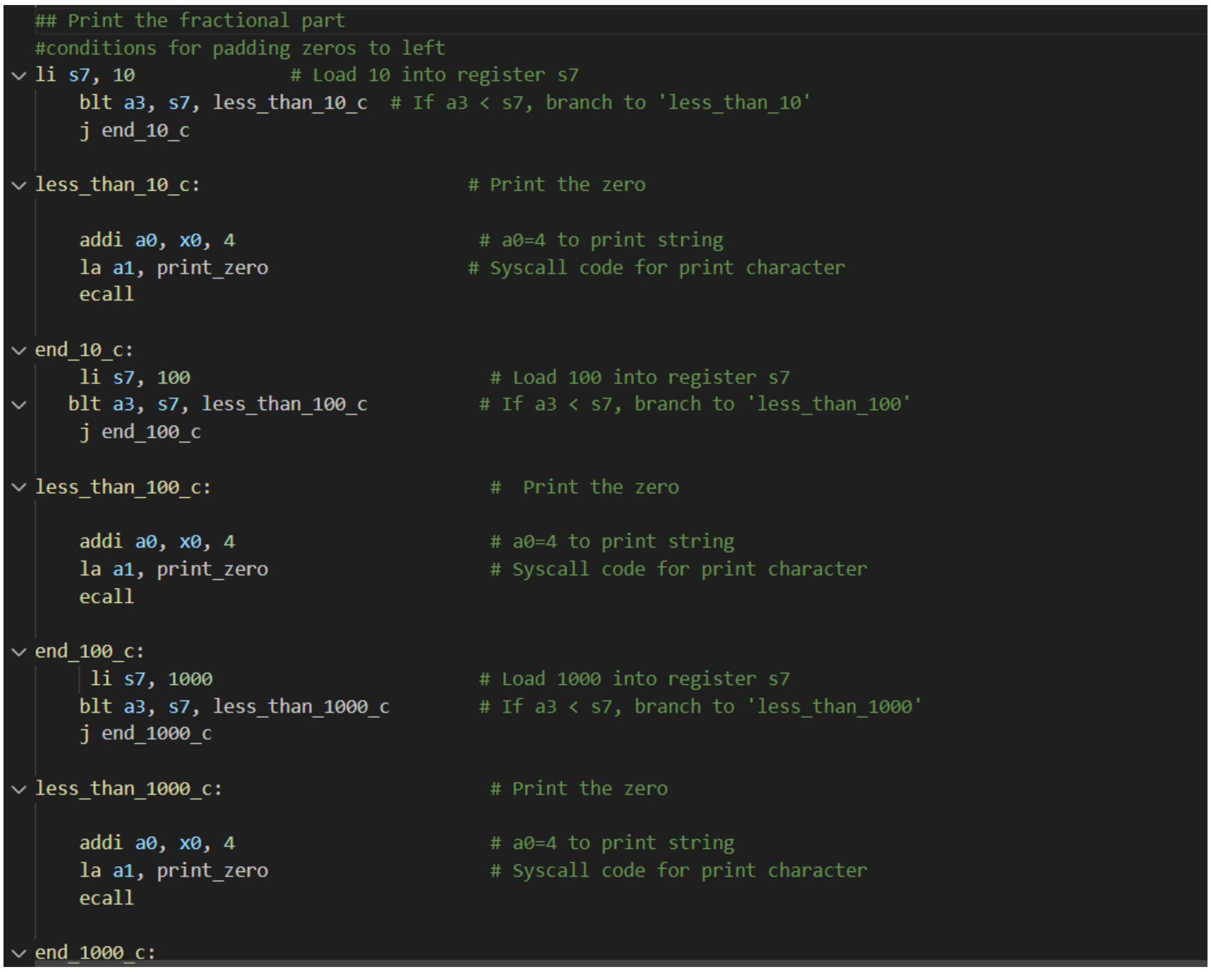


```
inverse matrix A
3.6250  -3.2500  -3.7500  2.3750  -0.2500
-3.8472  3.3611  3.8611  -2.2639  0.3611
0.4028  -0.1389  -0.6389  0.4861  -0.1389
-2.7083  2.4167  2.9167  -1.9583  0.4167
1.4444  -1.2222  -1.2222  0.7778  -0.2222
```

7) Printing verification Matrix:

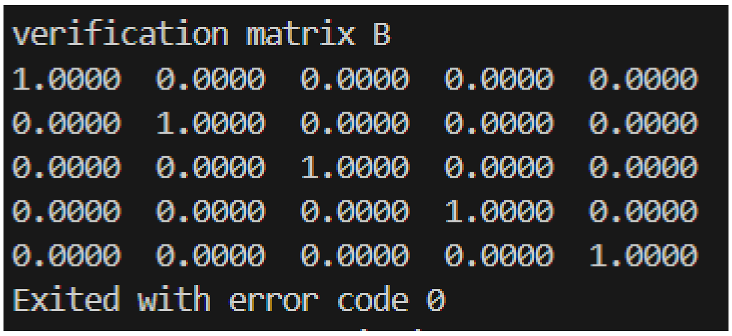
Printing verification matrix is same as printing inverse matrix.

Code snippet for padding zeros after decimal point for printing:



```
## Print the fractional part
#conditions for padding zeros to left
li s7, 10              # Load 10 into register s7
    blt a3, s7, less_than_10_c  # If a3 < s7, branch to 'less_than_10'
    j end_10_c

less_than_10_c:                       # Print the zero

    addi a0, x0, 4                     # a0=4 to print string
    la a1, print_zero                 # Syscall code for print character
    ecall

end_10_c:
    li s7, 100                          # Load 100 into register s7
   blt a3, s7, less_than_100_c         # If a3 < s7, branch to 'less_than_100'
    j end_100_c

less_than_100_c:                        #  Print the zero

    addi a0, x0, 4                      # a0=4 to print string
    la a1, print_zero                   # Syscall code for print character
    ecall

end_100_c:
     li s7, 1000                       # Load 1000 into register s7
    blt a3, s7, less_than_1000_c       # If a3 < s7, branch to 'less_than_1000'
    j end_1000_c

less_than_1000_c:                       # Print the zero

    addi a0, x0, 4                      # a0=4 to print string
    la a1, print_zero                   # Syscall code for print character
    ecall

end_1000_c:
```

Example of Verification Matrix :

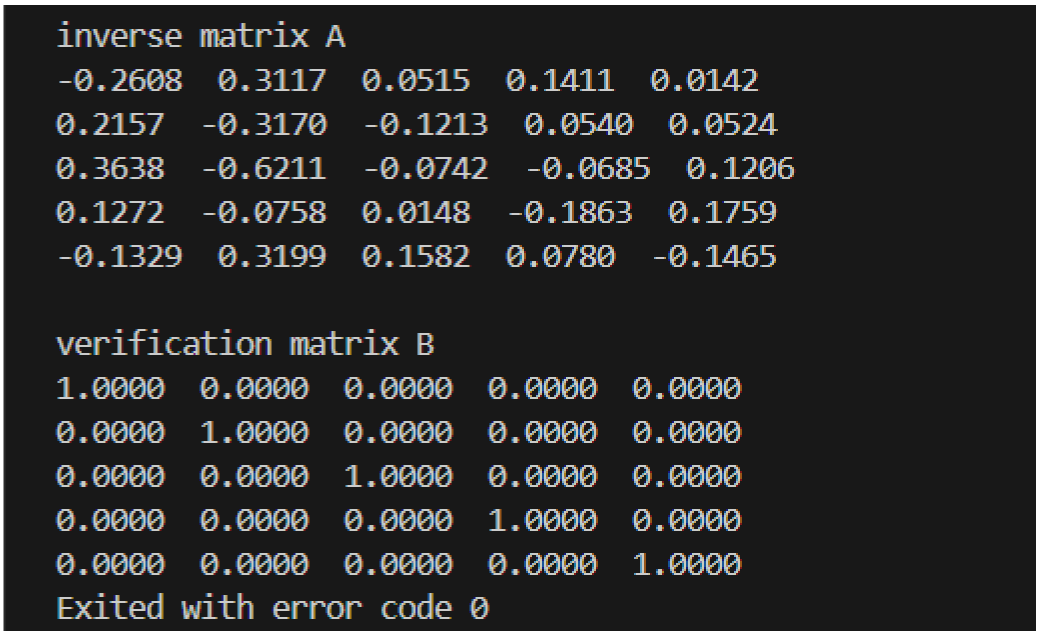


```
verification matrix B
1.0000  0.0000  0.0000  0.0000  0.0000
0.0000  1.0000  0.0000  0.0000  0.0000
0.0000  0.0000  1.0000  0.0000  0.0000
0.0000  0.0000  0.0000  1.0000  0.0000
0.0000  0.0000  0.0000  0.0000  1.0000
Exited with error code 0
```

8) Test Cases:

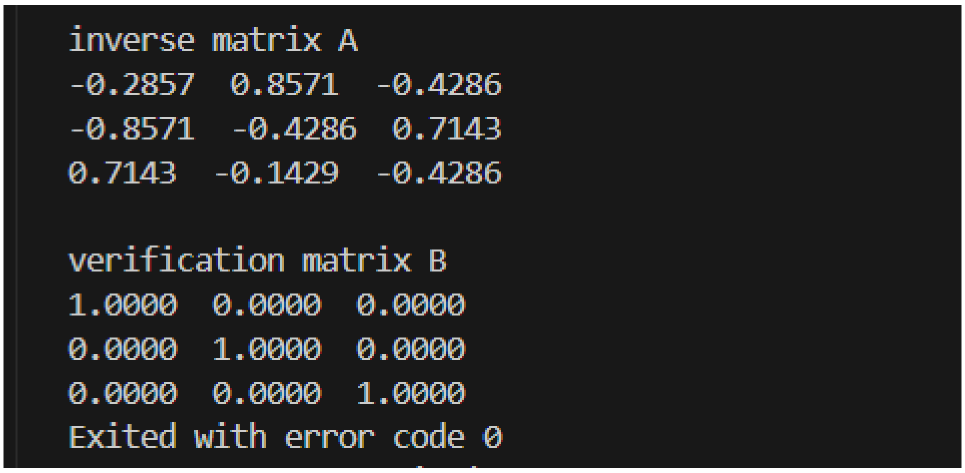
a) N=5

-2, 8, -3, 4, 5
-1, 5, -4, 3, 2
3, -4, 6, 1, 5
4, 5, 1, 0, 3
5, 2, 1, 4, 0



```
inverse matrix A
-0.2608  0.3117  0.0515  0.1411  0.0142
0.2157  -0.3170  -0.1213  0.0540  0.0524
0.3638  -0.6211  -0.0742  -0.0685  0.1206
0.1272  -0.0758  0.0148  -0.1863  0.1759
-0.1329  0.3199  0.1582  0.0780  -0.1465

verification matrix B
1.0000  0.0000  0.0000  0.0000  0.0000
0.0000  1.0000  0.0000  0.0000  0.0000
0.0000  0.0000  1.0000  0.0000  0.0000
0.0000  0.0000  0.0000  1.0000  0.0000
0.0000  0.0000  0.0000  0.0000  1.0000
Exited with error code 0
```

b) N=3
   -2 -3 -3
   -1 -3 -4
   -3 -4 -6



```
inverse matrix A
-0.2857  0.8571  -0.4286
-0.8571  -0.4286  0.7143
0.7143  -0.1429  -0.4286

verification matrix B
1.0000  0.0000  0.0000
0.0000  1.0000  0.0000
0.0000  0.0000  1.0000
Exited with error code 0
```

Conclusion:

In RISC-V assembly using the Venus simulator, calculating the matrix inverse involves implementing matrix operations such as row reductions and elementary transformations. The process typically includes manipulating matrices through nested loops, many condition checks and ensuring proper use of floating-point registers, integer registers and memory addressing. Given the limitations of assembly language and the Venus simulator, efficiently handling floating-point operations and avoiding precision errors are key challenges.