COMPILER DESIGN

DIGITAL ASSIGNMENT 3

TEAM MEMBERS:
S.MUKUNTH(22BRS1021)
KAVIN KARTHIK V(22BRS1049)
NANTHAN S NAIR(22BRS1070)

CODES AND THEIR DESCRIPTION:

MATRIX:

```
#include <iostream>
using namespace std;

void multiply(int A[2][2], int B[2][2], int C[2][2]) {
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            C[i][j] = 0;
            for (int k = 0; k < 2; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    int A[2][2] = {{1, 2}, {3, 4}};
    int B[2][2] = {{5, 6}, {7, 8}};
    int C[2][2];

    multiply(A, B, C);

    cout << "Result Matrix:" << endl;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            cout << C[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

1. Function Definition ('multiply'):
   - The function `multiply` takes three 2x2 matrices (A,B, and C) as input.
   - It uses three nested loops:
     - The outer loop iterates over rows of matrix A.
     - The middle loop iterates over columns of matrix B.
     - The innermost loop performs the dot product calculation for each element in the resulting matrix.

2. Matrix Initialization:
   - In the main function, matrices A and B are initialized with predefined values:

A = {{1, 2}, {3, 4}}
B = {{5, 6}, {7, 8}}
   - Matrix C is declared to store the result.

3. Output:
   - After calling the multiply function, the program prints the resulting matrix (C) using nested loops.

```
nanthan@nanthan-HP-240-G7-Notebook-PC:~$ clang++ -S -emit-llvm matrix.cpp -o matrix.ll
nanthan@nanthan-HP-240-G7-Notebook-PC:~$ cat matrix.ll
; ModuleID = 'matrix.cpp'
source_filename = "matrix.cpp"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

%"class.std::ios_base::Init" = type { i8 }
%"class.std::basic_ostream" = type { i32 (...)**, %"class.std::basic_ios" }
%"class.std::basic_ios" = type { %"class.std::ios_base", %"class.std::basic_ostream"*, i8, i8, %"class.std::basic_streambuf"*, %"class.std::ctype"*, %"class.std::num_put
"*, %"class.std::num_get"* }
%"class.std::ios_base" = type { i32 (...)**, i64, i64, i32, i32, i32, %"struct.std::ios_base::_Callback_list"*, %"struct.std::ios_base::_Words", [8 x %"struct.std::ios_b
ase::_Words"], i32, %"struct.std::ios_base::_Words"*, %"class.std::locale" }
%"struct.std::ios_base::_Callback_list" = type { %"struct.std::ios_base::_Callback_list"*, void (i32, %"class.std::ios_base"*, i32)*, i32, i32 }
%"struct.std::ios_base::_Words" = type { i8*, i64 }
%"class.std::locale" = type { %"class.std::locale::_Impl"* }
%"class.std::locale::_Impl" = type { i32, %"class.std::locale::facet"**, i64, %"class.std::locale::facet"**, i8** }
%"class.std::locale::facet" = type <{ i32 (...)**, i32, [4 x i8] }>
%"class.std::basic_streambuf" = type { i32 (...)**, i8*, i8*, i8*, i8*, i8*, i8*, %"class.std::locale" }
%"class.std::ctype" = type <{ %"class.std::locale::facet.base", [4 x i8], %struct.__locale_struct*, i8, [7 x i8], i32*, i32*, i16*, i8, [256 x i8], [256 x i8], i8, [6 x
i8] }>
%"class.std::locale::facet.base" = type <{ i32 (...)**, i32 }>
%struct.__locale_struct = type { [13 x %struct.__locale_data*], i16*, i32*, i32*, [13 x i8*] }
%struct.__locale_data = type opaque
%"class.std::num_put" = type { %"class.std::locale::facet.base", [4 x i8] }
%"class.std::num_get" = type { %"class.std::locale::facet.base", [4 x i8] }

@_ZStL8__ioinit = internal global %"class.std::ios_base::Init" zeroinitializer, align 1
@__dso_handle = external hidden global i8
@__const.main.A = private unnamed_addr constant [2 x [2 x i32]] [[2 x i32] [i32 1, i32 2], [2 x i32] [i32 3, i32 4]], align 16
@__const.main.B = private unnamed_addr constant [2 x [2 x i32]] [[2 x i32] [i32 5, i32 6], [2 x i32] [i32 7, i32 8]], align 16
@_ZSt4cout = external global %"class.std::basic_ostream", align 8
@.str = private unnamed_addr constant [15 x i8] c"Result Matrix:\00", align 1
```

1. Compilation Command:
   - The user runs the command clang++ -S -emit-llvm matrix.cpp -o matrix.ll, which compiles the matrix.cpp file into LLVM IR. The output is saved in a file named matrix.ll.

2. LLVM IR File Content:

- The content of the matrix.ll file is displayed using the cat matrix.ll command. It contains LLVM IR code generated from the C++ source file.
- The IR includes metadata about the source file (matrix.cpp), target architecture (x86_64-pc-linux-gnu), and various data structures and global variables.

3. Matrix Data Representation:

The IR defines constants for two matrices (A and B) initialized with values:

@const.main.A = private unnamed addr constant [2 x [2 x i32]] [[i32 1, i32 2], [i32 3, i32 4]], align 16

@const.main.B = private unnamed addr constant [2 x [2 x i32]] [[i32 5, i32 6], [i32 7, i32 8]], align 16

```

- These matrices are likely used for matrix multiplication in the program.

4. Result Matrix String:

- A string constant Result Matrix:\00 is defined in the IR, which will be used for displaying output.

```
; Function Attrs: nounwind
declare void @_ZNSt8ios_base4InitD1Ev(%"class.std::ios_base::Init"* noundef nonnull align 1 dereferenceable(1)) unnamed_addr #2

; Function Attrs: nounwind
declare i32 @__cxa_atexit(void (i8*)*, i8*, i8*) #3

; Function Attrs: mustprogress noinline nounwind optnone uwtable
define dso_local void @_Z8multiplyPA2_iS0_S0_([2 x i32]* noundef %0, [2 x i32]* noundef %1, [2 x i32]* noundef %2) #4 {
  %4 = alloca [2 x i32]*, align 8
  %5 = alloca [2 x i32]*, align 8
  %6 = alloca [2 x i32]*, align 8
  %7 = alloca i32, align 4
  %8 = alloca i32, align 4
  %9 = alloca i32, align 4
  store [2 x i32]* %0, [2 x i32]** %4, align 8
  store [2 x i32]* %1, [2 x i32]** %5, align 8
  store [2 x i32]* %2, [2 x i32]** %6, align 8
  store i32 0, i32* %7, align 4
  br label %10

10:   ; preds = %63, %3
  %11 = load i32, i32* %7, align 4
  %12 = icmp slt i32 %11, 2
  br i1 %12, label %13, label %66

13:   ; preds = %10
  store i32 0, i32* %8, align 4
  br label %14

14:   ; preds = %59, %13
  %15 = load i32, i32* %8, align 4
  %16 = icmp slt i32 %15, 2
  br i1 %16, label %17, label %62

17:   ; preds = %14
  %18 = load [2 x i32]*, [2 x i32]** %6, align 8
  %19 = load i32, i32* %7, align 4
  %20 = sext i32 %19 to i64
  %21 = getelementptr inbounds [2 x i32], [2 x i32]* %18, i64 %20
```
```

This image contains LLVM intermediate representation (IR) code, which is a low-level representation of a program. The code appears to define functions with attributes such as nounwind and optnone, and it includes memory allocations (alloca), pointer manipulations, and control flow (br instructions). Specifically, the function shown in the image processes 2x2 integer matrices, storing and loading values while performing comparisons and branching. The structure suggests that it might be part of a compiled C++ program, potentially implementing matrix multiplication or similar operations.

```
17:                                               ; preds = %14
  %18 = load [2 x i32]*, [2 x i32]** %6, align 8
  %19 = load i32, i32* %7, align 4
  %20 = sext i32 %19 to i64
  %21 = getelementptr inbounds [2 x i32], [2 x i32]* %18, i64 %20
  %22 = load i32, i32* %8, align 4
  %23 = sext i32 %22 to i64
  %24 = getelementptr inbounds [2 x i32], [2 x i32]* %21, i64 0, i64 %23
  store i32 0, i32* %24, align 4
  store i32 0, i32* %9, align 4
  br label %25

25:                                               ; preds = %55, %17
  %26 = load i32, i32* %9, align 4
  %27 = icmp slt i32 %26, 2
  br i1 %27, label %28, label %58

28:                                               ; preds = %25
  %29 = load [2 x i32]*, [2 x i32]** %4, align 8
  %30 = load i32, i32* %7, align 4
  %31 = sext i32 %30 to i64
  %32 = getelementptr inbounds [2 x i32], [2 x i32]* %29, i64 %31
  %33 = load i32, i32* %9, align 4
  %34 = sext i32 %33 to i64
  %35 = getelementptr inbounds [2 x i32], [2 x i32]* %32, i64 0, i64 %34
  %36 = load i32, i32* %35, align 4
  %37 = load [2 x i32]*, [2 x i32]** %5, align 8
  %38 = load i32, i32* %9, align 4
  %39 = sext i32 %38 to i64
  %40 = getelementptr inbounds [2 x i32], [2 x i32]* %37, i64 %39
  %41 = load i32, i32* %8, align 4
  %42 = sext i32 %41 to i64
  %43 = getelementptr inbounds [2 x i32], [2 x i32]* %40, i64 0, i64 %42
  %44 = load i32, i32* %43, align 4
  %45 = mul nsw i32 %36, %44
  %46 = load [2 x i32]*, [2 x i32]** %6, align 8
  %47 = load i32, i32* %7, align 4
```

This image displays LLVM IR code, continuing from the previous function. It includes pointer manipulations, integer extensions (sext), and memory access operations (load, store). The code references 2x2 integer arrays and performs index calculations using getelementptr. Additionally, control flow instructions (br) indicate loop execution, likely iterating over matrix elements. The presence of multiple load and store instructions suggests this code is part of a matrix computation, possibly a multiplication or accumulation step in a nested loop.

```
!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 1}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{!"Ubuntu clang version 14.0.0-1ubuntu1.1"}
!6 = distinct !{!6, !7}
!7 = !{!"llvm.loop.mustprogress"}
!8 = distinct !{!8, !7}
!9 = distinct !{!9, !7}
!10 = distinct !{!10, !7}
!11 = distinct !{!11, !7}
```

This image shows LLVM metadata, which provides additional information about the compiled program. It includes llvm.module.flags, specifying properties like wchar_size, PIC Level (Position Independent Code), PIE Level (Position Independent Executable), uwtable (unwind table for exception handling), and frame-pointer settings. The llvm.ident metadata identifies the compiler version, in this case, Ubuntu clang version 14.0.0. Additionally, there are distinct metadata nodes related to loop properties, such as llvm.loop.mustprogress, which ensures loop execution is treated as always making progress.

```
nanthan@nanthan-HP-240-G7-Notebook-PC:~$ python3 translate.py
pPIM ISA Generated! Check output_pPIM.isa
nanthan@nanthan-HP-240-G7-Notebook-PC:~$ cat output_pPIM.isa
PROG 10 000000 0000000000
EXE 01 000010 0000000000
EXE 01 000010 0000000000
EXE 01 000010 0000000000
EXE 01 000010 0000000000
EXE 01 000010 0000000000
EXE 01 000010 0000000000
EXE 01 000001 0000000000
EXE 01 000100 0000000000
EXE 01 000100 0000000000
EXE 01 000100 0000000000
EXE 01 000100 0000000000
EXE 01 000011 0000000000
EXE 01 000100 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000100 0000000000
EXE 01 000100 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000001 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
```

```
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000001 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000010 0000000000
EXE 01 000100 0000000000
EXE 01 000011 0000000000
EXE 01 000010 0000000000
EXE 01 000100 0000000000
EXE 01 000011 0000000000
EXE 01 000010 0000000000
EXE 01 000100 0000000000
EXE 01 000011 0000000000
EXE 01 000010 0000000000
EXE 01 000100 0000000000
EXE 01 000100 0000000000
EXE 01 000001 0000000000
EXE 01 000100 0000000000
EXE 01 000011 0000000000
EXE 01 000100 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000011 0000000000
EXE 01 000010 0000000000
EXE 01 000100 0000000000
EXE 01 000011 0000000000
EXE 01 000010 0000000000
EXE 01 000100 0000000000
END 00 000000 0000000000
```

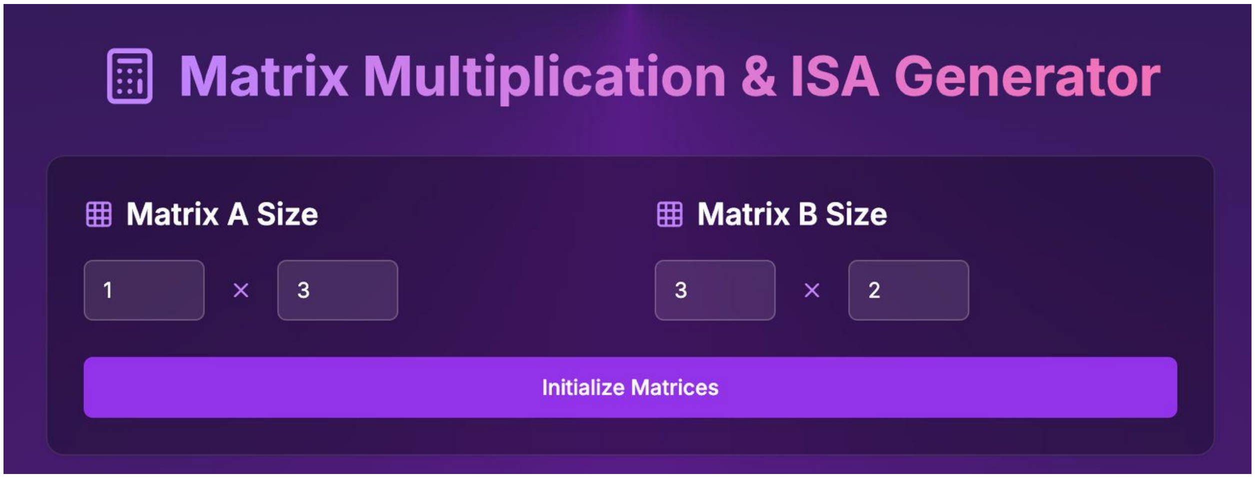
This image shows a terminal session where a Python script (translate.py) is executed to generate pPIM ISA (Instruction Set Architecture) code. The output file (output_pPIM.isa) is then displayed using the cat command. The file contains a sequence of instructions, each starting with EXE, followed by binary-encoded operation codes. The program appears to involve a repetitive execution pattern, likely related to a simple computation or testing routine for the pPIM architecture. The structured format suggests that this might be an intermediate step in compiling or simulating low-level instructions.

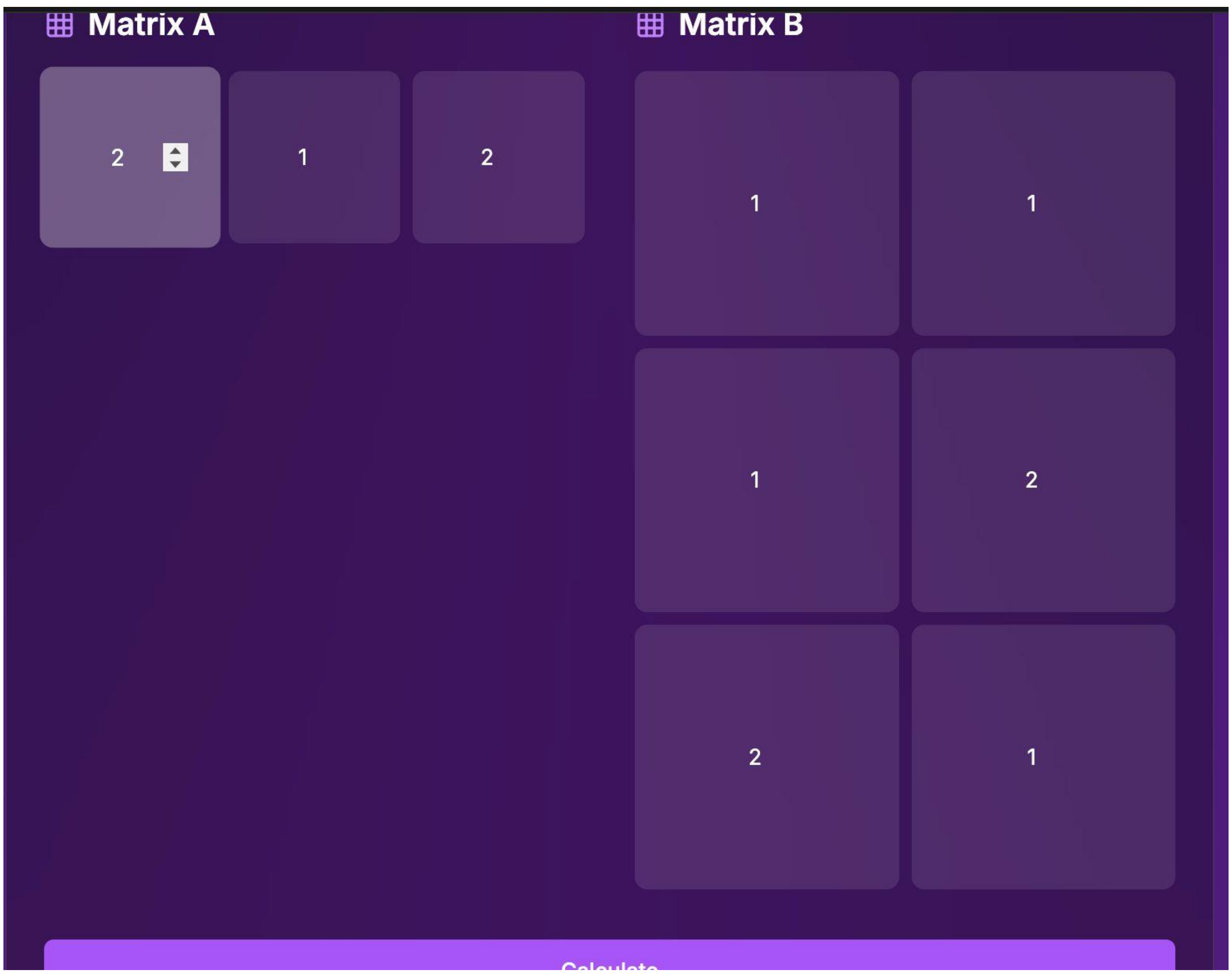
```
nanthan@nanthan-HP-240-G7-Notebook-PC:~$ nano trans.py
nanthan@nanthan-HP-240-G7-Notebook-PC:~$ python3 trans.py
Custom ISA Generated! Check output.isa
nanthan@nanthan-HP-240-G7-Notebook-PC:~$ cat output.isa
ADD R3, R4, R5
ADD R3, R4, R5
ADD R3, R4, R5
ADD R3, R4, R5
ADD R3, R4, R5
ADD R3, R4, R5
MUL R1, R2, R3
STORE R3, MEM2
STORE R3, MEM2
STORE R3, MEM2
STORE R3, MEM2
LOAD R1, MEM1
STORE R3, MEM2
LOAD R1, MEM1
LOAD R1, MEM1
LOAD R1, MEM1
LOAD R1, MEM1
STORE R3, MEM2
STORE R3, MEM2
LOAD R1, MEM1
LOAD R1, MEM1
LOAD R1, MEM1
LOAD R1, MEM1
LOAD R1, MEM1
LOAD R1, MEM1
LOAD R1, MEM1
LOAD R1, MEM1
LOAD R1, MEM1
MUL R1, R2, R3
LOAD R1, MEM1
LOAD R1, MEM1
LOAD R1, MEM1
LOAD R1, MEM1
ADD R3, R4, R5
STORE R3, MEM2
```

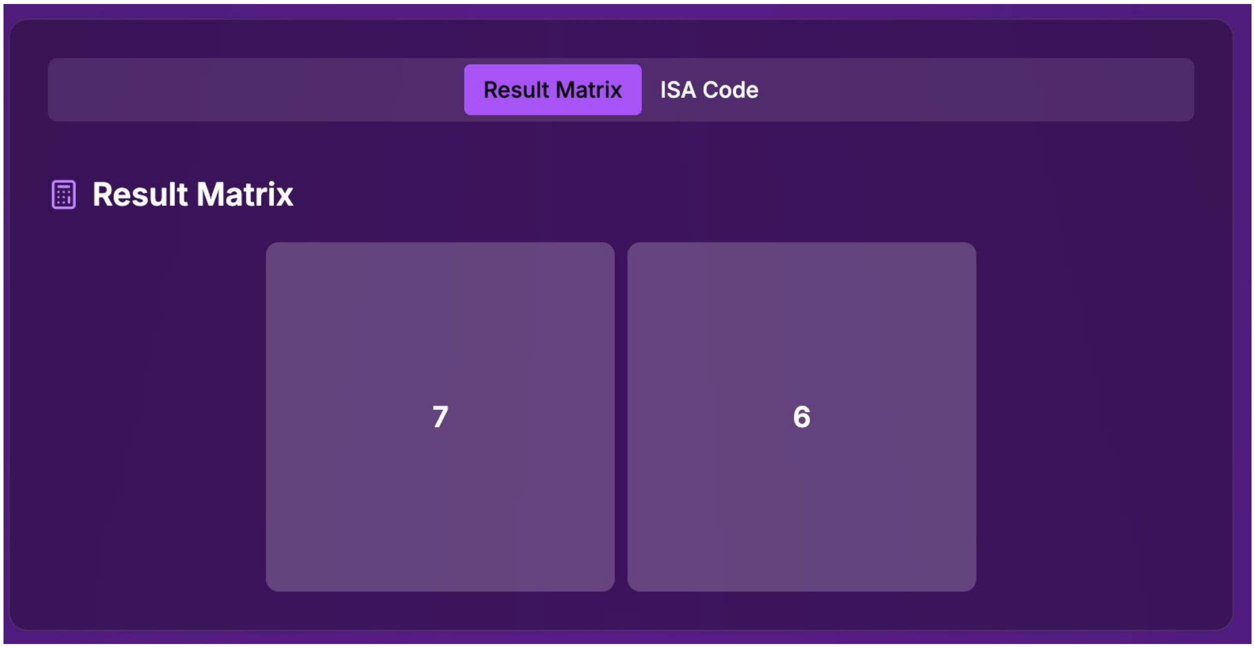
```
STORE R3, MEM2
LOAD R1, MEM1
ADD R3, R4, R5
STORE R3, MEM2
LOAD R1, MEM1
ADD R3, R4, R5
STORE R3, MEM2
STORE R3, MEM2
MUL R1, R2, R3
STORE R3, MEM2
LOAD R1, MEM1
STORE R3, MEM2
LOAD R1, MEM1
LOAD R1, MEM1
LOAD R1, MEM1
LOAD R1, MEM1
LOAD R1, MEM1
ADD R3, R4, R5
STORE R3, MEM2
LOAD R1, MEM1
ADD R3, R4, R5
```

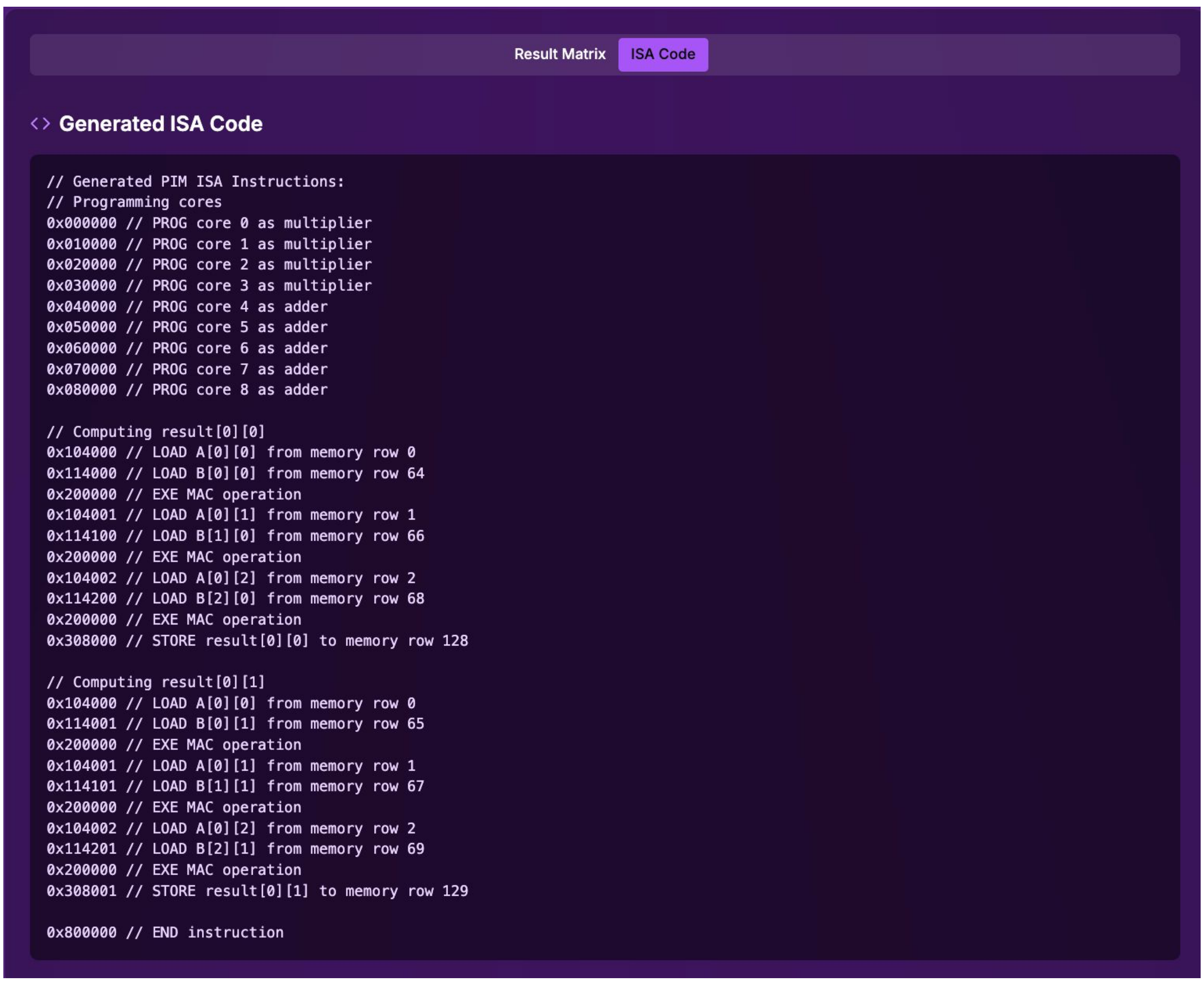
The LOAD instruction retrieves a value from a specified memory location (e.g., MEM1) and stores it into a register (e.g., R1), enabling data transfer from memory to CPU for processing. The STORE instruction does the reverse, writing the value from a register (e.g., R3) into a designated memory address (e.g., MEM2), ensuring data persistence. The ADD instruction performs arithmetic addition between two source registers (e.g., R4 and R5), storing the result in a destination register (e.g., R3). This sequence allows data movement (LOAD), computation (ADD), and storage (STORE), forming the fundamental operations of a basic program execution cycle.

FRONT-END OUTPUT SCREENSHOTS:


Matrix Multiplication & ISA Generator
Matrix A Size
1
×
3
Matrix B Size
3
×
2
Initialize Matrices


Matrix A
2
1
2
Matrix B
1
1
1
2
2
1


Result Matrix
ISA Code
Result Matrix
7
6


Result Matrix
ISA Code
Generated ISA Code
// Generated PIM ISA Instructions:
// Programming cores
0x000000 // PROG core 0 as multiplier
0x010000 // PROG core 1 as multiplier
0x020000 // PROG core 2 as multiplier
0x030000 // PROG core 3 as multiplier
0x040000 // PROG core 4 as adder
0x050000 // PROG core 5 as adder
0x060000 // PROG core 6 as adder
0x070000 // PROG core 7 as adder
0x080000 // PROG core 8 as adder
// Computing result[0][0]
0x104000 // LOAD A[0][0] from memory row 0
0x114000 // LOAD B[0][0] from memory row 64
0x200000 // EXE MAC operation
0x104001 // LOAD A[0][1] from memory row 1
0x114100 // LOAD B[1][0] from memory row 66
0x200000 // EXE MAC operation
0x104002 // LOAD A[0][2] from memory row 2
0x114200 // LOAD B[2][0] from memory row 68
0x200000 // EXE MAC operation
0x308000 // STORE result[0][0] to memory row 128
// Computing result[0][1]
0x104000 // LOAD A[0][0] from memory row 0
0x114001 // LOAD B[0][1] from memory row 65
0x200000 // EXE MAC operation
0x104001 // LOAD A[0][1] from memory row 1
0x114101 // LOAD B[1][1] from memory row 67
0x200000 // EXE MAC operation
0x104002 // LOAD A[0][2] from memory row 2
0x114201 // LOAD B[2][1] from memory row 69
0x200000 // EXE MAC operation
0x308001 // STORE result[0][1] to memory row 129
0x800000 // END instruction