BARZ Compiler: An Overview of Memory Allocation/Management and Code Generation

Ismael Talib Ridha Barzani
Student ID: 40188139
Department of Electrical and Computer Engineering
Concordia University - Gina Cody School
Montreal, Quebec, Canada
ismaelmergasori@gmail.com

***Abstract*—In this final iteration, I present the design and implementation of a code generator for the BARZ compiler targeting the MOON virtual processor architecture. My approach employs a dual-phase visiting strategy where memory requirements are first analyzed, followed by the actual code generation phase. I update a symbol-table entry taken from symbol-table generation phase in which it calculates memory sizes and offsets for all program components while handling various data types including integers, floats, arrays, and objects. The implementation addresses key challenges including register allocation management, function call handling with parameter passing and return values, and memory offset calculations for complex data structures. Experimental results show successful implementation of core language features including arithmetic expressions, control flow statements, I/O operations, and function calls.**

***Index Terms*—Compiler Design, Memory Management, Code Generation**

I. Introduction

Compilers serve as the critical bridge between high-level programming languages and machine code through a series of complex transformations. In this paper, I focus on implementing the final phase of compilation: code generation. My work implements a code generator for the BARZ compiler targeting the MOON virtual processor architecture, completing the compiler pipeline that includes lexical analysis, parsing, abstract syntax tree (AST) construction, and semantic analysis.

The code generation phase translates the semantically validated AST into executable MOON assembly code while managing computational resources and memory layout. This transformation presented several challenges unique to the MOON architecture, including its register-based computation model with limited registers (r1-r15), stack-based memory management, and lack of native support for floating-point operations.

My implementation follows a dual-phase approach. First, I created a Memory Size Visitor, `MemSizeVisitor`, that traverses the AST to calculate memory requirements for all program components, enriching a deep-copied symbol table with size and offset information. This phase handles memory allocation for integers, floats, arrays, objects, and temporary variables. Second, I developed a Code Generation Visitor, `CodeGenVisitor`, that produces MOON assembly instructions by implementing a systematic translation strategy for various language constructs.

Throughout my implementation, I addressed several technical challenges inherent to code generation:

- Register allocation through a pool-based strategy that efficiently manages the given register set
- Stack frame management for function calls with parameter passing and return value handling
- Offset calculation for complex data structures including arrays and object members
- Control flow implementation for conditional statements and loops
- Arithmetic and logical expression evaluation with proper temporary variable generation

I implemented a comprehensive feature set progressing from fundamental operations (variable declarations, assignments, basic I/O) to complex language constructs (expressions, control structures, function calls). My current implementation has limitations around object-oriented features, notably member function calls and complex member access operations. Additionally, array element access and expressions with array indices or object members remain as areas for future work. The remainder of this paper details my implementation approach, discusses challenges encountered, and evaluates results against the target architecture constraints.

II. Implemented Code Generation Items

In this section, I provide an overview of the code generation features implemented in my compiler. I categorize these implementations into five main areas: memory allocation, function operations, program statements, data access, and expression computation.

This is submitted as part of COMP-6421 Assignment #5.

A. Memory Allocation

✓ **1.1. Basic Types** - Successfully implemented memory allocation for both integer and float data types. Integer values are allocated 4 bytes while float values receive 8 bytes to accommodate their precision requirements.

✓ **1.2. Array Allocation** - Implemented allocation for single and multi-dimensional arrays of basic types. Memory is allocated contiguously with dimensions calculated at compile time. Size calculations respect element size differences (4 bytes for int, 8 bytes for float).

✓ **1.3. Object Allocation** - Implemented memory allocation for class instances, accounting for all member variables and their alignment requirements. Object fields are laid out sequentially in memory with proper padding.

✓ **1.4. Composite Structure Allocation** - Successfully handled arrays of objects, calculating total memory requirements as the product of object size and array dimensions.

B. Function Operations

✓ **2.1. Function Execution Flow** - Implemented branching to function code blocks and returning to calling functions using proper stack frame management and link register handling.

✓ **2.2. Parameter Passing** - Successfully implemented parameter passing to functions with values stored in the callee's stack frame.

✓ **2.3. Return Value Handling** - Implemented return value passing from functions back to calling contexts through a dedicated stack location.

✗ **2.4. Member Function Calls** - Not implemented: The current implementation lacks support for member function calls within class contexts. This would require resolving the 'this' pointer and handling class-scoped variable access.

C. Program Statements

✓ **3.1. Assignment Statements** - Fully implemented assignment operations with arbitrary expressions on the right-hand side. The implementation correctly evaluates expressions and stores results in target variables.

✓ **3.2. Conditional Statements** - Implemented if-then-else control flow with proper branching mechanics and condition evaluation.

✓ **3.3. Loop Statements** - Successfully implemented while loops with condition evaluation and branching mechanisms.

✓ **3.4. I/O Operations** - Implemented read and write statements using MOON machine's keyboard input and console output facilities.

D. Data Access Operations

✗ **4.1. Array Element Access** - Not implemented

✗ **4.2. Array Object Member Access** - Not implemented

✗ **4.3. Object Member Access** - Not implemented

✗ **4.4. Complex Object Access** - Not implemented

E. Expression Computation

✓ **5.1.1. Arithmetic Operations** - Implemented support for basic arithmetic operations: addition, subtraction, multiplication, and division.

✓ **5.1.2. Logical Operations** - Successfully implemented logical operations: 'and' and 'or' operators with proper boolean evaluation.

✗ **5.2. Array Factor Expressions** - Not implemented

✗ **5.3. Member Access Expressions** - Not implemented

The implementation strategy followed a progressive approach, beginning with fundamental memory allocation and variables, then basic control structures, followed by function handling and finally expressions.

III. Implementation and Design Details

A. Overall Architecture

The code generator was implemented using a two-phase visitor pattern approach. First, a `MemSizeVisitor` calculates memory requirements by traversing the AST and enriching the symbol table with size and offset information. Then, a `CodeGenVisitor` generates MOON assembly instructions by visiting the AST again with memory information available.

B. Memory Allocation Strategy

I implemented a stack-based memory allocation strategy with careful attention to alignment requirements:

- 4-byte alignment for integers and pointers
- 8-byte alignment for floating-point values
- Contiguous allocation for arrays with proper dimension calculations
- Function-specific stack frames with dedicated areas for parameters, local variables, and return values

The memory layout follows a convention where:

- Return value is stored at offset `0(r14)`
- Return address is stored at offset `-[return_size+4](r14)`
- Parameters start at offsets below these reserved areas
- Local and temporary variables are allocated in declaration order below parameters

C. Register Allocation

I implemented a pool-based register allocation strategy:

- Register r0 is reserved for constant 0
- Registers r1-r12 form the allocation pool for general computations
- Register r13 is used for function return values
- Register r14 serves as the frame pointer
- Register r15 serves as the link register for function calls

Registers are allocated on demand and freed as soon as their values are no longer needed, maximizing register reuse within expressions.

D. Code Organization

The generated code is organized into two main sections:

- Data section: Contains buffer space for I/O operations
- Code section: Contains function definitions and executable instructions

Each function follows a standard signature-body-end structure, ensuring proper stack frame management and return address handling.

E. Expression Evaluation

For expression evaluation, I implemented a bottom-up approach where:

- Leaf nodes (literals, variables) compute their values first
- Intermediate operation nodes process their operands before combining results
- Results are stored in temporary variables with metadata attached to AST nodes
- Parent nodes retrieve results from child nodes' metadata

This approach allows for complex expression trees while minimizing register pressure and maintaining consistent state between node visits.

IV. Tools

For implementing the code generator, I utilized several key tools throughout the development process:

- **C++ Programming Language**: The entire compiler was implemented in C++17, making extensive use of smart pointers and standard library containers.
- **MOON Virtual Machine**: The target architecture for code generation is the MOON virtual processor developed by Dr. Grogono. This simulator provides a simple assembly language with register-based computation.

V. Conclusion

In this paper, I presented a code generator for the BARZ compiler targeting the MOON virtual processor architecture, implementing a two-phase approach with memory analysis followed by code generation. The implementation successfully handles memory allocation for various data types, function calls with parameters and return values, control flow statements, and arithmetic/logical expressions. While fundamental language features are fully operational, the current implementation has limitations in object-oriented programming features and array access operations which represent opportunities for future work. Through this project, I gained practical experience in compiler construction techniques, particularly in navigating the complexities of memory layout and register allocation within the constraints of a target architecture.