WACC Compiler Project Report

Ahmad Jamsari, Andrey Kravtsov, Mikhail Bazarsadaev

March 16, 2025

1 The Final Product

Our compiler fully satisfies the WACC language functional specification, demonstrating correct implementation of lexical and semantic analysis, assembly code generation, and optimization across our comprehensive test suite. Our codebase is very modular being split into multiple crates and IR's. However, our testing infrastructure presents opportunities for improvement. Currently, the test suite is tightly coupled to our main function, creating maintenance challenges when interface changes occur. A more decoupled approach would involve directly executing our actual compiler binary. Additionally, implementing supplementary utility functions within our test suite would facilitate more efficient testing and directory-based compilation control. The Rust compilation times for our compiler could also be optimized to support more rapid iterative development.

Regarding performance metrics, despite the extended compilation time for our compiler itself, its execution speed for WACC programs is exceptional. When processing a substantial 3,000-line `printAllTypes.wacc` file with all optimizations enabled, our compiler completes execution in approximately 1.5 seconds, while the reference compiler fails to compile, timing out after 10 seconds. Even with a more modest 300-line `printAllTypes.wacc` file, our implementation demonstrates superior efficiency—completing in 0.03 seconds compared to the reference compiler's 7 seconds, representing a performance improvement exceeding 200x.

2 Project Management

2.1 Analysis of Group Organization and Project Management Approach

- **Group Structure and Organization:** Our group adopted a collaborative yet independent structure. Tasks were divided among members, with each person working on their respective branches individually. Periodically, we merged our work to ensure integration and consistency. Towards the end of the project, all members contributed to debugging and refining the final product. For the back-end, Ahmad took the lead in designing the main structure of the middle IR and back-end IR, which served as a foundation for others to build upon.

- **Coordination Techniques and Work Distribution:** We relied on a combination of tools and platforms for coordination. Git was instrumental in enabling us to collaborate effectively, as it allowed us to work independently while still reviewing and contributing to each other's branches. Communication was facilitated through a WhatsApp group chat and a Discord server, where we used voice chats during merging sessions and for general discussions. The independence of our tasks minimized the occurrence of complex merge conflicts.

- **Tools That Helped or Hindered Progress:** Git proved to be a highly effective tool for our workflow. It streamlined collaboration, allowing us to assist each other with code while maintaining individual progress. We did not encounter any significant issues with Git throughout the project. Additionally, the use of WhatsApp and Discord for communication further enhanced our coordination and ensured smooth progress.

- **What Went Well:** Our process was efficient due to the independence of tasks and the clear division of responsibilities. The collaborative debugging phase towards the end of the project was particularly effective in identifying and resolving issues. The foundational work on the IR by Ahmad provided a strong starting point for the rest of the team, which facilitated smoother development.
- **What We Would Do Differently:** In hindsight, we believe that involving multiple team members in drafting the IR design earlier in the process would have been beneficial. This approach would have allowed for broader input and consensus on design decisions, rather than relying on a single individual to make these choices initially.

3 Design Choices and Implementation Details

In this section, we analyze the key design choices made during the development of the WACC compiler. These choices include the selection of implementation tools, the challenges encountered during implementation, and the design patterns adopted, along with their rationale.

3.1 Implementation Language and Tool Choices

We decided to use the Rust programming language for the WACC compiler due to its rich library ecosystem for compiler development, its clean and expressive syntax, strong type system, memory safety guarantees, and exceptional performance. For parsing, we used the ‘Chumsky‘ parser combinator crate in Rust, which offers an intuitive API and excellent support for error pretty-printing.

3.2 Challenges Encountered

During implementation, one of the primary challenges we faced was the verbose and cryptic error messages emitted by ‘Chumsky‘ due to generic type mismatches, which were often difficult to debug. Additionally, we encountered a bug in ‘Chumsky‘ that generated incorrect error spans, complicating error handling further.

Another recurring issue stemmed from Rust’s borrow checker, particularly while constructing the control flow graph (CFG) and interference graph. While the borrow checker ensures memory safety, its strict constraints require careful design and refactoring to resolve issues related to ownership and lifetimes. Sometimes we’d have to mark the objects for deletion first before deleting them like a garbage collector.

3.3 Design Patterns

Throughout the development process, we relied on several design patterns to structure and simplify our implementation. One notable pattern was the *Folder Pattern*, a variation of the Visitor Pattern, which we used extensively to transform intermediate representations (IRs) during the lowering process. For example, this pattern was critical when lowering from one IR to another.

Another pattern we employed was the *Strategy Pattern*, which allowed us to share similar logic across optimization stages. Specifically, we applied this pattern to replace operands in our assembly instructions and to reuse a common iterative function for backward dataflow analyses, such as liveness analysis, in both the middle-end and back-end.

3.4 Structure of the Compiler

The architecture of our compiler is organized into three main components: the front-end, middle-end, and back-end. Below, we describe each component in detail.

3.4.1 Front-End

The front-end begins with a lexing phase, which tokenizes the input file. These tokens are then streamed to the parser, which constructs the *Syntax Abstract Syntax Tree* (AST). After parsing, we perform a renaming pass that transforms the Syntax AST into a *Higher IR* (HIR) AST. During this renaming phase, different loop types (e.g., `while`, `do-while`) are normalized into a common representation.

Next, type checking is performed on the HIR, resulting in a *Typed Higher IR* (THIR). Errors encountered during this process are reported to the user. Finally, the THIR is lowered into our custom intermediate representation, called *WackyIR*. This IR is a Three-Address Code (TAC) representation that is linear, simple, and well-suited for optimization. Unlike higher-level IRs, WackyIR abstracts away scopes and loops, instead offering a structure closer to assembly, with the added convenience of creating temporaries as needed.

3.4.2 Middle-End

The middle-end operates on the WackyIR, which can optionally be converted into a Control Flow Graph (CFG). Optimization passes are performed iteratively on the CFG until a fixed point is reached. This iterative approach mitigates the *phase ordering problem*, as each pass guarantees progress toward the fixed point. Once optimizations are complete, the CFG is converted back into WackyIR.

3.4.3 Back-End

The back-end lowers WackyIR into *AssemblyIR*, our custom representation of x86-64 assembly. We chose to target x86-64 due to its native compatibility with our development environments and our familiarity with the architecture. We have the notion of pseudoregister which acts as an infinite storage place.

The back-end includes a register allocation pass, which maps pseudo-registers to physical registers or stack locations. If register allocation is disabled, all pseudo-registers default to stack locations. After allocation, a *pseudo-register replacement* pass ensures that any remaining pseudo-registers are assigned to the stack.

To handle architectural constraints, we include an *instruction fix-up pass*. For example, x86-64 does not permit direct memory-to-memory operations, such as moving data from one memory address to another. To resolve this, the fix-up pass introduces a temporary register to mediate such operations. While this approach simplifies the lowering process by decoupling operand validation from core logic, it reduces the number of available registers for register allocation by two.

4 Beyond the Specification

4.1 Our extensions

- Added support for conditional if-expressions, similar to if-statements, with syntax `'if' <expr> 'then' <expr> ('elif' <expr> 'then' <expr>)* 'else' <expr> 'fi'`.
- Expanded if-statements to contain an arbitrary number of `elif` clauses between the `if` and `else` clauses; optionally the `else` clause can be ommited.
- Additional Loop Control Statements and Features
  - Added support for Java-style (optional) loop-labels, with syntax `<label> ::= (<ident> ':')?`.
  - Added `break` and `nextloop` (WACC-equivalent of `continue`) statements; they can optionally name a loop-label to target, enabling outer-loop control from a nested inner-loop.
  - Added pure **loop-do** statement which represents an infinite loop with the new syntax of `<label> 'loop' 'do' <stmt> 'done'`.

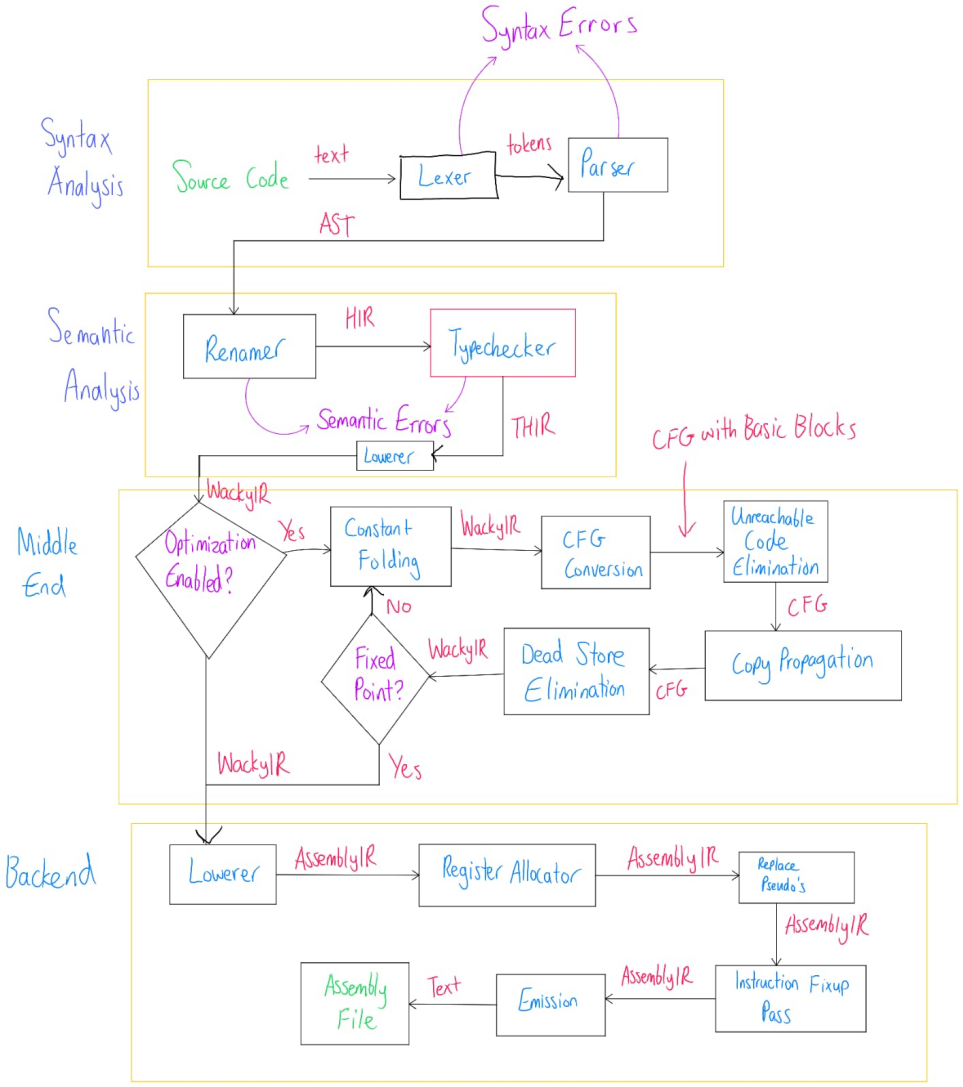

Figure 1: System Architecture Diagram of our WACC Compiler

  - Added **do-while** statement which runs a body once, and *then* loops conditionally; it has syntax of `<label> 'do' <stmt> 'while' <expr> 'done'`.

- Optimisation -Tail-Recursion Otimisation
  - The WACC language grammar doesn't allow for **direct** tail-call returns (e.g. `return call foo(bar)`), so we defined a tail-call to be: a variable-definition function-call, followed immediately returning that variable.
  - If a function has at least one self-recursive call, and all self-recursive calls are tail-calls *(as defined above)* then we define that function to be tail-recursive.
  - A `tailrec` modifier was added to start of function definitions, to make the compiler enforce that a function is tail-recursive; tail-recursive functions without that modifier are still optimized.
  - The body of a tail-recursive function is placed in a giant outer `loop-do` statement, and all tail-recursive calls are replaced with in-place parameter/argument variable updates, followed by a `nextloop <outer-tailrec-loop>` statement to jump to the start of the function again.

- Optimisation - Constant Propagation (and Folding) + Local Static Analysis - We perform this on our TAC IR. We simply evaluate any instruction with only constants and an operand. We also do

compile time checks if we overflow or divide by zero or free a null. If these happen, we replace the instruction with a jump to an error handler. We also convert all conditional jumps ie JmpIfZero(0) with an unconditional Jump and so on. We also have a separate copy propagation pass using forward dataflow analysis that enables us to propagate constants as well as if x = y along both paths of an if statement. These passes together implement the 2 extensions. In a long nested constant expression, we found that we could get up to 500x speedup as seen in our table's.

- Optimisation - Control-Flow analysis - We perform unreachable code elimination by performing a depth first search through our CFG and removing any basic blocks that can't be reached. We also remove useless jumps and labels and remove any empty basic blocks. We also perform a separate dead-store elimination using a backwards dataflow liveness analysis to remove any stores that aren't used later on. Altogether this and Constant Propagation and Folding work really well together in a loop, this itself just cleans up the assembly code and doesn't help speed up much. However it reduce the size of the binary generated.

- Optimisation - Efficient Register Allocation - We first construct an interference graph and attempt to coalesce registers. We unify pseudo-registers and hard registers using the union-find data structure. We perform Brigg's test and George's test to know if it's safe to coalesce registers without making it harder to colour. After coalescing, we attempt to colour the interference graph. Then we use this graph to replace pseudoregisters with hard physical registers. Coalescing removes any redundant copies (such as `mov, rax, rdx` , `mov rdx, rax`) from our previous lowering but doesn't really speedup much since move's are very cheap as seen in the difference between table 3 and 4. It also removes the redundant storing of parameters into another register in a function. Furthermore at least from the tests I've done, there doesn't seem to be a noticeable improvement using register allocation compared to stack based locations (as seen between Table 2 and 3) other than `manyArgs.wacc`. Perhaps larger and longer programs would see a bigger difference.

4.2 CLI

The language extensions are enabled by default and cannot be easily disabled. On the other hand, our compiler CLI interface allows the user to enable or disable optimizations and to view the IR after the selected phase. We also have the option of printing out the Control Flow Graph.

4.3 Future extensions

Our team managed to implement many extensions, but there are some more we wish we could include if we had more time.

1. **Optimisations to do with Loops and Inlining**. Our optimisations currently don't work with functions and loops with constant amount of iterations. If we had inlining and loop unrolling, we'd be able to turn a lot more programs into one liner print statements.

2. **Garbage Collector**. Considering that our compiler provides a decent type of information, and the simplicity of WACC language even with implemented extensions, constructing a basic semi-space garbage collector could be feasible given more time.

5 Appendix

The tests are constructed using some `.wacc` files from the testsuite but running them inside loops. These numbers are found using the following commands:

1. `llvm-mca manyArgs.s | grep "Block RThroughput"`

2. `perf stat ./executable`

3. `perf stat ./compile opt-test/benchmarking/printAllTypesLong.wacc`

File Name	Compile Time (ms)	Runtime (ms)	User Time (ms)	llvm-mca cycle
mediumSplitExprLoop.wacc	7	82	2	533
fibonacciIt.wacc	3	1090	132	23
fibonacciTailRec.wacc	3	491	84	16
fixedPointLoop.wacc	5	154	23	123
longSplitExpr2Long.wacc	335	1	1	5042
longSplitLoop.wacc	351	1319	976	5040
manyArgs.wacc	3	532	252	35
printAllTypesLong.wacc	7	4	1	14

Table 1: Performance metrics without any optimizations

File Name	Compile Time (ms)	Runtime (ms)	User Time (ms)	llvm-mca cycle
mediumSplitExprLoop.wacc	18	8	2	12
fibonacciIt.wacc	3	1089	124	22
fibonacciTailRec.wacc	3	498	61	16
fixedPointLoop.wacc	5	153	20	107
longSplitExpr2Long.wacc	944	1	0	9
longSplitLoop.wacc	1378	686	79	12
manyArgs.wacc	3	533	241	26
printAllTypesLong.wacc	914	4	0	14

Table 2: Performance metrics with O0 flag - Constant Folding and Propagation, Unreachable code elimination and dead code elimination

File Name	Compile Time (ms)	Runtime (ms)	User Time (ms)	llvm-mca cycle
mediumSplitExprLoop.wacc	20	8	2	11
fibonacciIt.wacc	4	1110	129	16
fibonacciTailRec.wacc	3	521	66	16
fixedPointLoop.wacc	5	150	16	105
longSplitExpr2Long.wacc	932	1	0	9
longSplitLoop.wacc	944	685	69	11
manyArgs.wacc	4	527	248	16
printAllTypesLong.wacc	1000	4	0	14

Table 3: Performance metrics with O1 flag (O0 + register allocation)

File Name	Compile Time (ms)	Runtime (ms)	User Time (ms)	llvm-mca cycle
mediumSplitExprLoop.wacc	20	8	1	11
fibonacciIt.wacc	3	1116	130	16
fibonacciTailRec.wacc	3	508	55	16
fixedPointLoop.wacc	6	156	20	104
longSplitExpr2Long.wacc	932	1	0	9
longSplitLoop.wacc	1284	702	79	11
manyArgs.wacc	3	533	268	14
printAllTypesLong.wacc	1092	3	1	14

Table 4: Performance metrics of O2 flag (O1 + register coalescing)