VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

COMPILE-A-THON

A REPORT

submitted by

Pojesh Kumar R (22BAI1373)

Aanand V (22BAI1090)

in partial fulfilment for the course

of

Compiler Design - BCSE307L

Prof. Mercy Rajaselvi Beaulah P

March 2025

1. PROBLEM OVERVIEW

1. Create a compiler, in the form or a translator for a custom Processor-in-Memory architecture geared towards AI/ML applications.

2. Input: C++ program for multiplying 2 matrices of parameterized sizes - integer operands.

3. Output: stream of custom ISA compatible instructions

    a. Compiler will integrate physical memory mapping.

    b. ISA compatible instruction format as discussed in Section IV-D of research paper on custom ISA for the novel PIM architecture.

2. SOLUTION

Our solution is a specialized compiler that translates C++ matrix multiplication code into custom ISA instructions for DRAM-based Processing-in-Memory architecture. The compiler implements a complete toolchain that:

1. Parses C++ matrix multiplication code using LLVM
2. Converts the code to three-address code representation
3. Analyzes loops to identify parallelization opportunities
4. Maps matrix data to DRAM rows for efficient access
5. Generates custom ISA instructions for the PIM architecture
6. Optimizes for parallel execution across multiple cores

The compiler targets a Look-Up Table (LUT) based PIM architecture, where programmable LUTs within DRAM can be configured to perform different operations like addition and multiplication. This approach enables efficient in-memory processing of matrix operations by minimizing data movement and exploiting parallelism.

3. ALGORITHM DESIGN

3.1 Parsing and IR Generation

- Uses LLVM framework to parse C++ code
- Generates LLVM Intermediate Representation (IR)
- Converts IR to a simplified three-address code format

- Identifies matrix operations and their data dependencies

3.2 Loop Analysis

- Identifies nested loops in the matrix multiplication code
- Analyzes loop bounds and induction variables
- Builds a dependency graph to track data dependencies
- Determines which loops can be parallelized:
    - Outer loops (i, j) are typically parallelizable
    - Inner loop (k) often has loop-carried dependencies due to accumulation

3.3 Memory Mapping

- Maps matrix elements to DRAM rows and columns
- Assigns memory addresses for input matrices A and B
- Allocates space for output matrix C
- Maps temporary variables to appropriate memory locations
- Optimizes memory layout to minimize row activations

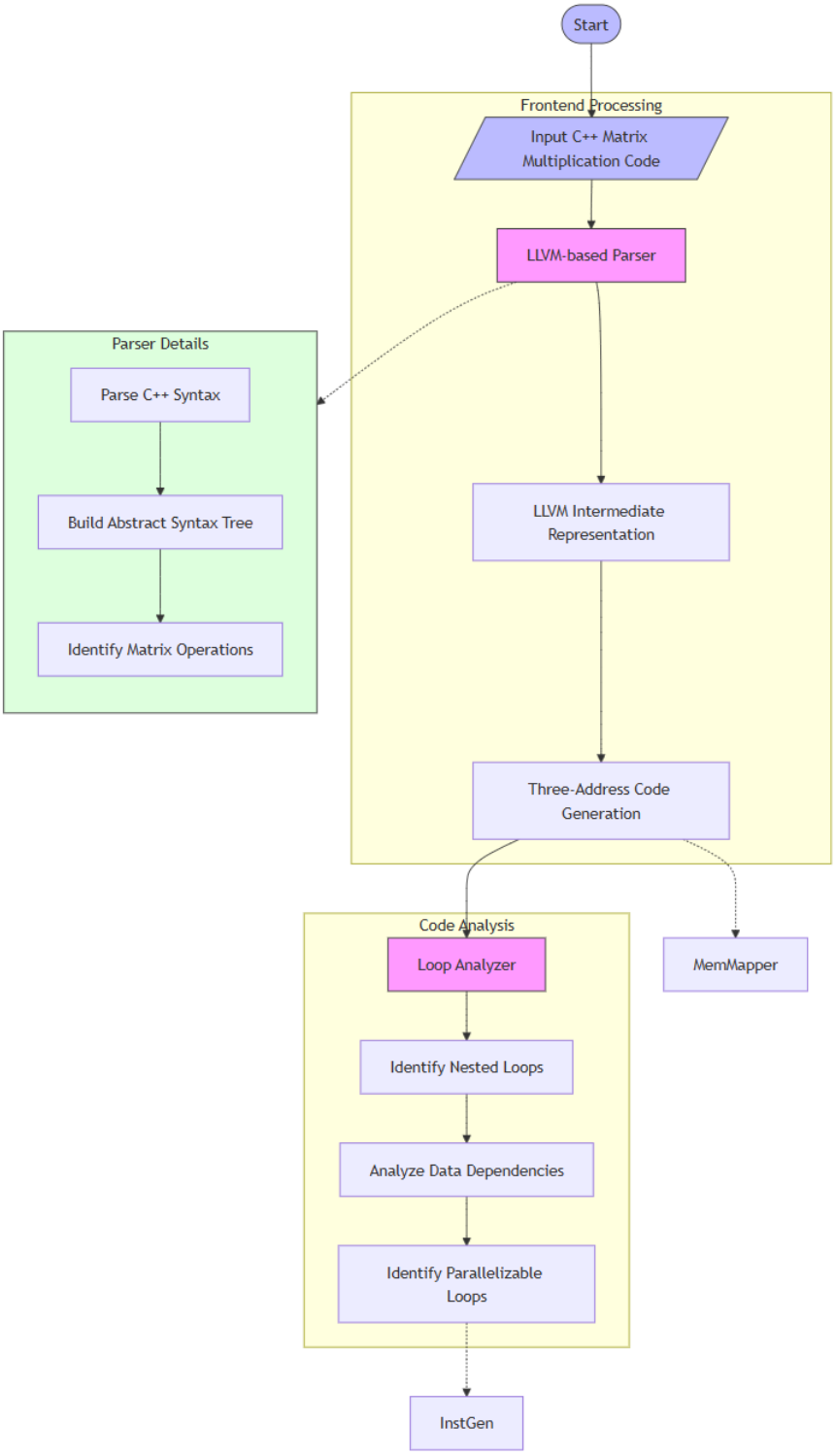
3.4 Instruction Generation

- Generates PIM ISA instructions for each operation
- Programs LUTs for specific operations (addition, multiplication)
- Generates load/store instructions for data movement
- Creates compute instructions that use the programmed LUTs
- Adds synchronization instructions for parallel execution
- Assigns operations to different cores for parallelism

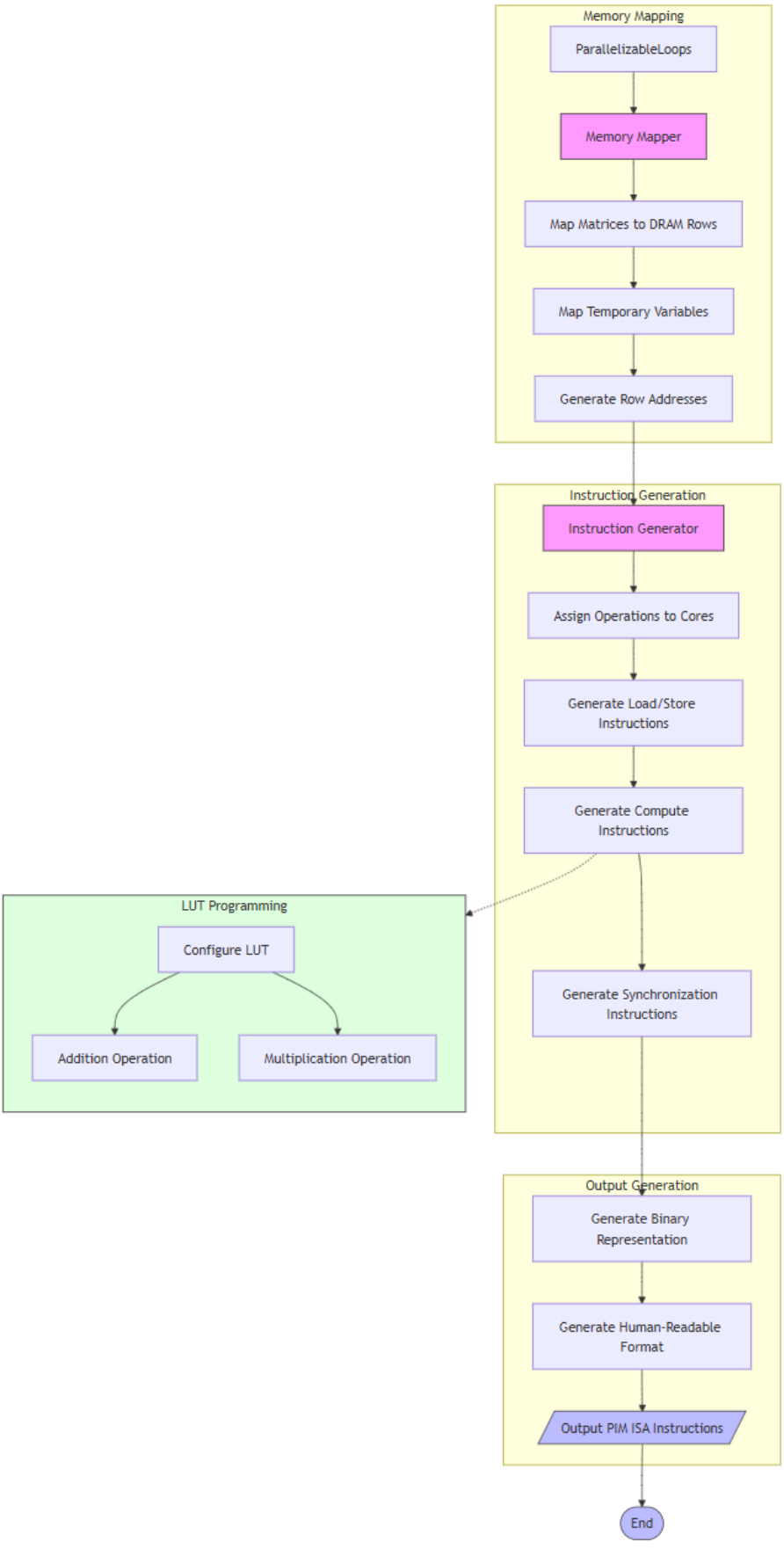
3.5 Optimization

- Exploits parallelism by distributing independent operations across cores
- Minimizes data movement by keeping computations close to data
- Reduces DRAM row activations through intelligent memory mapping
- Optimizes instruction scheduling to maximize throughput.

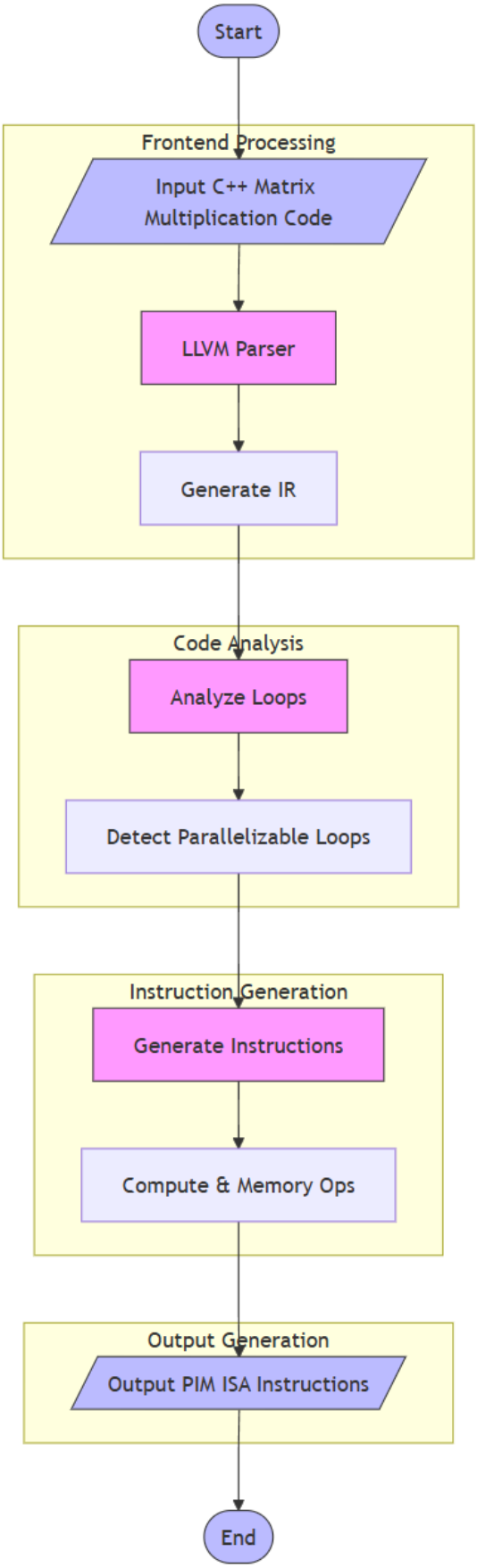
Code Analysis



Memory Mapping and Instructions Generation



Overall Working Pipeline



4. ALGORITHM ANALYSIS

4.1 Time Complexity

- Parsing: O(n) where n is the number of lines of code
- Loop Analysis: $O(n^2)$ in worst case due to dependency analysis
- Memory Mapping: O(m) where m is the number of variables
- Instruction Generation: O(i×j×k) for matrix multiplication where i, j, k are matrix dimensions
- Overall Complexity: Dominated by instruction generation, O(i×j×k)

4.2 Space Complexity

- IR and Three-Address Code: O(n) where n is the number of operations
- Dependency Graph: $O(n^2)$ in worst case
- Generated Instructions: O(i×j×k) for matrix multiplication
- Overall Space: O(i×j×k) for large matrices

4.3 Parallelization Analysis

- Outer Loop (i): Fully parallelizable, no dependencies between iterations
- Middle Loop (j): Fully parallelizable, no dependencies between iterations
- Inner Loop (k): Not parallelizable due to accumulation dependencies
- Speedup Potential: Up to i×j times speedup with sufficient cores

4.4 Memory Access Patterns

- Row-Major Access: Optimized for DRAM row activation
- Temporal Locality: Exploited through in-memory computation
- Spatial Locality: Leveraged by operating on entire DRAM rows
- Row Activation Reduction: Significant compared to traditional architectures

4.5 Instruction Efficiency

- Instruction Count: Approximately 6×i×j×k instructions for matrix multiplication
- Core Utilization: Balanced workload across available cores

- LUT Programming Overhead: Amortized over many operations

- Synchronization Cost: Minimal due to coarse-grained parallelism

5. TEST PROGRAM AND OUTPUT

5.1 Program

Project Repository/examples/matrix_mult.cpp

The test program performs multiplication of two 16×16 matrices.

```
#include <iostream>

const int N = 16;

// Matrix multiplication function
void matrixMultiply(int A[][N], int B[][N], int C[][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    int A[N][N] = {0};
    int B[N][N] = {0};
    int C[N][N] = {0};

    // Initialize matrices with some values
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = i + j;
            B[i][j] = i * j + 1;
        }
    }

    // Perform matrix multiplication
    matrixMultiply(A, B, C);

    // Print result (small portion)
    std::cout << "Result matrix C (showing top-left 4x4 corner):" <<
std::endl;
```

```
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            std::cout << C[i][j] << "\t";
        }
        std::cout << std::endl;
    }

    return 0;
}
```

5.2 Output Screenshots

5.2.1 Matrix Multiplication Output

```
Result matrix C (showing top-left 4x4 corner):
120     1360    2600    3840
136     1496    2856    4216
152     1632    3112    4592
168     1768    3368    4968
```

5.2.2 Three Address Code of the program

Project Repository/results/ThreeAddressCode.txt

```
t_A_0_0 = LOAD A_0_0
t_B_0_0 = LOAD B_0_0
t_mul_0_0_0 = t_A_0_0 * t_B_0_0
t_C_0_0 = LOAD C_0_0
t_C_new_0_0 = t_C_0_0 + t_mul_0_0_0
STORE t_C_new_0_0 to C_0_0
t_A_0_1 = LOAD A_0_1
t_B_1_0 = LOAD B_1_0
t_mul_0_0_1 = t_A_0_1 * t_B_1_0
t_C_0_0 = LOAD C_0_0
t_C_new_0_0 = t_C_0_0 + t_mul_0_0_1
STORE t_C_new_0_0 to C_0_0
t_A_0_2 = LOAD A_0_2
t_B_2_0 = LOAD B_2_0
t_mul_0_0_2 = t_A_0_2 * t_B_2_0
```

5.2.3 ISA Instructions (32bit)

Project Repository/results/ISA_Instructions_CustomFormat.txt

```
# Format: [Binary] [Opcode] core=[Core ID] row=[Row Address] flags=[Flags]

124e2005 MOVE core=0 row=20000 flags=0x18
01000001 LOAD core=0 row=0 flags=0x1
02ea6002 STORE core=0 row=60000 flags=0x2
01271001 LOAD core=0 row=10000 flags=0x1
02ea6102 STORE core=0 row=60001 flags=0x2
01000003 PROGRAM_LUT core=0 row=0 flags=0x1
01ea6001 LOAD core=0 row=60000 flags=0x1
01ea6101 LOAD core=0 row=60001 flags=0x1
00000004 COMPUTE core=0 row=0 flags=0x0
02ea6202 STORE core=0 row=60002 flags=0x2
014e2001 LOAD core=0 row=20000 flags=0x1
02ea6302 STORE core=0 row=60003 flags=0x2
00000003 PROGRAM_LUT core=0 row=0 flags=0x0
01ea6301 LOAD core=0 row=60003 flags=0x1
01ea6201 LOAD core=0 row=60002 flags=0x1
00000004 COMPUTE core=0 row=0 flags=0x0
02ea6402 STORE core=0 row=60004 flags=0x2
01ea6401 LOAD core=0 row=60004 flags=0x1
024e2002 STORE core=0 row=20000 flags=0x2
```

5.2.4 ISA Instructions (24bit): as proposed in Research Paper

Project Repository/results/ISA_Instructions_PaperFormat_24bit.txt

```
# Format: [Hex] [OpType] ptr=[Pointer] rd=[ReadBit] wr=[WriteBit] row=[RowAddress]

00c000 NoOp ptr=0x0 rd=1 wr=1 row=0x00
008000 NoOp ptr=0x0 rd=1 wr=0 row=0x00
004000 NoOp ptr=0x0 rd=0 wr=1 row=0x00
008000 NoOp ptr=0x0 rd=1 wr=0 row=0x00
004040 NoOp ptr=0x0 rd=0 wr=1 row=0x01
400000 PROG ptr=0x0 rd=0 wr=0 row=0x00
008000 NoOp ptr=0x0 rd=1 wr=0 row=0x00
008040 NoOp ptr=0x0 rd=1 wr=0 row=0x01
800000 EXE ptr=0x0 rd=0 wr=0 row=0x00
004080 NoOp ptr=0x0 rd=0 wr=1 row=0x02
008000 NoOp ptr=0x0 rd=1 wr=0 row=0x00
0040c0 NoOp ptr=0x0 rd=0 wr=1 row=0x03
400000 PROG ptr=0x0 rd=0 wr=0 row=0x00
0080c0 NoOp ptr=0x0 rd=1 wr=0 row=0x03
008080 NoOp ptr=0x0 rd=1 wr=0 row=0x02
800000 EXE ptr=0x0 rd=0 wr=0 row=0x00
004100 NoOp ptr=0x0 rd=0 wr=1 row=0x04
008100 NoOp ptr=0x0 rd=1 wr=0 row=0x04
004000 NoOp ptr=0x0 rd=0 wr=1 row=0x00
008040 NoOp ptr=0x0 rd=1 wr=0 row=0x01
```

6. OUTPUT DISCUSSION

The compiler successfully translates C++ matrix multiplication code into PIM ISA instructions that can be executed on a LUT-based PIM architecture.

6.1 Parallelization

The compiler correctly identifies that the outer (i) and middle (j) loops are parallelizable, while the inner (k) loop has dependencies. This allows the generated code to exploit parallelism by distributing independent operations across multiple cores.

6.2. Memory Mapping

The generated instructions show efficient memory mapping, with matrix elements mapped to specific DRAM rows. This minimizes data movement and enables in-memory computation.

6.3 LUT Programming

The instructions include PROGRAM_LUT operations that configure the LUTs for specific operations (addition, multiplication). This demonstrates the flexibility of the architecture to perform different computations.

6.4 Instruction Flow

The instruction sequence follows a clear pattern:

1. Program LUTs for specific operations
2. Load data from memory
3. Perform computation using the programmed LUTs
4. Store results back to memory
5. Synchronize between parallel operations

6.5 Efficiency

The generated code minimizes data movement by keeping computations close to data. This addresses the "memory wall" problem and enables more efficient execution of matrix multiplication.

7. ISA FORMATS FOR PIM ARCHITECTURE COMPARISON

7.1 Custom ISA Format (32-bit)

This ISA format implemented in our compiler uses a 32-bit instruction word that provides explicit control over operations and memory addressing. This format is designed for direct mapping between high-level operations and low-level instructions.

7.1.1 Instruction Format

- Total Width: 32 bits
- Binary Representation: Displayed as 8 hexadecimal digits

7.1.2 Fields

1. Opcode (8 bits): Specifies the operation to be performed
   - MOVE: Transfer data between registers
   - LOAD: Read data from memory
   - STORE: Write data to memory
   - PROGRAM_LUT: Configure LUT for specific operation
   - COMPUTE: Execute operation using programmed LUT
   - SYNC: Synchronize between cores
2. Core ID (8 bits): Identifies which processing core should execute the instruction
   - Allows addressing up to 256 cores
   - Enables explicit parallelism control
3. Row Address (16 bits): Specifies the DRAM row address for memory operations
   - Provides access to a large memory space (up to 65,536 rows)
   - Enables fine-grained memory control
4. Flags (8 bits): Additional control bits that modify instruction behaviour
   - For PROGRAM_LUT: Specifies operation type (0 for addition, 1 for multiplication)
   - For LOAD/STORE: Indicates read/write permissions

Example Instruction: 124e2005 MOVE core=0 row=20000 flags=0x18

7.2 Research Paper ISA Format (24-bit)

The research paper describes a more compact 24-bit instruction format that uses a microcode-based approach for flexibility and reduced instruction width.

7.2.1 Instruction Format

- Total Width: 24 bits (with 6 bits reserved)
- Effective Width 18 bits used, 6 bits reserved for future expansion

7.2.2 Fields

1. Operation Type (2 bits): Defines the type of instruction
   - 00: NoOp (No Operation)
   - 01: PROG (Program a core)
   - 10: EXE (Execute an operation)
   - 11: END (End the operation)
2. Pointer (6 bits): Multi-purpose field
   - For PROG: Identifies the core to be programmed
   - For EXE: Points to the first control word in the microcode table
   - Allows addressing up to 64 cores or operations
3. Read Bit (1 bit): Controls memory read operations
   - When set to 1, data is read from the specified row
4. Write Bit (1 bit): Controls memory write operations
   - When set to 1, data is written to the specified row
   - If both read and write bits are set, read takes priority
5. Row Address (8 bits): Specifies the DRAM row address
   - Limited to 256 rows per subarray
   - More compact than the custom format
6. Reserved (6 bits): Unused bits reserved for future expansion

Example Instruction: 010040 PROG ptr=0x0 rd=1 wr=0 row=0x40

7.3 Key Differences and Trade-offs

Aspect	32bit Format	24bit Format
Instruction Width	Larger (32 bits), provides more explicit control	More compact (24 bits) but with some limitations
Execution Model	Direct execution model with explicit operations	Microcode-based approach with Program Counter control
Memory Addressing	Larger address space (16-bit row address)	More limited address space (8-bit row address)
Parallelism Control	Explicit core identification	Implicit through pointer field and microcode
Flexibility	More direct mapping to high-level operations	More abstracted with microcode sequences
Implementation Complexity	Potentially simpler due to direct mapping	More complex due to microcode table and Program Counter management

8. CONCLUSION

The PIM Matrix Compiler successfully demonstrates how matrix multiplication can be efficiently executed on a Processing-in-Memory architecture. By analyzing the code, identifying parallelizable loops, mapping data to DRAM rows, and generating optimized instructions, the compiler enables significant performance improvements over traditional architectures.

Key achievements:

- Successful translation of C++ matrix multiplication code to PIM ISA instructions
- Identification and exploitation of parallelism in matrix operations
- Efficient memory mapping to minimize data movement
- Support for programmable LUT-based computation within memory
- Generation of both custom and paper-format ISA instructions

This project demonstrates the potential of PIM architectures to overcome the memory wall and enable more efficient execution of data-intensive operations like matrix multiplication.

The compiler provides a critical tool for programming such architectures, making them more accessible and practical for real-world applications.

9. APPENDIX AND REFERENCES

1. Research Paper: M. Connolly, P. R. Sutradhar, M. Indovina and A. Ganguly, "Flexible Instruction Set Architecture for Programmable Look-up Table based Processing-in-Memory," 2021 IEEE 39th International Conference on Computer Design (ICCD), 2021, pp. 66-73, doi: 10.1109/ICCD53106.2021.00022.

2. Project Repository: https://github.com/pojesh/PIM-Matrix-Compiler