COMPILER DESIGN

DIGITAL ASSIGNEMENT – 3

TEAM MEMBERS:

T HARISH – 22BRS1048

SIDHARTH P J – 22BRS1086

FACULTY UNDER GUIDANCE : MANJU G

COURSE CODE: BCSE307L

SLOT: B1+TB1

Introduction

This report covers the process of taking a matrix multiplication code, converting it into LLVM IR and a custom ISA, identifying parallel loops for parallel execution, and creating a lookup table for the custom ISA instructions. The goal is to demonstrate how compiler design principles can be applied to optimize and execute code on custom architectures.

Matrix Multiplication Code

The starting point is a simple matrix multiplication code written in C++:

```cpp
#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);

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

    return 0;
}
```

Conversion to LLVM IR

To convert this code into LLVM IR, you would typically use the clang compiler with the -emit-llvm option. However, for simplicity, we'll focus on the conceptual steps rather than the actual LLVM IR output.

1. **Compilation**: The C++ code is compiled into LLVM IR using clang -emit-llvm -S matrix.cpp.
2. **LLVM IR Output**: The resulting LLVM IR file (matrix.ll) would contain the intermediate representation of the code, which is platform-independent.

Conversion to Custom ISA

Next, we convert the LLVM IR into a custom ISA. This involves defining the custom ISA's instruction set and translating the LLVM IR into these instructions.

Custom ISA Instructions

For demonstration purposes, let's define a simplified custom ISA with the following instructions:

- **LOAD**: Load a value into a register.
- **STORE**: Store a value in memory.
- **MUL**: Multiply two values and store in a register.
- **ADD**: Add two values and store in memory or a register.
- **JMP**: Unconditional jump to a label.
- **CMP**: Compare a register with a value.
- **JGE**: Conditional jump if greater or equal.
- **HALT**: Stop execution.

Three-Address Code

Before converting to the custom ISA, we represent the code in three-address code. This step simplifies the translation by breaking down complex operations into basic ones.

For the matrix multiplication, the three-address code might look like this:

```
i = 0
L1: if i >= 2 goto L2
j = 0
L3: if j >= 2 goto L4
C[i][j] = 0
k = 0
L5: if k >= 2 goto L6
C[i][j] = C[i][j] + A[i][k] * B[k][j]
k = k + 1
goto L5
L6: j = j + 1
goto L3
L4: i = i + 1
goto L1
```

```
L2: end
```

Custom ISA Code

Now, we translate the three-address code into the custom ISA:

```
LOAD R1, 0         // i = 0
L1: CMP R1, 2      // if i >= 2
JGE L2             // goto L2
LOAD R2, 0         // j = 0
L3: CMP R2, 2      // if j >= 2
JGE L4             // goto L4
STORE 0, C[R1][R2] // C[i][j] = 0
LOAD R3, 0         // k = 0
L5: CMP R3, 2      // if k >= 2
JGE L6             // goto L6
MUL R4, A[R1][R3], B[R3][R2]
ADD C[R1][R2], C[R1][R2], R4
ADD R3, R3, 1      // k = k + 1
JMP L5             // goto L5
L6: ADD R2, R2, 1  // j = j + 1
JMP L3             // goto L3
L4: ADD R1, R1, 1  // i = i + 1
JMP L1             // goto L1
L2: HALT           // end
```

Parallelization

To parallelize the matrix multiplication, we focus on the outer loops (for i and for j). These loops are independent and can be executed concurrently using multiple processing elements.

Parallel Loop Identification

The parallel loops are identified as follows:

```
for (int i = 0; i < 2; i++) {

    for (int j = 0; j < 2; j++) {

        // Inner loop operations

    }

}
```

Parallel Execution

To execute these loops in parallel, each iteration of the outer loop can be assigned to a different processing element. For example, using OpenMP, you could parallelize the loops like this:

```
#pragma omp parallel for

for (int i = 0; i < 2; i++) {

    #pragma omp parallel for

    for (int j = 0; j < 2; j++) {

        // Inner loop operations

    }

}
```

However, due to the dependency in the inner loop, only the outermost loop should be parallelized for simplicity.

Lookup Table

Finally, we create a lookup table for the custom ISA instructions. This table maps each instruction to its operation, operands, and description.

Lookup Table Implementation

Here's a Python script that generates the lookup table:

```
class Instruction:

    def __init__(self, opcode, operation, operands, description):

        self.opcode = opcode

        self.operation = operation
        self.operands = operands

        self.description = description


class LookupTable:

    def __init__(self):

        self.table = []


    def add_instruction(self, opcode, operation, operands, description):

        instruction = Instruction(opcode, operation, operands, description)

        self.table.append(instruction)


    def print_lookup_table(self):

        print("Lookup Table:")

        print(f"{'Opcode':^8} | {'Operation':^15} | {'Operands':^20} |
{'Description':^30}")

        print("-" * 80)

        for instruction in self.table:

            print(f"{instruction.opcode:^8} | {instruction.operation:^15} |
{instruction.operands:^20} | {instruction.description:^30}")


def main():

    lookup_table = LookupTable()


    # Custom ISA Instructions

    lookup_table.add_instruction("LOAD", "Load immediate", "R1, 0", "Load a value into a
register")

    lookup_table.add_instruction("STORE", "Store value", "0, C[R1][R2]", "Store a value
in memory")

    lookup_table.add_instruction("MUL", "Multiply", "R4, A[R1][R3], B[R3][R2]",
"Multiply two values and store in a register")
```

```
    lookup_table.add_instruction("ADD", "Add", "C[R1][R2], C[R1][R2], R4", "Add two values and store in memory")

    lookup_table.add_instruction("ADD", "Add", "R3, R3, 1", "Increment a register")

    lookup_table.add_instruction("JMP", "Jump", "L1", "Unconditional jump to a label")

    lookup_table.add_instruction("CMP", "Compare", "R1, 2", "Compare a register with a value")

    lookup_table.add_instruction("JGE", "Jump if greater or equal", "L2", "Conditional jump if greater or equal")

    lookup_table.add_instruction("HALT", "Halt", "", "Stop execution")


    lookup_table.print_lookup_table()


if __name__ == "__main__":

    main()
```

This lookup table can be used in compiler design to map instructions to their respective operations and operands, facilitating the translation of high-level code into machine code for the custom ISA.

This report covers the entire process from matrix multiplication code to custom ISA conversion, parallelization, and lookup table creation. It demonstrates how compiler design principles can be applied to optimize and execute code on custom architectures.