Indian Institute of Technology Delhi

ELL782 COMPUTER ARCHITECTURE, 2023

Report On

Assignment-1: Gaussian Elimination by RISC-V Assembly language and Python

---

Name:	Arnab Haldar
Entry No:	2023EET2756
Course Instructor:	Kaushik Saha

Abstract

This report presents the implementation and analysis of the Gaussian elimination method in the context of RISC-V assembly language programming. To accommodate the limitations of the RISC-V architecture, values are stored in fixed-point notation (12.16 format) for both input matrices and output results. This format uses 12 bits for the integer part, 16 bits for the decimal part, and 4 most significant bits for the sign. The output format varies depending on the nature of the solution: "No solution exists," "Unique solution," or "Infinitely many solutions exist." To validate the correctness of the RISC-V code, a Python script is developed to generate random test cases, solve the Gaussian elimination problem in floating-point format, and convert the results to fixed-point format. Error percentages between the RISC-V code and Python script results are calculated and reported. In summary, this report presents a comprehensive overview of the Gaussian elimination method's implementation in RISC-V assembly language, detailing the process, testing, and analysis conducted as part of the assignment requirements.

Contents

List of Figures

1 Software Requirement Specifications

Specification	Name	Gaussian Elimination for linear System
	Version	1.0
	Implemented Language	RISC-V ISA
	OS Dependent	No
	Prerequisite Software Required	Any Software that can run RISC-ISA with *.s* extension
Functionality	Purpose	Finding the mean error percentage of Solution solved in RISC-ISA and Python
	Input	$5 * 6$ Augmented Matrix in float data type
	Scope	Output only for $5 * 6$ matrix
	Output	No solution, Infinite Solution and Solution vector X5 in case of Unique solution
Execution platform	Processor Type	RISC-V ISA
	Platform	RISC-V ISA
	Specified Version	RV32IMAF
	Memory Requirement	Minimal
	IDE	*Visual Studio Code* for RISC-ISA and *Jupyter Notebook* for Python

2 Design Considerations

- *Visual Studio Code(VS Code)* is required in the machine to execute RISC-V Software.
- *RISC-V Venus Simulator* extension is installed in VS Code with *RISC-V Support* for syntax highlighting and snippets.
- Stored Array Matrix in *Row Major Form.*
- Matrix elements are given as Inputs should be in *float format single precision.* The Double Precision format is not supported by the Venus Simulator.

- Register $a0(x10)$, $a1(x11)$ are not used for any ALU and Memory Operations. They are reserved for system calls (*ecall*) and to avoid any conflicts.
- If the system is consistent, solutions are printed in *IEEE Standard 754 Floating Point Numbers* in *hex* Notation (*Eg.:0x00000000*).
- Conversion required in Python script to convert the RISC-V Solution to Decimal value.
- *Jupyter Notebook* is required in the machine to execute Python Script.
- *Numpy*, *struct*, *random* and *sys* libraries are used in this Python script.
- Error Percentage are calculated in Python Script

3 Architectural Strategies

3.1 Algorithms:

- *Forward Elimination* for conversion of Upper Triangular Matrix.
- *Partial Pivoting* for finding system is consistent or not.
- *Back Substitution* for finding the Solution.

3.2 Approach

- During Development, we divide the Assembly code into multiple blocks by labeling them to control flow.

3.3 Data Types and Sizes

- *Single Precision Floating Point* is used i.e., *32-bit* number is used, *1 bit* for *sign*, *8 bits* for *Exponent*, and *23 bits* for *mantissa*.
- Since we required data in *12.16* binary format, we used Python script to generate random numbers.

4 Software Architecture

4.1 Linear System into Matrix

- Given a set of five linear equations represented as below:

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 + a_{15}x_5 = b_1 \\
a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 + a_{25}x_5 = b_2 \\
a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 + a_{35}x_5 = b_3 \\
a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 + a_{45}x_5 = b_4 \\
a_{51}x_1 + a_{52}x_2 + a_{53}x_3 + a_{54}x_4 + a_{55}x_5 = b_5
\end{aligned}
$$

- We convert this as AX = B, where A is a 5×5 matrix, X and Bis a 5×1 matrix.

Such that : $\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & | & b_1 \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & | & b_2 \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & | & b_3 \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & | & b_4 \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & | & b_5 \end{bmatrix}$

4.2 Gaussian Elimination Method

- Gaussian Elimination can be divided into 3 Subparts.
- Conversion of the augmented matrix into *Upper triangular matrix*.
- Adopting *Partial pivoting* to identify *no solution system* or *infinite solution system* if the matrix is *singular*.
- *Back substitution* applied in Upper triangular matrix to find the Solution Matrix(X)
- We Can refer to Fig.1

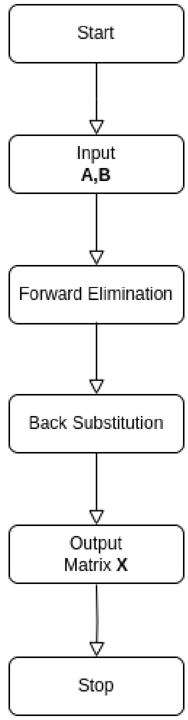

Figure 1: Basic Flow of Gaussian Elimination.

4.3 Upper Triangular Conversion

- For conversion of the Augmented Matrix to an Upper Triangular Matrix, we used the Row Echelon Method.
- Also during conversion, partial pivoting is implemented.
- This process is also called Forward Elimination.
- If any diagonal matrix is found zero, then the Matrix is singular.
- We Can refer to Fig.2

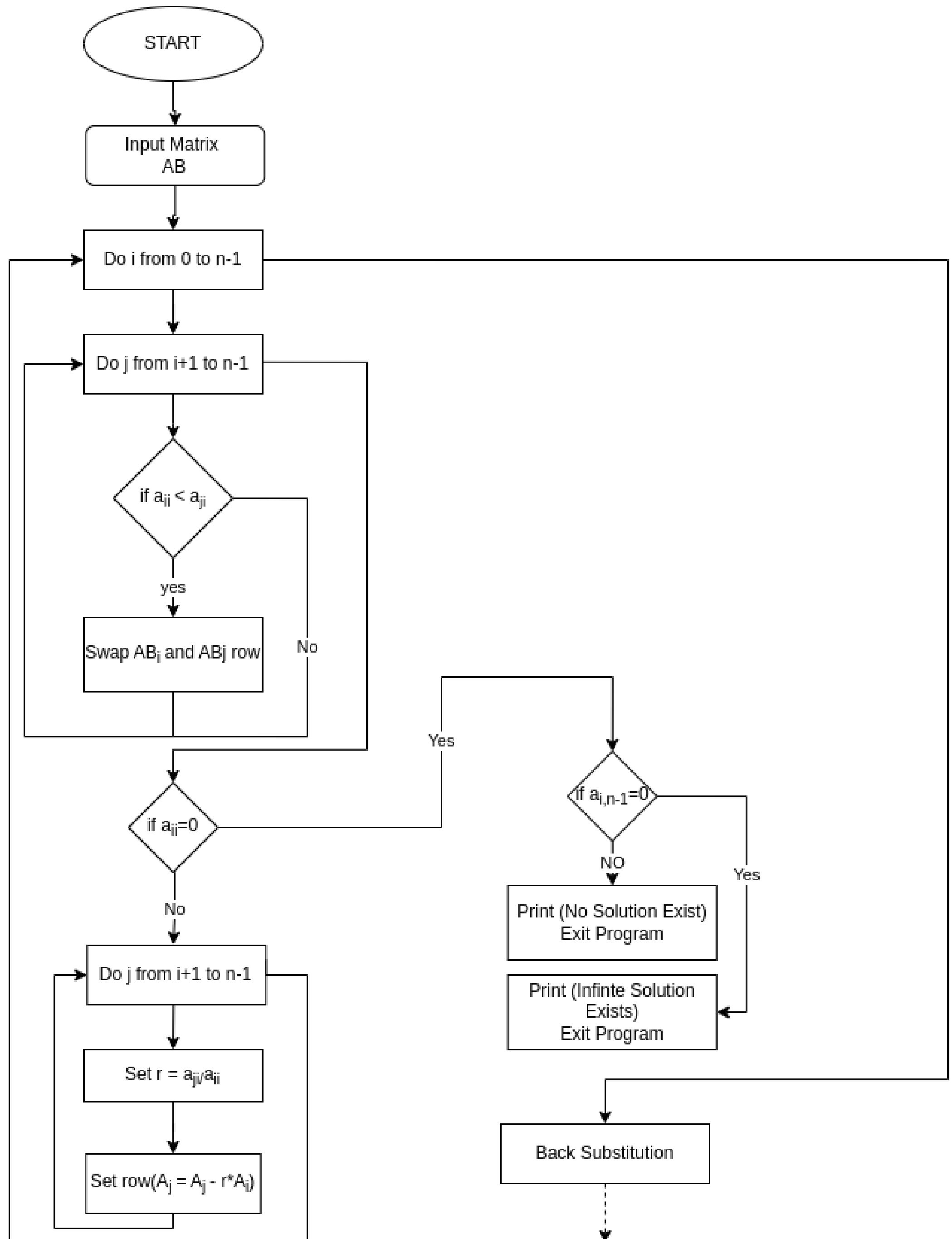

Figure 2: Forward elimination using Row Echelon and Partial Pivoting

4.4 Back Substitution

- For conversion of the Augmented Matrix to an Upper Triangular Matrix, we used the Row Echelon Method.

- Also during conversion, partial pivoting is implemented.
- This process is also called Forward Elimination.
- If any diagonal matrix is found zero, then the Matrix is singular.
- We Can refer to Fig.3

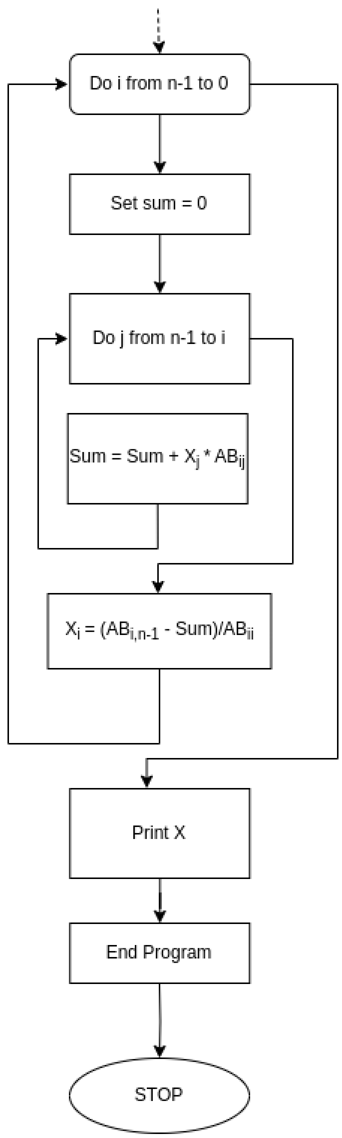

Figure 3: Finding solution using Back Substitution

5 Detailed System Design

5.1 Input of Linear System

- To generate a random set of equations. We used a Python script to find Input values. To get values in the 12.16 binary range, we take random values in the range (-2^{27}, 2^{27}) and divide them with a random number in the range (1, 2^{16}) as shown Fig.4.

```
Random Augmented Matrix Generated:
8543.200734956363 -583.7564186452296 -2754.636553907333 3647.9539166256764 -1446.6787970337818 : -281.783187889581
5

1385.0352102102102 2901.154069507728 -8142.008307181097 1123.314740104986 -2014.709381473809 : -5955.902411539329

25236.103122730572 -776.7793690687698 1995.3801054231478 -62.62675964909027 1028.8135245122276 : 9809.247116360617

-2325.80950321639 5799.913407821229 620.9523550724638 -2874.319851405711 -2548.6485416712735 : -112665.99078341013

-5520.035545410076 -2006.9842974628987 3065.8343272205534 -6928.120853566787 579.042470441817 : -2142.262475758914
```

Figure 4: Input Matrix Generated in Python

- We take this Augmented matrix as inputs in RISC-V Code in *.data* Segment as shown Fig.5.

```
.data
#random generated input from python script
arr : .float 8543.200734956363, -583.7564186452296 ,-2754.636553907333, 3647.9539166256764 ,-1446.6787970337818 ,-281.7831878895815,
             1385.0352102102102, 2901.154069507728, -8142.008307181097, 1123.314740104986, -2014.709381473809, -5955.902411539329,
             25236.103122730572, -776.7793690687698, 1995.3801054231478, -62.62675964909027, 1028.8135245122276, 9809.247116360617,
             -2325.80950321639, 5799.913407821229, 620.9523550724638, -2874.319851405711, -2548.6485416712735, -112665.99078341013,
             -5520.035545410076, -2006.9842974628987, 3065.8343272205534, -6928.120853566787, 579.042470441817, -2142.262475758914
```

Figure 5: Input Matrix values showed in RISC-V

5.2 Pseudo Code

We can divide the Algorithm into two Pseudo Codes. We considered $n = 5$.

- Forward Elimination

    $i \leftarrow 0$
    while $i < n$ **do**
        $j \leftarrow i + 1$
        while $j < n$ **do**
            if $a_{ji} > a_{ii}$ **then**
                Swap a_i row and a_j row
            end if
            $j \leftarrow j + 1$
        end while

if $a_{ii} = 0$ **then**
    if $a_{in-1} = 0$ **then**
        Print('Infinite Solution Exists')
    else
        Print('No Solution Exist')
    end if
    Swap a_i row and a_j row
else
    $j \leftarrow i + 1$
    while $j < n$ **do**
        $ratio \leftarrow a_{ji}/a_{ii}$
        Set row($a_j \leftarrow a_j - ratio * a_i$)
        $j \leftarrow j + 1$
    end while
end if
$i \leftarrow i + 1$
end while

- Back Substitution

$i \leftarrow n - 1$
while $i > 0$ **do**
    $j \leftarrow n - 1$
    $Sum \leftarrow 0$
    while $j > i$ **do**
        $Sum \leftarrow Sum + X_j * a_{ij}$
    end while
    $X_i \leftarrow (a_{in-1} - Sum)/a_{ii}$
    if $a_{ii} = 0$ **then**
        if $a_{in-1} = 0$ **then**
            Print('Infinite Solution Exists')
        else
            Print('No Solution Exist')
        end if
        Swap a_i row and a_j row
    else
        $j \leftarrow i + 1$
        while $j < n$ **do**
            $ratio \leftarrow a_{ji}/a_{ii}$
            Set row($a_j \leftarrow a_j - ratio * a_i$)
            $j \leftarrow j + 1$
        end while
    end if
    $i \leftarrow i + 1$

end while

5.3 Output

- Since Venus Simulator only supports *ecall* id: 34, thus *hex* format supports. Thus Output will be printed in Hex format.

- If the System is consistent then after forward elimination, the output will be in hex format as shown in Fig.6 & Fig.7 which is an upper triangular matrix generated by RISC-ISA and Python respectively.

```
Matrix in Row Echleon:
0x46C52835    0xC44231E1    0x44F96C2A    0xC27A81CD    0x44809A08    0x461944FD
0x00000000    0x45B30298    0x4449366F    0xC5340177    0xC5195D4C    0xC7DA48FA
0x00000000    0x00000000    0x456E0281    0xC5FB2290    0xC3006E53    0xC725E4B7
0x00000000    0xAB000000    0x00000000    0xC674FC80    0xC489CC46    0xC7327DB3
0x00000000    0x29ED70B2    0x00000000    0x00000000    0xC4DFDDB1    0xC710986C
```

Figure 6: Upper Triangular Matrix in RISC-ISA

```
Gaussian Matrix:
 [[ 2.52361031e+04 -7.76779369e+02  1.99538011e+03 -6.26267596e+01
    1.02881352e+03  9.80924712e+03]
  [ 0.00000000e+00  5.72832387e+03  8.04850558e+02 -2.88009166e+03
   -2.45383104e+03 -1.11761951e+05]
  [ 0.00000000e+00  0.00000000e+00  3.80815666e+03 -8.03632004e+03
   -1.28431056e+02 -4.24687192e+04]
  [ 0.00000000e+00  0.00000000e+00  0.00000000e+00 -3.63560007e+03
   -2.04654335e+03 -4.76116603e+04]
  [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
    7.72367227e+03  1.59639429e+05]]
```

Figure 7: Upper Triangular Matrix in Python

- Final output:
  - For Unique Solution as shown in Fig.8

```
Linear System is consistent. Solutions Are:
X[0]: 0xBE105514    X[1]: 0xC10E2DCC    X[2]: 0xC0EBE47D    X[3]: 0x3FBB052C    X[4]: 0x41A559CF    Exited with error code 0
Stop program execution!
```

Figure 8: Unique Solution Output

  - For Infinitely Many Solution as shown in Fig.9

```
Infinitely many solution exists
Exited with error code 0
Stop program execution!
```

Figure 9: Infinitely Many Solutions

– For Unique Solution as shown in Fig.10

```
No solution exists
Exited with error code 0
Stop program execution!
```

Figure 10: No Solutions

- Output of python code as shown in Fig.11 will be used as a standard and referring to find a mean error in Section 6.1

```
Required solution is:
X0 = -0.14094967263053093(hex: 0xbe10551c )     X1 = -8.886181230059561(hex: 0xc10e2dcc )       X2 = -7.3716401342
84201(hex: 0xc0ebe47a ) X3 = 1.4610961731229326(hex: 0x3fbb0533 )       X4 = 20.66885069176456(hex: 0x41a559ce )
```

Figure 11: Output of Python Script

6 Testing

6.1 Conversion of RISC-V code to decimal

- Python Script is created to take the current Hex Output as input and Convert it into Decimal As shown in Fig.12 is the decimal Conversion of Output of RISC-V.

```
X_risc = np.array(['0xBE105514', '0xC10E2DCC', '0xC0EBE47D', '0x3FBB052C', '0x41A559CF'])

for i in  range (0,len(x)):
    decR=struct.unpack('f', struct.pack('I', int(X_risc[i],16)))[0]
    print('IEEE-754 HEX = {} ---> decimal = {}'.format(X_risc[i],decR))
```

```
IEEE-754 HEX = 0xBE105514 ---> decimal = -0.140949547290802
IEEE-754 HEX = 0xC10E2DCC ---> decimal = -8.886180877685547
IEEE-754 HEX = 0xC0EBE47D ---> decimal = -7.371641635894775
IEEE-754 HEX = 0x3FBB052C ---> decimal = 1.4610953330993652
IEEE-754 HEX = 0x41A559CF ---> decimal = 20.668851852416992
```

Figure 12: Decimal Conversion of Output of RISC V

6.2 Calculation of Mean Error

- Mean Error is calculated as $\sum_{n=1}^{5}(Absolute(XRisc_n - XPython_n)/XPython_n) * 100\%$

- $XRisc_n \& XPython_n$ is the n^{th} element of Solution Matrix of RISC-V and Python Output Respectively.

- We can observe the following approach in Fig.13 as we convert the RISC-V Raw output to Decimal
- Error Percentage has been calculated for each element as shown in Fig.13 and *Mean Error* has been calculated as final output.

```
X_risc = np.array(['0xBE105514', '0xC10E2DCC', '0xC0EBE47D', '0x3FBB052C', '0x41A559CF'])
error=np.zeros(n)


for i in  range (0,len(x)):
    decP=x[i]
    decR=struct.unpack('f', struct.pack('I', int(X_risc[i],16)))[0]
    error[i]= abs((decR-decP)/decP)*100
    print('Error Percentage at index {} : {}'.format(i,error[i]))

print('Mean Error Percentage:', np.mean(error))
```

```
Error Percentage at index 0 : 8.892516498260506e-05
Error Percentage at index 1 : 3.96541557378784e-06
Error Percentage at index 2 : 2.037010145905659e-05
Error Percentage at index 3 : 5.749269506173367e-05
Error Percentage at index 4 : 5.615466715409267e-06
Mean Error Percentage: 3.5273768758518485e-05
```

Figure 13: Python snippet of Mean Error Calculation

- For the above output *Mean Error* is $3.5273768758518485 * 10^{-5}\%$ as shown in Fig.13