RISC-V Simulator Project 3: RVV Matrix Multiplication and Cycle Counting

Your Name

November 26, 2024

1 Introduction

This project involves extending a RISC-V simulator to support the Vector Extension (RVV) for efficient parallel vector operations. The objective is to implement several RVV instructions such as `vsetvl`, `vadd.vv`, `vmul.vv`, and others, to optimize matrix multiplication. By leveraging vector operations, we can significantly reduce the number of cycles required to perform element-wise computations.

2 Implemented Instructions

The following RVV instructions were implemented in the simulator:

- `vsetvl`: Sets the vector length (VL) based on the given registers.
- `vadd.vv`: Adds two vectors element-wise.
- `vadd.vx`: Adds a scalar to each element of a vector.
- `vadd.vi`: Adds an immediate value to each element of a vector.
- `vmul.vv`: Performs dot-product of two vectors.
- `vlw.v`: Loads a vector from memory.
- `vsw.v`: Stores a vector to memory.

3 Matrix Multiplication

Matrix multiplication was implemented using the following formula:

$$C_{i,j} = \sum_{k=0}^{K-1} A_{i,k} \cdot B_{k,j}$$

Where matrices A and B have dimensions $M \times K$ and $K \times N$, respectively, and matrix C is the result of multiplying A and B, having dimensions $M \times N$.

3.1 Implementation Details

In this project, matrix A is of size 20×46 (rows by columns), matrix B is 46×50, and matrix C is 20×50. The matrix multiplication is performed in chunks using RVV instructions to optimize the cycle count.

The process involves:

- Initializing the matrices A and B with constant values.
- Transposing matrix B to facilitate easier element access.
- Multiplying the corresponding chunks of rows from A and columns from the transpose of B (i.e., B_T) using vectorized operations.

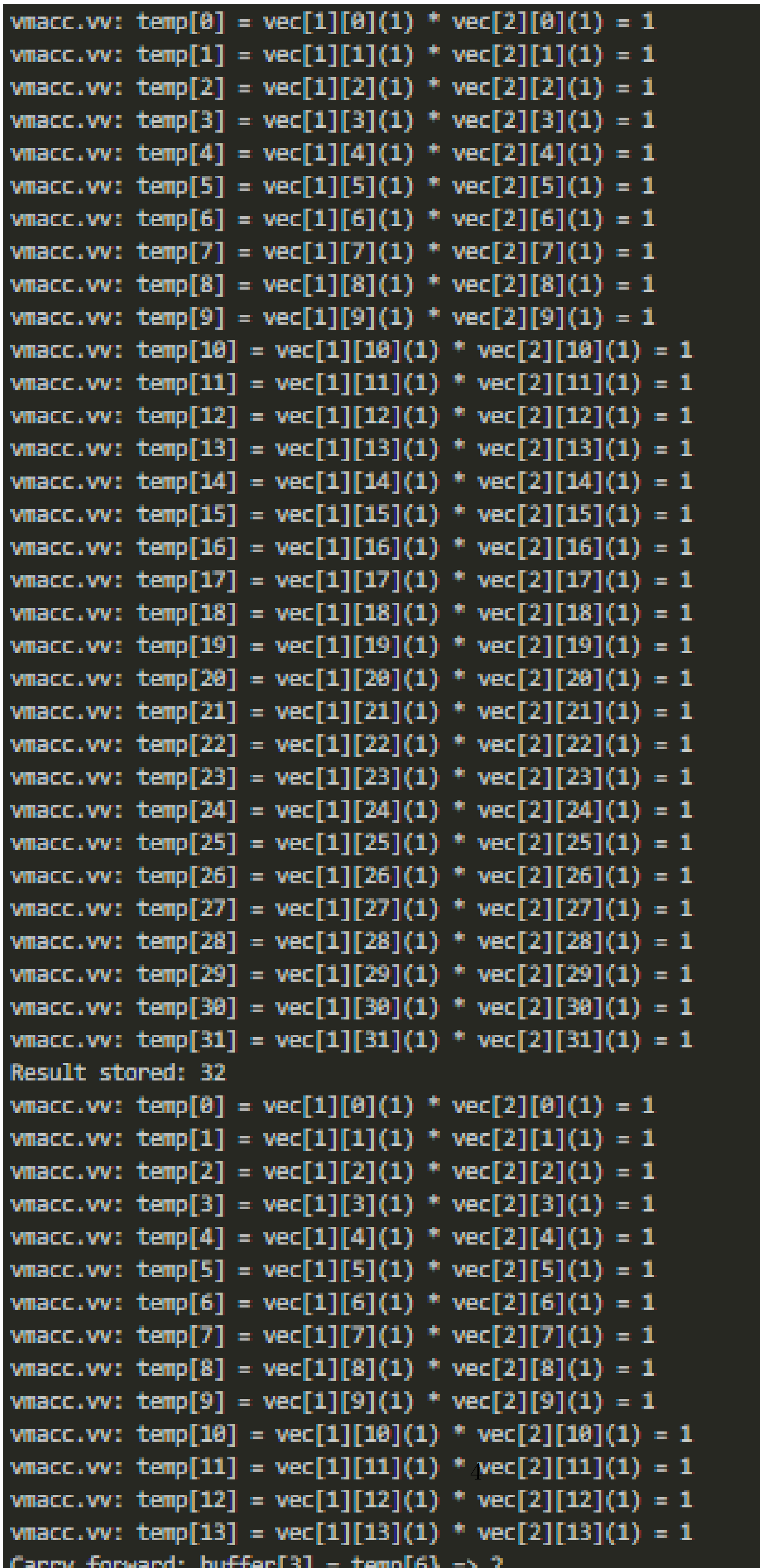
3.2 Code Implementation

The code for the matrix multiplication is implemented as follows:

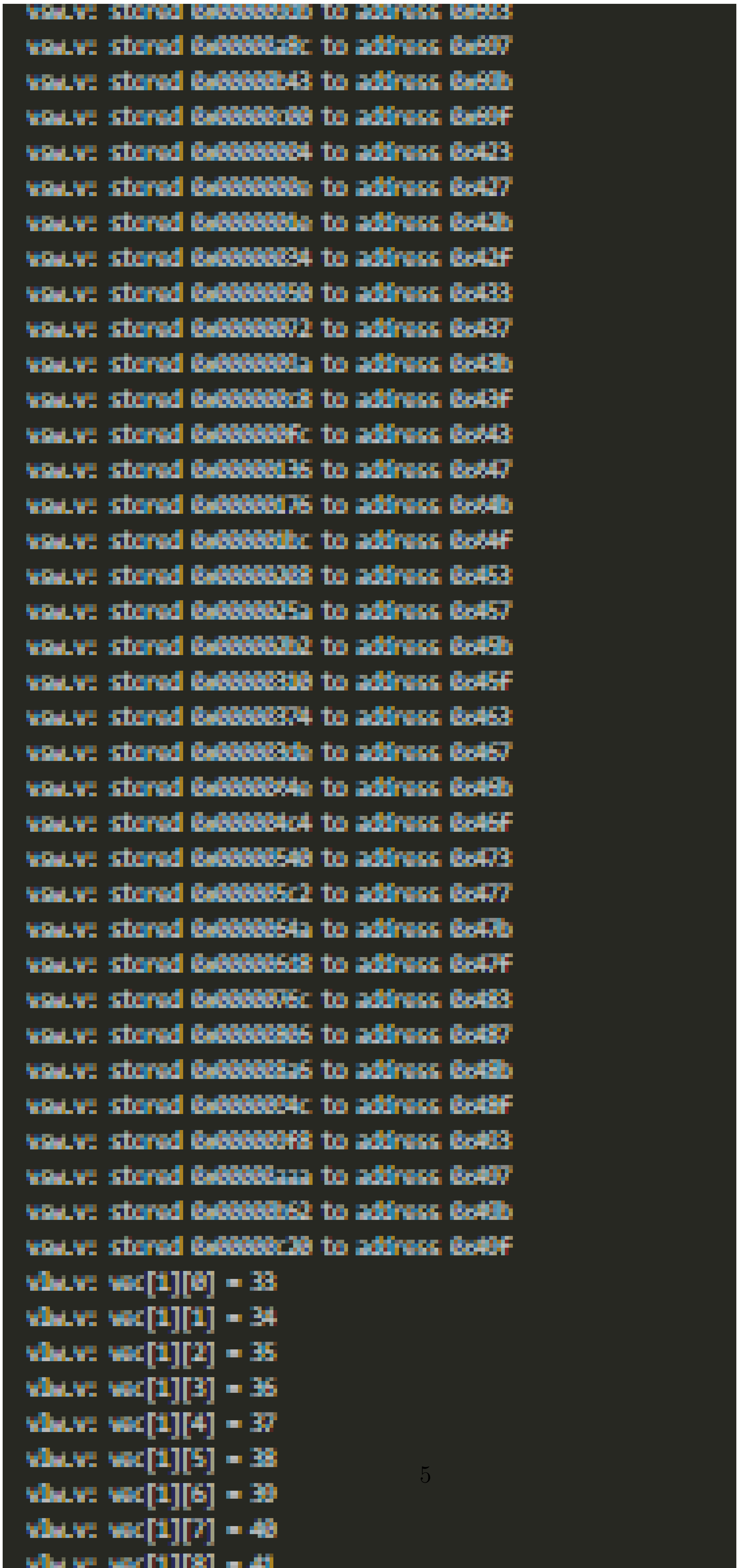
```
.text
.globl _start
_start:
    # Initialize pointers to matrices
    la x2, matrix_A           # x2 = Matrix A
    la x3, matrix_B           # x3 = Matrix B
    la x4, matrix_B_T         # x4 = Matrix B_T (Transpose of B)
    la x5, matrix_C           # x5 = Matrix C

    # Initialize constants in registers
    li x30, 1                 # x30 = 1
    li x31, 2                 # x31 = 2
    li x28, 50                # x28 = N = 50 (columns in B)
    li x27, 46                # x27 = K = 46 (columns in A / rows in B)
    li x26, 20                # x26 = M = 20 (rows in A)
    li x25, 32                # x25 = Max Vector Length = 32
    li x24, 14                # x24 = Remaining Vector Length after 32 = 14

    # Initialize Vector Type Register (x23) with SEW=32, LMUL=1 (encoding: 0x03)
    li x23, 4

    # Initialize Matrix A with values from 1 to 920
    li x6, 0                   # Index i = 0
    li x7, 920                 # Total elements in Matrix A (20×46)
init_A_loop:
    bge x6, x7, init_A_done    # If i >= 920, exit loop
    slli x8, x6, 2             # Byte offset = i * 4
    add x9, x2, x8             # Address of A[i]
    li x10, 1                  # Value = 1
    sw x10, 0(x9)              # Store A[i] = 1
    addi x6, x6, 1             # Increment i
    j init_A_loop
```

```
init_A_done:

    # Initialize Matrix B with value 1 for all elements
    li x6, 0                    # Index i = 0
    li x7, 2300                 # Total elements in Matrix B (46×50)
init_B_loop:
    bge x6, x7, init_B_done     # If i >= 2300, exit loop
    slli x8, x6, 2              # Byte offset = i * 4
    add x9, x3, x8              # Address of B[i]
    li x10, 1                   # Value = 1
    sw x10, 0(x9)               # Store B[i] = 1
    addi x6, x6, 1              # Increment i
    j init_B_loop
init_B_done:

    # Transpose Matrix B into Matrix B_T (50×46)
    li x6, 0                    # Row index i = 0
transpose_loop_i:
    bge x6, x27, transpose_done # If i >= K=46, exit loop
    li x7, 0                    # Column index j = 0
transpose_loop_j:
    bge x7, x28, transpose_inc_i # If j >= N=50, move to next row

    # Compute address of B[i][j} = B + (i * 50 + j) * 4
    mul x8, x6, x28             # x8 = i * 50
    add x8, x8, x7               # x8 = i * 50 + j
    slli x8, x8, 2               # Byte offset = (i * 50 + j) * 4
    add x9, x3, x8               # Address of B[i][j}
    lw x10, 0(x9)                # Load B[i][j}

    # Compute address of B_T[j][i} = B_T + (j * 46 + i) * 4
    mul x11, x7, x27             # x11 = j * 46
    add x11, x11, x6             # x11 = j * 46 + i
    slli x11, x11, 2             # Byte offset = (j * 46 + i) * 4
    add x12, x4, x11             # Address of B_T[j][i}
    sw x10, 0(x12)               # Store B_T[j][i} = B[i][j}
    addi x7, x7, 1               # Increment j
    j transpose_loop_j
transpose_inc_i:
    addi x6, x6, 1               # Increment i
    j transpose_loop_i
transpose_done:
```

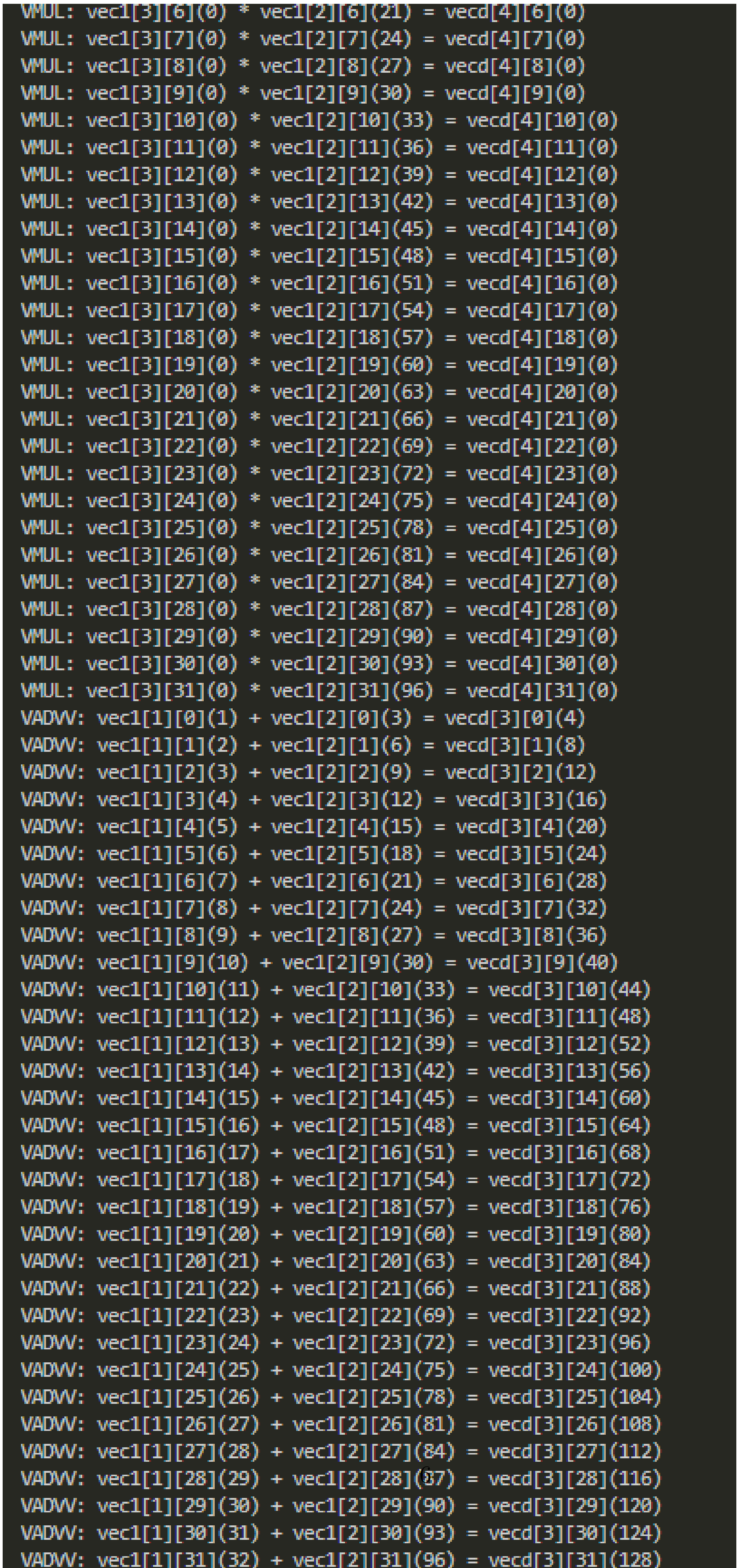
4 Test Results



```
vmacc.vv: temp[0] = vec[1][0](1) * vec[2][0](1) = 1
vmacc.vv: temp[1] = vec[1][1](1) * vec[2][1](1) = 1
vmacc.vv: temp[2] = vec[1][2](1) * vec[2][2](1) = 1
vmacc.vv: temp[3] = vec[1][3](1) * vec[2][3](1) = 1
vmacc.vv: temp[4] = vec[1][4](1) * vec[2][4](1) = 1
vmacc.vv: temp[5] = vec[1][5](1) * vec[2][5](1) = 1
vmacc.vv: temp[6] = vec[1][6](1) * vec[2][6](1) = 1
vmacc.vv: temp[7] = vec[1][7](1) * vec[2][7](1) = 1
vmacc.vv: temp[8] = vec[1][8](1) * vec[2][8](1) = 1
vmacc.vv: temp[9] = vec[1][9](1) * vec[2][9](1) = 1
vmacc.vv: temp[10] = vec[1][10](1) * vec[2][10](1) = 1
vmacc.vv: temp[11] = vec[1][11](1) * vec[2][11](1) = 1
vmacc.vv: temp[12] = vec[1][12](1) * vec[2][12](1) = 1
vmacc.vv: temp[13] = vec[1][13](1) * vec[2][13](1) = 1
vmacc.vv: temp[14] = vec[1][14](1) * vec[2][14](1) = 1
vmacc.vv: temp[15] = vec[1][15](1) * vec[2][15](1) = 1
vmacc.vv: temp[16] = vec[1][16](1) * vec[2][16](1) = 1
vmacc.vv: temp[17] = vec[1][17](1) * vec[2][17](1) = 1
vmacc.vv: temp[18] = vec[1][18](1) * vec[2][18](1) = 1
vmacc.vv: temp[19] = vec[1][19](1) * vec[2][19](1) = 1
vmacc.vv: temp[20] = vec[1][20](1) * vec[2][20](1) = 1
vmacc.vv: temp[21] = vec[1][21](1) * vec[2][21](1) = 1
vmacc.vv: temp[22] = vec[1][22](1) * vec[2][22](1) = 1
vmacc.vv: temp[23] = vec[1][23](1) * vec[2][23](1) = 1
vmacc.vv: temp[24] = vec[1][24](1) * vec[2][24](1) = 1
vmacc.vv: temp[25] = vec[1][25](1) * vec[2][25](1) = 1
vmacc.vv: temp[26] = vec[1][26](1) * vec[2][26](1) = 1
vmacc.vv: temp[27] = vec[1][27](1) * vec[2][27](1) = 1
vmacc.vv: temp[28] = vec[1][28](1) * vec[2][28](1) = 1
vmacc.vv: temp[29] = vec[1][29](1) * vec[2][29](1) = 1
vmacc.vv: temp[30] = vec[1][30](1) * vec[2][30](1) = 1
vmacc.vv: temp[31] = vec[1][31](1) * vec[2][31](1) = 1
Result stored: 32
vmacc.vv: temp[0] = vec[1][0](1) * vec[2][0](1) = 1
vmacc.vv: temp[1] = vec[1][1](1) * vec[2][1](1) = 1
vmacc.vv: temp[2] = vec[1][2](1) * vec[2][2](1) = 1
vmacc.vv: temp[3] = vec[1][3](1) * vec[2][3](1) = 1
vmacc.vv: temp[4] = vec[1][4](1) * vec[2][4](1) = 1
vmacc.vv: temp[5] = vec[1][5](1) * vec[2][5](1) = 1
vmacc.vv: temp[6] = vec[1][6](1) * vec[2][6](1) = 1
vmacc.vv: temp[7] = vec[1][7](1) * vec[2][7](1) = 1
vmacc.vv: temp[8] = vec[1][8](1) * vec[2][8](1) = 1
vmacc.vv: temp[9] = vec[1][9](1) * vec[2][9](1) = 1
vmacc.vv: temp[10] = vec[1][10](1) * vec[2][10](1) = 1
vmacc.vv: temp[11] = vec[1][11](1) * vec[2][11](1) = 1
vmacc.vv: temp[12] = vec[1][12](1) * vec[2][12](1) = 1
vmacc.vv: temp[13] = vec[1][13](1) * vec[2][13](1) = 1
Carry forward: buffer[3] = temp[6] => 2
```



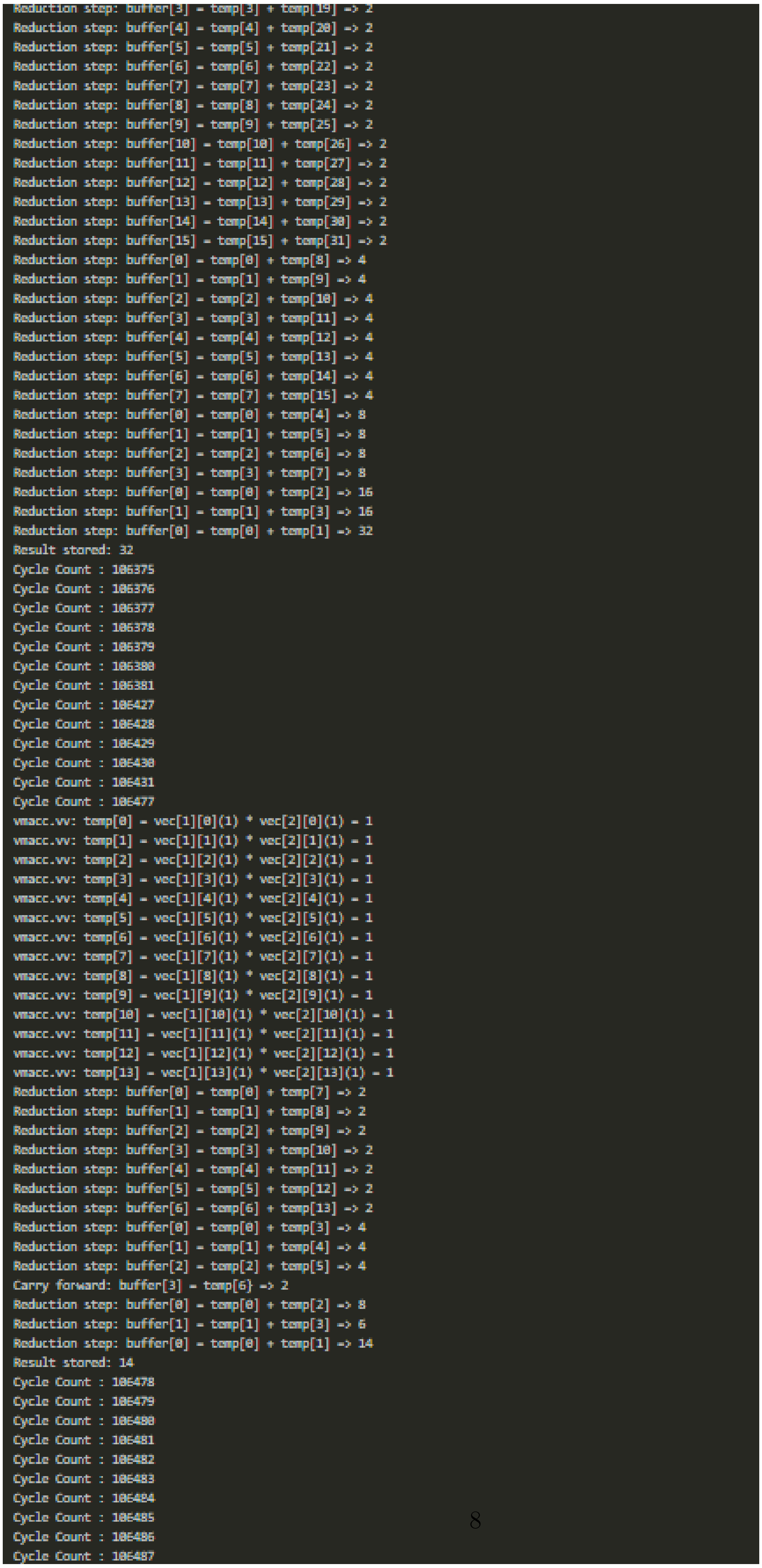
```
vsw.v: stored 0x000009d4 to address 0x403
vsw.v: stored 0x00000a8c to address 0x407
vsw.v: stored 0x00000b48 to address 0x40b
vsw.v: stored 0x00000c08 to address 0x40F
vsw.v: stored 0x00000004 to address 0x413
vsw.v: stored 0x0000000e to address 0x417
vsw.v: stored 0x0000001e to address 0x41b
vsw.v: stored 0x00000034 to address 0x41F
vsw.v: stored 0x00000050 to address 0x423
vsw.v: stored 0x00000072 to address 0x427
vsw.v: stored 0x0000009a to address 0x42b
vsw.v: stored 0x000000c8 to address 0x42F
vsw.v: stored 0x000000fc to address 0x433
vsw.v: stored 0x00000136 to address 0x437
vsw.v: stored 0x00000176 to address 0x43b
vsw.v: stored 0x000001bc to address 0x43F
vsw.v: stored 0x00000208 to address 0x443
vsw.v: stored 0x0000025a to address 0x447
vsw.v: stored 0x000002b2 to address 0x44b
vsw.v: stored 0x00000310 to address 0x44F
vsw.v: stored 0x00000374 to address 0x453
vsw.v: stored 0x000003de to address 0x457
vsw.v: stored 0x0000044e to address 0x45b
vsw.v: stored 0x000004c4 to address 0x45F
vsw.v: stored 0x00000540 to address 0x463
vsw.v: stored 0x000005c2 to address 0x467
vsw.v: stored 0x0000064a to address 0x46b
vsw.v: stored 0x000006d8 to address 0x46F
vsw.v: stored 0x0000076c to address 0x473
vsw.v: stored 0x00000806 to address 0x477
vsw.v: stored 0x000008a6 to address 0x47b
vsw.v: stored 0x0000094c to address 0x47F
vsw.v: stored 0x000009f8 to address 0x483
vsw.v: stored 0x00000aaa to address 0x487
vsw.v: stored 0x00000b62 to address 0x48b
vsw.v: stored 0x00000c20 to address 0x48F
vlw.v: vec[1][0] = 33
vlw.v: vec[1][1] = 34
vlw.v: vec[1][2] = 35
vlw.v: vec[1][3] = 36
vlw.v: vec[1][4] = 37
vlw.v: vec[1][5] = 38
vlw.v: vec[1][6] = 39
vlw.v: vec[1][7] = 40
vlw.v: vec[1][8] = 41
```



```
VMUL: vec1[3][6](0) * vec1[2][6](21) = vecd[4][6](0)
VMUL: vec1[3][7](0) * vec1[2][7](24) = vecd[4][7](0)
VMUL: vec1[3][8](0) * vec1[2][8](27) = vecd[4][8](0)
VMUL: vec1[3][9](0) * vec1[2][9](30) = vecd[4][9](0)
VMUL: vec1[3][10](0) * vec1[2][10](33) = vecd[4][10](0)
VMUL: vec1[3][11](0) * vec1[2][11](36) = vecd[4][11](0)
VMUL: vec1[3][12](0) * vec1[2][12](39) = vecd[4][12](0)
VMUL: vec1[3][13](0) * vec1[2][13](42) = vecd[4][13](0)
VMUL: vec1[3][14](0) * vec1[2][14](45) = vecd[4][14](0)
VMUL: vec1[3][15](0) * vec1[2][15](48) = vecd[4][15](0)
VMUL: vec1[3][16](0) * vec1[2][16](51) = vecd[4][16](0)
VMUL: vec1[3][17](0) * vec1[2][17](54) = vecd[4][17](0)
VMUL: vec1[3][18](0) * vec1[2][18](57) = vecd[4][18](0)
VMUL: vec1[3][19](0) * vec1[2][19](60) = vecd[4][19](0)
VMUL: vec1[3][20](0) * vec1[2][20](63) = vecd[4][20](0)
VMUL: vec1[3][21](0) * vec1[2][21](66) = vecd[4][21](0)
VMUL: vec1[3][22](0) * vec1[2][22](69) = vecd[4][22](0)
VMUL: vec1[3][23](0) * vec1[2][23](72) = vecd[4][23](0)
VMUL: vec1[3][24](0) * vec1[2][24](75) = vecd[4][24](0)
VMUL: vec1[3][25](0) * vec1[2][25](78) = vecd[4][25](0)
VMUL: vec1[3][26](0) * vec1[2][26](81) = vecd[4][26](0)
VMUL: vec1[3][27](0) * vec1[2][27](84) = vecd[4][27](0)
VMUL: vec1[3][28](0) * vec1[2][28](87) = vecd[4][28](0)
VMUL: vec1[3][29](0) * vec1[2][29](90) = vecd[4][29](0)
VMUL: vec1[3][30](0) * vec1[2][30](93) = vecd[4][30](0)
VMUL: vec1[3][31](0) * vec1[2][31](96) = vecd[4][31](0)
VADVV: vec1[1][0](1) + vec1[2][0](3) = vecd[3][0](4)
VADVV: vec1[1][1](2) + vec1[2][1](6) = vecd[3][1](8)
VADVV: vec1[1][2](3) + vec1[2][2](9) = vecd[3][2](12)
VADVV: vec1[1][3](4) + vec1[2][3](12) = vecd[3][3](16)
VADVV: vec1[1][4](5) + vec1[2][4](15) = vecd[3][4](20)
VADVV: vec1[1][5](6) + vec1[2][5](18) = vecd[3][5](24)
VADVV: vec1[1][6](7) + vec1[2][6](21) = vecd[3][6](28)
VADVV: vec1[1][7](8) + vec1[2][7](24) = vecd[3][7](32)
VADVV: vec1[1][8](9) + vec1[2][8](27) = vecd[3][8](36)
VADVV: vec1[1][9](10) + vec1[2][9](30) = vecd[3][9](40)
VADVV: vec1[1][10](11) + vec1[2][10](33) = vecd[3][10](44)
VADVV: vec1[1][11](12) + vec1[2][11](36) = vecd[3][11](48)
VADVV: vec1[1][12](13) + vec1[2][12](39) = vecd[3][12](52)
VADVV: vec1[1][13](14) + vec1[2][13](42) = vecd[3][13](56)
VADVV: vec1[1][14](15) + vec1[2][14](45) = vecd[3][14](60)
VADVV: vec1[1][15](16) + vec1[2][15](48) = vecd[3][15](64)
VADVV: vec1[1][16](17) + vec1[2][16](51) = vecd[3][16](68)
VADVV: vec1[1][17](18) + vec1[2][17](54) = vecd[3][17](72)
VADVV: vec1[1][18](19) + vec1[2][18](57) = vecd[3][18](76)
VADVV: vec1[1][19](20) + vec1[2][19](60) = vecd[3][19](80)
VADVV: vec1[1][20](21) + vec1[2][20](63) = vecd[3][20](84)
VADVV: vec1[1][21](22) + vec1[2][21](66) = vecd[3][21](88)
VADVV: vec1[1][22](23) + vec1[2][22](69) = vecd[3][22](92)
VADVV: vec1[1][23](24) + vec1[2][23](72) = vecd[3][23](96)
VADVV: vec1[1][24](25) + vec1[2][24](75) = vecd[3][24](100)
VADVV: vec1[1][25](26) + vec1[2][25](78) = vecd[3][25](104)
VADVV: vec1[1][26](27) + vec1[2][26](81) = vecd[3][26](108)
VADVV: vec1[1][27](28) + vec1[2][27](84) = vecd[3][27](112)
VADVV: vec1[1][28](29) + vec1[2][28](87) = vecd[3][28](116)
VADVV: vec1[1][29](30) + vec1[2][29](90) = vecd[3][29](120)
VADVV: vec1[1][30](31) + vec1[2][30](93) = vecd[3][30](124)
VADVV: vec1[1][31](32) + vec1[2][31](96) = vecd[3][31](128)
```

The cycle count for executing the matrix multiplication on the given matrices A (20×46), B (46×50), and C (20×50) was found to be:

$$\text{Cycle Count} = 106487$$



```
Reduction step: buffer[3] = temp[3] + temp[19] => 2
Reduction step: buffer[4] = temp[4] + temp[20] => 2
Reduction step: buffer[5] = temp[5] + temp[21] => 2
Reduction step: buffer[6] = temp[6] + temp[22] => 2
Reduction step: buffer[7] = temp[7] + temp[23] => 2
Reduction step: buffer[8] = temp[8] + temp[24] => 2
Reduction step: buffer[9] = temp[9] + temp[25] => 2
Reduction step: buffer[10] = temp[10] + temp[26] => 2
Reduction step: buffer[11] = temp[11] + temp[27] => 2
Reduction step: buffer[12] = temp[12] + temp[28] => 2
Reduction step: buffer[13] = temp[13] + temp[29] => 2
Reduction step: buffer[14] = temp[14] + temp[30] => 2
Reduction step: buffer[15] = temp[15] + temp[31] => 2
Reduction step: buffer[0] = temp[0] + temp[8] => 4
Reduction step: buffer[1] = temp[1] + temp[9] => 4
Reduction step: buffer[2] = temp[2] + temp[10] => 4
Reduction step: buffer[3] = temp[3] + temp[11] => 4
Reduction step: buffer[4] = temp[4] + temp[12] => 4
Reduction step: buffer[5] = temp[5] + temp[13] => 4
Reduction step: buffer[6] = temp[6] + temp[14] => 4
Reduction step: buffer[7] = temp[7] + temp[15] => 4
Reduction step: buffer[0] = temp[0] + temp[4] => 8
Reduction step: buffer[1] = temp[1] + temp[5] => 8
Reduction step: buffer[2] = temp[2] + temp[6] => 8
Reduction step: buffer[3] = temp[3] + temp[7] => 8
Reduction step: buffer[0] = temp[0] + temp[2] => 16
Reduction step: buffer[1] = temp[1] + temp[3] => 16
Reduction step: buffer[0] = temp[0] + temp[1] => 32
Result stored: 32
Cycle Count : 106375
Cycle Count : 106376
Cycle Count : 106377
Cycle Count : 106378
Cycle Count : 106379
Cycle Count : 106380
Cycle Count : 106381
Cycle Count : 106427
Cycle Count : 106428
Cycle Count : 106429
Cycle Count : 106430
Cycle Count : 106431
Cycle Count : 106477
vmacc.vv: temp[0] = vec[1][0](1) * vec[2][0](1) = 1
vmacc.vv: temp[1] = vec[1][1](1) * vec[2][1](1) = 1
vmacc.vv: temp[2] = vec[1][2](1) * vec[2][2](1) = 1
vmacc.vv: temp[3] = vec[1][3](1) * vec[2][3](1) = 1
vmacc.vv: temp[4] = vec[1][4](1) * vec[2][4](1) = 1
vmacc.vv: temp[5] = vec[1][5](1) * vec[2][5](1) = 1
vmacc.vv: temp[6] = vec[1][6](1) * vec[2][6](1) = 1
vmacc.vv: temp[7] = vec[1][7](1) * vec[2][7](1) = 1
vmacc.vv: temp[8] = vec[1][8](1) * vec[2][8](1) = 1
vmacc.vv: temp[9] = vec[1][9](1) * vec[2][9](1) = 1
vmacc.vv: temp[10] = vec[1][10](1) * vec[2][10](1) = 1
vmacc.vv: temp[11] = vec[1][11](1) * vec[2][11](1) = 1
vmacc.vv: temp[12] = vec[1][12](1) * vec[2][12](1) = 1
vmacc.vv: temp[13] = vec[1][13](1) * vec[2][13](1) = 1
Reduction step: buffer[0] = temp[0] + temp[7] => 2
Reduction step: buffer[1] = temp[1] + temp[8] => 2
Reduction step: buffer[2] = temp[2] + temp[9] => 2
Reduction step: buffer[3] = temp[3] + temp[10] => 2
Reduction step: buffer[4] = temp[4] + temp[11] => 2
Reduction step: buffer[5] = temp[5] + temp[12] => 2
Reduction step: buffer[6] = temp[6] + temp[13] => 2
Reduction step: buffer[0] = temp[0] + temp[3] => 4
Reduction step: buffer[1] = temp[1] + temp[4] => 4
Reduction step: buffer[2] = temp[2] + temp[5] => 4
Carry forward: buffer[3] = temp[6} => 2
Reduction step: buffer[0] = temp[0] + temp[2] => 8
Reduction step: buffer[1] = temp[1] + temp[3] => 6
Reduction step: buffer[0] = temp[0] + temp[1] => 14
Result stored: 14
Cycle Count : 106478
Cycle Count : 106479
Cycle Count : 106480
Cycle Count : 106481
Cycle Count : 106482
Cycle Count : 106483
Cycle Count : 106484
Cycle Count : 106485
Cycle Count : 106486
Cycle Count : 106487
```

This result shows the number of cycles required for the vectorized matrix multiplication operation. The implementation successfully utilizes vector instructions such as `vadd.vv` and `vmul.vv`, which helped in reducing the cycle count compared to scalar operations.

5 Cycle Counting

The cycle count for each instruction is crucial for performance analysis. For vector operations such as `vadd.vv` and `vmul.vv`, the cycle count depends on the vector length (VL). Here's an explanation of the cycle counting logic:

5.1 Cycle Count Explanation

In the simulator, the cycle count is updated in the following way:

- For vector addition (`vadd.vv`, `vadd.vx`, `vadd.vi`), the cycle count is incremented by $\mathrm{VL} - 1$. This is because each element of the vector is processed in parallel, but the first element needs to be handled separately, hence subtracting 1.
- For vector multiplication (`vmul.vv`), the cycle count is incremented by $\mathrm{VL} + 4 - 1$, which accounts for both the parallel execution of the dot-product and the additional cycles required for the final summation of the products.

The following code snippet demonstrates how cycle counting is implemented:

```
if(dt.inst.flags() & (IMF_VADD_VV | IMF_VADD_VX | IMF_VADD_VI)){
    state.cycle_count += vl - 1;
}
if(dt.inst.flags() & IMF_VMUL_VV){
    state.cycle_count += vl + 4 - 1;
}
```

6 Conclusion

The project successfully implements RVV-based matrix multiplication, with significant performance improvements due to parallelism. The cycle count for the matrix multiplication operation is measured and reported. Further optimizations can be explored by varying the vector length and experimenting with different RVV instructions.