ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica — Scienza e Ingegneria
Corso di Laurea Magistrale in Informatica

An Ahead-of-Time and On-Device WebAssembly Compiler for Resource Constrained Devices

Relatori:
Prof. Ivan Zyrianoff
Prof. Davide Berardi
Prof. Marco Di Felice

Presentata da:
Luca Orlandello

Sessione Marzo 2026
Anno Accademico 2024/2025

Abstract

Existing WebAssembly Ahead-of-Time compilers rely on heavyweight frameworks such as LLVM and require gigabytes of RAM, making on-device compilation on resource-constrained devices infeasible. This thesis presents `wasmc`, a lightweight Ahead-of-Time compiler that translates WebAssembly modules into native 32-bit RISC-V machine code directly on the `ESP32-C3` microcontroller (400 KB of RAM). Key design decisions—function-at-a-time compilation, a memory efficient single pass control flow graph construction in static single assignment form adapted from Braun et al. [3], input proportional heap allocation, zero-copy WebAssembly module decoding, the deliberate omission of an intermediate representation optimizer (since WebAssembly modules are typically produced by optimizing compilers), and a back end implemented from scratch featuring linear scan register allocation, ABI enforcement, static single assignment form deconstruction, and naive macro expansion instruction selection—keep peak heap usage within a few kilobytes per function. Benchmarks on ten programs show that `wasmc` compiles all of them directly on the device, with peak heap usage never exceeding 16.8 KB and compilation time never exceeding 26.6 ms. The produced executables achieve 9.8×–24.2× speedups over the WebAssembly Micro Runtime interpreter. A secondary contribution of this thesis is the first complete documentation of the WebAssembly Micro Runtime AOT executable file format by reverse engineering the WebAssembly Micro Runtime source code. This documentation is complemented by `readaot`, an inspection utility analogous to `readelf` for AOT binaries. These results demonstrate that full Ahead-of-Time compilation of WebAssembly modules on a resource-constrained microcontroller is practically feasible without sacrificing correctness or execution efficiency.

Contents

Chapter 1

Introduction

WebAssembly (WASM) is a binary instruction format for a stack-based virtual machine. It was first introduced and designed in 2017 [10]. Originally, WASM was developed as an alternative to JavaScript in web browsers to enable high-performance applications on the Web. It was designed to be safe, fast, portable and a compact compilation target for high-level languages such as C, C++, Rust, and Go. Currently, major web browsers, including Chrome, Firefox, Safari, and Edge, support the execution of WASM. Although WASM was initially proposed for the Web, it has rapidly expanded into other environments, including resource-constrained devices such as Internet of Things (IoT) devices and embedded systems [30]. Its appeal in these contexts stems from its portability, its sandboxed execution model, and the availability of mature toolchains for compiling existing codebases to the WASM target. A WASM module can be executed either in a web browser using the browser's WASM runtime or outside web browser using a standalone WASM runtime. As outlined in a recent survey [44] on WASM runtimes from October 2025, there exists a wide variety of runtimes, including interpreters, Just-In-Time (JIT) compilers, and Ahead-of-Time (AoT) compilers. Table 1.1 lists open-source, fully functional, standard-compliant, and actively maintained WASM runtimes available both on the Web and outside the Web as a standalone runtime.

WASM Runtime	Language	Standalone	Interpreter	JIT	AoT
V8 (Chrome/Edge) [29]	C++	✗	✓	✓	✗
SpiderMonkey (Firefox) [21]	C, C++	✗	✓	✓	✗
WebKit-JavaScriptCore (Safari) [39]	C++	✗	✓	✓	✗
Wizard [27]	Virgil	✓	✓	✓	✗
Wazero [26]	Go	✓	✓	✗	✓
wasmtime [35]	Rust	✓	✗	✓	✓
wasmer [33]	Rust	✓	✗	✓	✓
WAMR [6]	C	✓	✓	✓	✓
WasmEdge [32]	C, C++	✓	✗	✗	✓
wasm3 [31]	C	✓	✓	✗	✗
wasmi [34]	Rust	✓	✓	✗	✗

Table 1.1: Open-source, fully functional, standard-compliant, and actively maintained WASM runtimes.

Despite the high number of WASM runtimes, only `wasmi`, `wasm3`, and the WebAssembly Micro Runtime (WAMR) interpreter, `iwasm`, can run on resource-constrained devices. Hence, the only practical way to execute a WASM module on such devices is currently through an interpreter. None of the runtimes listed in Table 1.1 support AoT or JIT compilation on resource-constrained devices. Resource-constrained devices present a major challenge for WASM runtimes, as they typically provide only a few dozen to a few hundred kilobytes of RAM. For example, the `ESP32-C3` microcontroller has only 400 KB of RAM [8, p. 4]. This memory limitation renders most existing WASM runtimes, especially JIT and AoT compilers, impractical. All WASM JIT or AoT compilers in Table 1.1 are built on top of heavyweight compiler back-end frameworks (e.g., LLVM [16], Cranelift [5]) that cannot be easily ported to devices with very limited RAM. The objective of this thesis is to enable AoT compilation of WASM modules directly on resource-constrained devices. Achieving this requires a compiler that is itself lightweight enough to run on a microcontroller: it must operate within a few hundred kilobytes of RAM, avoid heavyweight dependencies, and yet produce correct and efficient native code. This is a fundamentally different design point from existing WASM AoT compilers, which are designed to run on a host machine with gigabytes of RAM and then deploy the compiled artifact to the target. To this end, this thesis presents `wasmc`, a lightweight AoT compiler that translates WASM modules into native 32-bit RISC-V machine code directly on a resource-constrained device. `wasmc` is implemented for the `ESP32-C3` microcontroller. Several key design decisions make this possible:

- **Function-at-a-time compilation.** `wasmc` compiles one WASM function at a time. For each function, a Control Flow Graph (CFG) in Static Single Assignment (SSA) form is constructed as an intermediate

representation (IR), passed to the back end to emit native code, and then immediately freed. At any given time, at most one CFG resides in memory, bounding peak heap usage to the size of the largest function rather than the size of the entire module.

- **Memory-efficient CFG in SSA from construction.** The compilation from WASM to the IR is performed in a single pass using a variant of the algorithm by Braun et al. [3], adapted to WASM. It requires no preliminary dataflow analysis and no complex intermediate data structures. This keeps the heap memory allocation during the CFG in SSA from construction phase minimal.

- **Input-proportional memory allocation.** The compiler avoids large fixed-size buffers entirely. Every data structure is heap allocated according to information extracted from the WASM binary itself (e.g., number of functions, function code size, types, globals, and local variables), so memory consumption scales with the actual workload rather than a worst-case estimate.

- **Zero-copy WASM decoding.** During WASM module decoding, large blocks of data within the WASM binary module are not copied into the internal data structures. Instead, pointers are kept directly in the WASM raw binary, avoiding redundant allocations.

- **No IR optimizer.** `wasmc` deliberately omits an IR optimization pass, as IR optimizations are generally too costly in terms of time and memory for a resource-constrained device. This is acceptable because WASM modules are typically produced by optimizing compilers such as `clang`, so the input has already been heavily optimized before reaching `wasmc`.

- **Lightweight back end built from scratch.** Rather than relying on a heavyweight framework, `wasmc` implements its own back end pipeline from scratch: linear scan register allocation [19], Application Binary Interface (ABI) constraint enforcement, SSA deconstruction via parallel move serialization [23] and critical edge splitting [4, pp. 20–24], and instruction selection via naive macro expansion [2, Secs. 2.1 and 2.2]. The pipeline is designed and implemented with memory efficiency as a primary concern.

In addition to the compiler itself, this thesis provides a description of the

WAMR Ahead-Of-Time (AOT[1]) executable format, which `wasmc` adopts as the output format for generated executables. The AOT format is designed to facilitate the compilation of WASM modules into machine code. It is not a standard executable format; rather, it is used and developed by the WAMR project for its off-device AoT compiler, `wamrc`. An off-device AoT compiler performs compilation on a host machine prior to deployment on resource-constrained devices. However, the AOT format is not officially documented. The description presented in this thesis was obtained by reverse engineering the open-source WAMR source code, and documenting it constitutes one of the main contributions of this work. As a result, the `wasmc` compiler is compatible with the WAMR project: WASM modules can be compiled into AOT executables by `wasmc` and then executed by the WAMR AOT runtime entirely on resource-constrained devices. To reverse engineer and debug the AOT format, a command-line tool, `readaot`, has been developed for inspecting AOT binaries. The source code of the `wasmc` compiler and of the `readaot` utility are available at:

`https://github.com/llllluca/wasmc`

The remainder of this thesis is organized as follows: Chapter 2 introduces the background concepts on which `wasmc` is built, including WASM, parallel moves, three-address code, CFG, and SSA form. Chapter 3 describes the design goals of `wasmc`, its overall architecture, and usage. Chapter 4 presents the algorithm for compiling a WASM function into a CFG in SSA form, the IR used internally by `wasmc`. Chapter 5 describes the implementation of the `wasmc` back end in detail. Chapter 6 documents the non-standard AOT format for executables used as the output of `wasmc`. Chapter 7 presents benchmarks evaluating compilation time, peak memory usage during compilation, and execution performance of the generated code. Chapter 8 concludes the thesis. Appendix A presents the `wasmc` implementation status of the WASM 1.0 specification [25] and Appendix B briefly describes the capabilities of `readaot`.

1.1 Related Works

This related work section focuses on WASM runtimes for resource-constrained devices. On the interpreter side, `wasm3` [31], `wasmi` [34], and WAMR's `iwasm` [6] are the lightest fully functional options. There are also research projects and proof-of-concept implementations: `WARDuino` [9] is designed for

[1]The notation is somewhat unfortunate: AoT refers to the Ahead-of-Time compiler, while AOT refers to the Ahead-Of-Time executable format.

Arduino-compatible microcontrollers, and Jacobsson et al. [11] demonstrated one of the earliest runtimes on a wearable nRF52 board using Bluetooth Low Energy updates. On the compiled side, eWASM's `aWsm` [18] and WAMR's `wamrc` [6] are the only fully functional off-device AoT compilers that target resource-constrained devices. Both use LLVM to cross-compile WASM into native code on a host machine before deployment, achieving near-native performance but requiring off-device toolchains. In particular, the WAMR project designed a custom executable format called the AOT format. The `wamrc` compiler translates a WASM module into an AOT executable, which is then sent to the resource-constrained device, where the WAMR AOT runtime can execute it. Similarly, `ThingSpireOS` [14] and `Wasmachine` [40] follow the same off-device LLVM approach. Both projects provide a more comprehensive framework than a simple WASM runtime; in fact, they are referred to as "WebAssembly OS." `WAIT` [13] is the closest prior work to `wasmc`, being the first to perform AoT compilation directly on a constrained device. However, it uses a stack-emulation code generation strategy that trades execution speed for a minimal memory footprint during compilation. The compiled WASM module uses the call stack not as a stack of activation records, but as a stack of integer and floating-point values and a single WASM binary instruction (e.g., `i32.add`) is implemented using four native instructions: two load/pop operations, one addition between two registers and one store/push to mimic the stack-based execution of WASM. In contrast, `wasmc` performs full AoT compilation entirely on-device, without relying on LLVM or stack emulation, targeting 32-bit RISC-V on the `ESP32-C3` microcontroller.

Chapter 2

Background

This chapter presents the essential background necessary to understand the subsequent chapters and the development of `wasmc`. It introduces and explains the fundamental concepts of WebAssembly, parallel moves, three-address code, control flow graphs, and static single assignment form.

2.1 WebAssembly

WASM is a binary instruction format designed for a stack-based virtual machine. It was originally proposed in 2017 [10] to improve the performance of web applications that relied heavily on JavaScript. Currently, major web browsers, including Chrome, Firefox, Safari, and Edge, support the execution of WASM. Despite its web origins, WASM was designed from the outset with safety, portability and efficiency in mind, making no assumptions about the underlying browser, hardware or operating system. This design decision has allowed Wasm to spread well beyond the browser into other domains, including IoT and resource-constrained environments [30]. Developers produce WASM binaries in two ways. The most common path is to write source code in a high-level language and compile it using a compiler that targets WASM. For example, `clang` compile C/C++ programs to WASM. Alternatively, developers can write WASM programs directly in the WASM *text format* (`.wat`), a human-readable s-expression syntax, and convert it to the binary format (`.wasm`) using the `wabt` toolkit [36]. This second path is mainly used for learning, debugging, and small hand-written modules. Wasm uses a *stack machine* model. Instructions operate by pushing and popping typed values (32-bit and 64-bit integers, and 32-bit and 64-bit floats) on an operand stack. Control flow is strictly structured: there are no arbitrary jumps; instead, the language provides `block`, `loop`, and `if` constructs that enforce

structured control-flow. Memory in WASM is a *linear memory*: a contiguous array of bytes. Code can read and write any byte offset within the allocated range. The WASM specification is maintained by the W3C WebAssembly Working Group and evolves through a proposal-driven process. There are currently three versions of the WASM specification: version 1.0 [25], 2.0 [38], and 3.0 [37]. Crucially, successive versions of the WASM specification are not competing alternatives; rather, they form a strictly cumulative, backward-compatible lineage, analogous to how C++11, C++14, and C++17 relate to one another. A binary that is valid under WASM 1.0 remains valid and behaves identically under a WASM 2.0 or WASM 3.0 runtime; nothing is removed or redefined in a breaking way. Each new numbered version is therefore best understood as a snapshot that bundles together all proposals that have completed standardization since the previous release. In practice, runtime implementations often adopt individual proposals incrementally and independently of version boundaries, so capabilities are typically tracked per proposal rather than strictly by version number. `wasmc` implements the WASM 1.0 specification. However, not all the specification is currently supported; Appendix A summarizes the implementation status of WASM 1.0 in `wasmc`.

2.2 Parallel Move

A *parallel move* [23, Secs. 1-2] is a set of assignments that behave as if they all occur simultaneously, rather than sequentially. A parallel move is written as

$$(x_1, \ldots, x_n) := (y_1, \ldots, y_n)$$

where each x_i and y_i is a variable or register name. Some variables may appear on both the left-hand side and the right-hand side. The semantics of a parallel move are as follows: first, all right-hand side values $y_1, \ldots, y_n$ are read; then, all left-hand side variables $x_1, \ldots, x_n$ are updated. This ensures that no assignment interferes with another. For example, consider the parallel move

$$(a, b, c) := (b, c, a)$$

if initially $a = 1$, $b = 2$, and $c = 3$, then after executing the parallel move, the values become $a = 2$, $b = 3$, and $c = 1$. Since left-hand side variables may appear on the right-hand side, the effect of a parallel move generally differs from that of the sequence of elementary assignments $x_1 := y_1; \ldots; x_n := y_n$. Compiling a parallel move amounts to finding a *serialization*: a sequence of single assignments whose effect on the left-hand side variables x_i is identical to that

of the parallel move. Because parallel moves may implement variable permutations (e.g., $(x_1, x_2) := (x_2, x_1)$), the generated sequence of assignments generally cannot operate in-place on the x_i. Additional temporary variables are typically required. The `wasmc` compiler implements the Leroy serialization algorithm [23]. Parallel moves are a special case of the more general concept of *parallel assignment*, which is a programming construct available in many high-level languages. For example, see the assignment statements in the Python documentation [20, Sec. 7.2. Assignment statements].

2.3 Three-Address Code

Three-address code (TAC) [1, Sec. 6.2][7, Sec. 5.3.2] is a low-level, assembly-like language that smooths out the irregularities of the underlying hardware and allows the use of an arbitrary number of variables. TAC is often used as an IR in compilers. In the context of TAC and IRs, variables are also called *temporaries* or *virtual registers*. TAC is called "three-address" because each instruction can reference at most three addresses (or operands). Some works in the literature use the term *operand* instead of *address* (e.g., [42]). In this document, the term *operand* is used when referring to one of the three addresses of a TAC instruction. In its most common form, a TAC instruction has one output operand (or destination operand) and two input operands (or source operands):

$$x = op\ y\ z$$

where x is the output operand (where the result is stored), y is the first input operand, z is the second input operand, and *op* is the operation (e.g., $+$, $-$, $*$, $/$). The three operands can be one of the following:

- A *name*. For convenience, source-program names may appear as operands in three-address code. For example, the name of a function may appear in a call instruction. In an implementation, a source name is typically replaced by a pointer to its symbol-table entry, where all information about the name is stored.

- A *constant*. In practice, a compiler must handle many different types of constants and variables.

- A *compiler-generated variable*. These are temporary variables introduced by the compiler during translation.

Not all TAC instructions use all three operands, but three is the maximum. Some instructions use only two operands, such as unary operations or

copy instructions, which have one output operand and one input operand. There are also TAC instructions that do not produce an output operand but only consume input operands. For example, the TAC store instruction has three input operands: *store* y, *offset*(*memPtr*). The remainder of this section describes the three-address code used by `wasmc`. Each `wasmc` TAC instruction belongs to one of the following categories: integer arithmetic operations, bitwise operations, comparison operations, memory operations, function definitions and calls, data movement operations, and jumps. Instructions operate on 32-bit integer variables (written with a `%` prefix) and 32-bit immediate integer constants. 32-bit floating-point values and 64-bit integers or floating-point values are not currently supported. The following tables list all `wasmc` TAC instructions, their syntax, and their semantics. The meta-variables `A` and `B` denote either a 32-bit integer variable or a 32-bit immediate value. In the "result" column of each table, a C-like syntax is used to describe the semantics of the instruction.

Integer Arithmetic Operations

Instr	Description	Use	result
`add`	Addition	`%d = add A, B`	$d = a + b$
`sub`	Subtraction	`%d = sub A, B`	$d = a - b$
`mul`	Multiplication	`%d = mul A, B`	$d = a \times b$
`sdiv`	Integer Division (signed operands)	`%d = sdiv A, B`	$d = a \div b$
`udiv`	Integer Division (unsigned operands)	`%d = udiv A, B`	$d = a \div b$
`srem`	Remainder (signed operands)	`%d = srem A, B`	$d = a \mod b$
`urem`	Remainder (unsigned operands)	`%d = urem A, B`	$d = a \mod b$

Bitwise Operations

Instr	Description	Syntax	Semantics
`and`	Bitwise AND	`%d = and A, B`	$d = A \,\&\, B$
`or`	Bitwise OR	`%d = or A, B`	$d = A \mid B$
`xor`	Bitwise XOR	`%d = xor A, B`	$d = A \oplus B$
`shl`	Shift left	`%d = shl A, B`	$d = A << B$
`lshr`	Logical shift right	`%d = lshr A, B`	$d = A >> B$
`ashr`	Arithmetic shift right	`%d = ashr A, B`	$d = A \gg_a B$

Comparison Operations

Instr	Description	Use	result
`eqz`	Equal zero	`%d = eqz A`	$d = (a == 0)$
`eq`	Equal	`%d = eq A, B`	$d = (a == b)$
`ne`	Not equal	`%d = ne A, B`	$d = (a \neq b)$
`slt`	Less than (signed operands)	`%d = slt A, B`	$d = (a < b)$
`ult`	Less than (unsigned operands)	`%d = ult A, B`	$d = (a < b)$
`sgt`	Greater than (signed operands)	`%d = sgt A, B`	$d = (a > b)$
`ugt`	Greater than (unsigned operands)	`%d = ugt A, B`	$d = (a > b)$
`sle`	Less equal than (signed operands)	`%d = sle A, B`	$d = (a \leq b)$
`ule`	Less equal than (unsigned operands)	`%d = ule A, B`	$d = (a \leq b)$
`sge`	Greater equal than (signed operands)	`%d = sge A, B`	$d = (a \geq b)$
`uge`	Greater equal than (unsigned operands)	`%d = uge A, B`	$d = (a \geq b)$

Memory Operations

Let *mem* denote a byte-addressable memory array. The effective address is computed as $B + \mathit{offset}$. `zero_extend` extends an 8-bit value to 32 bits by filling the higher-order bits with zeros, while `sign_extend` replicates the most significant bit of the value in the higher-order bits in order to preserve the sign of a signed integer. $A[7:0]$ takes the least significant 8 bits of A.

Instr	Description	Syntax	Semantics
`store`	Store 32-bit word	`store A, offset(B)`	$mem[B + offset : B + offset + 3] = A$
`store8`	Store 8-bit byte	`store8 A, offset(B)`	$mem[B + offset] = A[7:0]$
`load`	Load 32-bit word	`%d = load offset(B)`	$d = mem[B + offset : B + offset + 3]$
`uload8`	Load 8-bit unsigned	`%d = uload8 offset(B)`	$d = \text{zero\_extend}(mem[B + offset])$
`sload8`	Load 8-bit signed	`%d = sload8 offset(B)`	$d = \text{sign\_extend}(mem[B + offset])$

Function Definitions and Calls

Instr	Description	Syntax
`call`	Call a non-void function	`%d = call funcName`
`call`	Call a void function	`call funcName`
`par`	Prepare a parameter	`par %p`
`arg`	Pass argument to function call	`arg A`

Example: call function `f` with arguments `x1` and `x2`.

```
arg x1
arg x2
call f
```

Data Movement Operations

Instr	Description	Syntax	Semantics
`copy`	Copy value	`%d = copy A`	Copy the value of A into the destination d
`push`	Push to stack	`push A`	Push the value of A onto the top of the stack
`pop`	Pop from stack	`%d = pop`	Pop the top value from the stack and store it in d

Jumps

Instr	Description	Syntax	Semantics
`jmp`	Unconditional jump	`jmp @L`	Transfer control unconditionally to label L
`jnz`	Conditional jump (non-zero)	`jnz A, @L1 @L2`	If $A \neq 0$, jump to label $L1$; otherwise, jump to label $L2$

2.4 Control Flow Graph

A *Control Flow Graph* [1, Sec. 8.4][7, p. 231] is a fundamental data structures in a compiler. A CFG is commonly used as a IR of a function, making the flow of control within the function body explicit. A CFG is a graph representation of all the paths that a function might traverse during its execution, nodes are blocks of instructions and edges correspond to possible control transfers between these blocks. Formally, a CFG is a directed graph $G = (N, E)$, where each node $n \in N$ corresponds to a *basic block* and each edge $e = (n_i, n_j) \in E$ represents a possible transfer of control from basic block n_i to basic block n_j during execution. A *basic block* is a sequence of straight-line, branch-free instructions. When control enters a basic block, execution always begins at the first instruction and exits at the last instruction. The instructions inside a basic block are typically not the original source-language instructions, instead, they are usually a three-address code. Each basic block begins with a labeled instruction and ends with a branch or jump instruction. Since jumps can only target the start of a basic block and no jumps occur in the middle of a block, a basic block represents a sequence of instructions that always execute together. Figure 2.1 shows a CFG for an iterative Fibonacci algorithm. Execution begins at the basic block `@start` Because a CFG represents a function, the starting basic block

typically contains the function parameters. In this example, there is a single parameter: `%n par`. The instructions within each basic block are expressed in three-address code, as used by `wasmc`. This CFG illustrates the flow of control for computing the n-th Fibonacci number iteratively:

1. Block `@start` checks the base case.
2. Block `@0` returns the input parameter if it is less than 2.
3. Blocks `@1` and `@2` implement the iterative loop.
4. Block `@3` returns the computed Fibonacci number.

Edges indicate all possible control transfers, including the loop back edge from `@2` to itself, which represents the iteration.

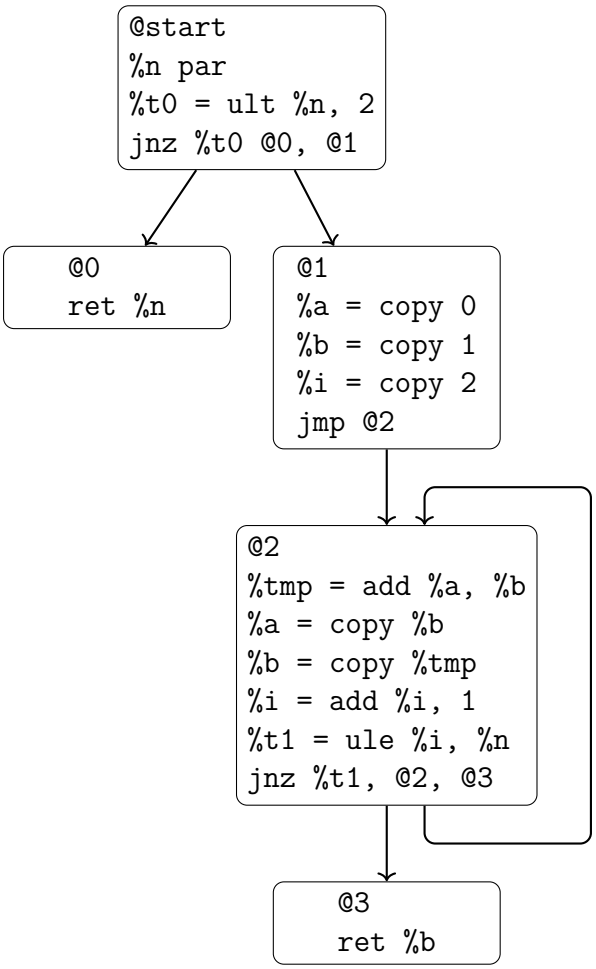
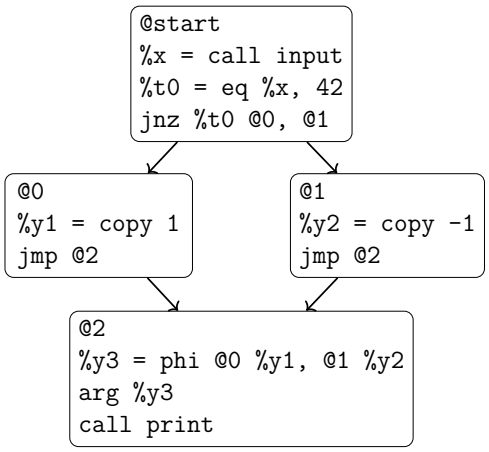


Figure 2.1: Control flow graph for iterative Fibonacci algorithm.

2.5 Single Static Assignment Form

The *Static Single Assignment* [22, Ch. 1][1, Sec. 6.2.4][7, Sec. 5.4.2] form imposes a constraint on variables in the basic blocks of a CFG and introduces a special statement called the ϕ-function. The constraint requires that each variable be assigned exactly once and never overwritten, similarly to variables in functional programming.[1] This uniqueness property greatly simplifies data flow analysis and enables more efficient optimizations. Enforcing this constraint creates difficulties when the front end of a compiler translates arbitrary programs into a CFG in SSA form. Consider the code snippet shown on the left.

```
x = input()
if x == 42:
    y = 1
else
    y = -1
print(y)
```



In this example, the variable y is assigned in both branches of the `if` statement. Consequently, multiple assignments to y reach the `print` statement. When transforming this program into a CFG in SSA form, the compiler must rename the two assignments to preserve the single-assignment property. The variable y is renamed to y_1 in the "then" branch and to y_2 in the "else" branch. However, which variable should be passed as the argument to `print` in order to preserve the semantics of the original program? If we use y_1, the transformed program behaves incorrectly when $x \neq 42$. If we use y_2, it behaves incorrectly when $x = 42$. Therefore, neither choice is correct. This

[1]The name "static single assignment" refers to this static uniqueness property of variable assignments. An assignment that satisfies the static uniqueness property may still be executed multiple times at runtime. For example, consider a CFG in SSA form containing a loop with an assignment $y = x + 1$ inside the loop body. Although the assignment appears only once syntactically, it is executed at every iteration of the loop. The term "static" means that uniqueness is enforced at compile time (on the program text), not dynamically during execution. There also exists a dynamic single assignment form, but that is beyond the scope of this discussion.

is precisely the purpose of ϕ-functions. A ϕ-function merges values coming from different incoming control-flow paths at a control-flow join point (i.e., at the beginning of a basic block with multiple predecessors). The correct CFG in SSA form is shown on the right. The ϕ-function introduces a new variable y_3 defined as:

$$y_3 = \phi(y_1, y_2).$$

Semantically, y_3 takes the value of y_1 if control reaches block `@2` from block `@0` (the "then" branch), and the value of y_2 if control reaches `@2` from block `@1` (the "else" branch).

2.5.1 The Semantics of the ϕ-Function

A ϕ-function may appear only at the beginning of a basic block. It cannot appear in the middle or at the end of a block. A basic block may contain zero, one, or multiple ϕ-functions. The number of arguments of a ϕ-function must be equal to the number of predecessors of the basic block in which it appears. Let B be a basic block with predecessors $P_1, \ldots, P_n$. Suppose the following ϕ-function appears at the beginning of B:

$$v = \phi(v_1, v_2, \ldots, v_n).$$

Each v_i is defined in predecessor P_i. When control transfers from predecessor P_i to B, the value of v_i is assigned to v. Thus, the ϕ-function selects its argument based solely on the control-flow edge taken to reach B. The semantics become more complex in the presence of multiple ϕ-functions. Suppose that m ϕ-functions appear at the beginning of B:

$$\begin{aligned} v_1 &= \phi(v_1^1, v_2^1, \ldots, v_n^1), \\ v_2 &= \phi(v_1^2, v_2^2, \ldots, v_n^2), \\ &\vdots \\ v_m &= \phi(v_1^m, v_2^m, \ldots, v_n^m). \end{aligned}$$

These ϕ-functions are evaluated *simultaneously*, not sequentially. When control transfers from some predecessor P_i to B, the values $v_i^1, \ldots, v_i^m$ are first read, and only afterwards are the variables $v_1, \ldots, v_m$ updated. In other words, multiple ϕ-functions at the beginning of a block have the semantics of a *parallel move*:

$$(v_1, v_2, \ldots, v_m) := (v_i^1, v_i^2, \ldots, v_i^m).$$

2.5.2 An Example of Control Flow Graph in SSA form

Figure 2.2 shows a CFG in SSA form for an iterative Fibonacci algorithm. Compare it with Figure 2.1, where the SSA property does not hold. In the non-SSA CFG, variables such as `%a`, `%b`, and `%i` appear multiple times on the left-hand side of assignments. In contrast, in Figure 2.2, each variable is assigned exactly once, as required by the SSA property. Also notice how ϕ-functions are used to merge the initialization value (coming from `@start`) and the value from the previous iteration of the loop (coming from `@1` itself). The instructions within each basic block are expressed in three-address code, as used by `wasmc`.

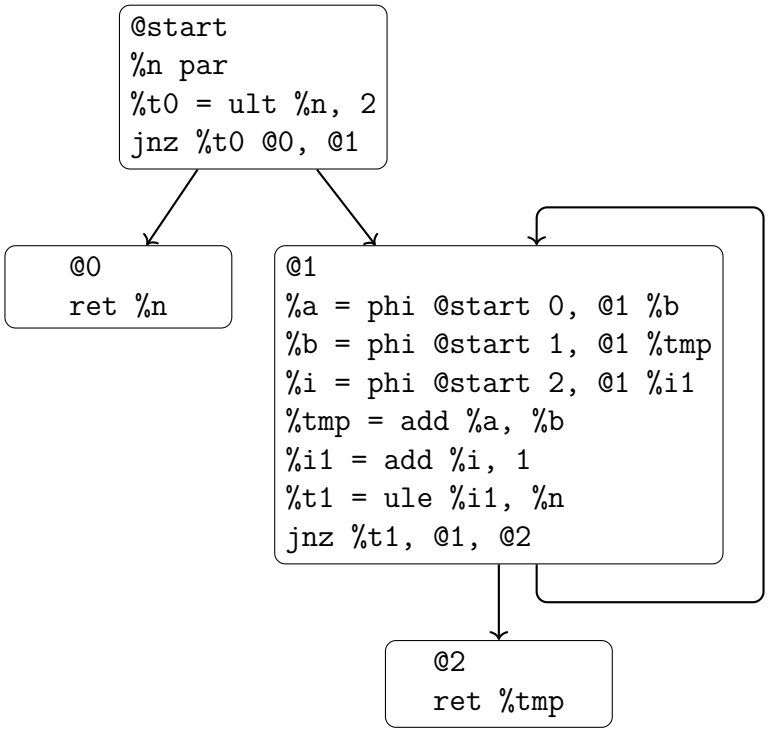


Figure 2.2: CFG in SSA form for an iterative Fibonacci algorithm.

Chapter 3

The `wasmc` Compiler Structure

This chapter describes the architecture and design of `wasmc`, a lightweight WASM compiler targeting resource-constrained devices. Section 3.1 outlines the key design goals that guided its development, namely minimizing memory usage during compilation while preserving the execution efficiency and size of the generated code. Section 3.2 illustrates how `wasmc` fits into a typical development workflow, from the compilation of a high-level language program to a WASM module to the final execution of the native executable on a resource-constrained device. Section 3.3 provides an overview of the `wasmc` compilation pipeline, which consists of a front end and a back. In particular, the front end is responsible for decoding and validating the input WASM module and translating each WASM function into a CFG SSA form, which serves as the IR of `wasmc`. The back end then takes each CFG in SSA form and performs register allocation, ABI constraint enforcement, SSA deconstruction, and instruction selection to produce the native code.

3.1 `wasmc` Design Goals

The design of `wasmc` is guided by the constraints of resource-constrained devices, most importantly their very limited RAM. Accordingly, the design and implementation of `wasmc` must satisfy the following goals:

- **Memory efficiency during compilation**. The primary goal of `wasmc` is to successfully compile WASM modules on resource-constrained devices. Resource-constrained devices typically include IoT devices or microcontrollers running a lightweight Real-Time Operating System (RTOS). The severe memory limitations of such devices render many existing WASM compilation approaches impractical because they are build on top of heavyweight compiler back-end frameworks such as

LLVM or Cranelift [33] [35] [6]. Thus, the most important guideline throughout the development of `wasmc` was to minimize the memory footprint, both statically at compile time and dynamically at run time. Statically, the compiler avoids large fixed-size buffers and instead sizes data structures based on information extracted from the WASM module (e.g., number of functions, types, and globals), ensuring that memory usage scales with the actual workload rather than a worst-case estimate. Dynamically, the compiler processes one function at a time: each function is compiled and its native code emitted before the next one is processed. Temporary data and IR are allocated only for the current function and releasing it immediately afterward to minimize peak heap usage. Another consideration is that `wasmc` may not be the only task running on the system. Consequently, even if most of the RAM is temporarily free, the compiler must avoid large heap peaks that could consume all available memory, since other tasks may require it.

- **Execution efficiency and size of generated code**. Existing approaches to compiling WASM modules on resource-constrained devices (e.g., [13]) often sacrifice execution efficiency and increase the size of the generated code in order to successfully compile a WASM module using only a few kilobytes of RAM. In contrast, `wasmc` is designed with a completely different approach that takes into account both execution efficiency and size of the generated code, while requiring slightly more RAM than the approach in [13]. For a fair comparison, it should be noted that the approach in [13] is designed to work on the `ATMega128` microcontroller, which has only 4 KB of RAM, making it a far more constrained device than the `ESP32-C3` for which `wasmc` was developed.

3.2 `wasmc` Workflow

Figure 3.1 illustrates how `wasmc` is used in practice and how it interacts with other software components. Developers write programs in a high-level language that can be compiled to a WASM module, such as C, C++, Rust, or Go. The program is first compiled into a WASM module, for example `clang` can compile C or C++ programs to the WASM target. The WASM module is then sent to the resource-constrained device in an unspecified way. On The resource-constrained device there are two main component: the `wasmc` compiler and the WAMR AOT runtime. The `wasmc` compiler then takes the WASM module as input and produces a native executable in the AOT format. The AOT format is not a standard executable format; it is specifically

developed and used by the WAMR project [6]. The AOT format is designed to facilitate compilation from a WASM module to machine code. Chapter 6 provides a detailed description of the AOT file format. The `wasmc` compiler supports as its output only executable in the AOT format targeting the 32-bit RISC-V architecture [24]. The compiler is compatible with WAMR, meaning that the AOT executables produced by `wasmc` can be executed by the WAMR AOT runtime on 32-bit RISC-V resource-constrained devices.

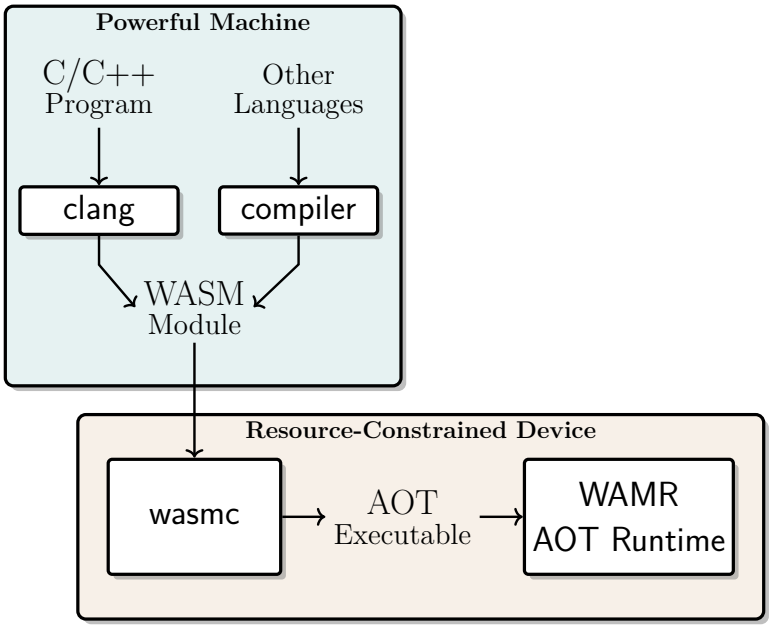


Figure 3.1: Practical usage of `wasmc` and how it connects with other software components.

3.3 Overview of `wasmc` Architecture

Figure 3.2 illustrates the complete structure and compilation pipeline of `wasmc`. The following sections summarize the steps executed by `wasmc`, as represented by the blocks in Figure 3.2. The front end of `wasmc` consists of a three-stage pipeline: (*i*) a WASM module decoding phase (Section 3.3.1), which parses the raw WASM binary into an internal WASM module data structure used for further processing; (*ii*) a WASM module validation phase (Section 3.3.2), which ensures that only well-formed modules can be compiled; and (*iii*) a compilation phase (Section 3.3.3) that translates each WASM function into a CFG in SSA form. The CFG in SSA form is the IR used by `wasmc`. Each WASM function's CFG is then passed to the back

end (Section 3.3.4) for code generation.

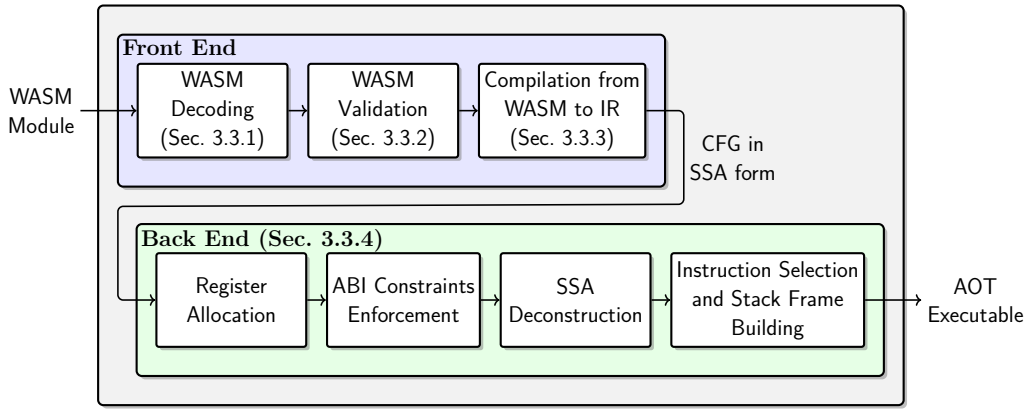


Figure 3.2: Overall structure of the `wasmc` compiler.

3.3.1 WebAssembly Module Decoding

WASM modules are distributed in a binary format [25, Ch. 5]. The decoding phase processes this binary format and converts it into a WASM module data structure. Essentially, the decoding phase acts as a parser that interprets the raw binary format for further processing. The size of the WASM module data structure depends on the input module, so it must be allocated on the heap. Because one of the goals of `wasmc` is to operate on resource-constrained devices, heap allocation should be minimized. Therefore, large blocks of data within the WASM binary module are not copied, instead the WASM module data structure retains pointers to the WASM binary module whenever possible.

3.3.2 WebAssembly Module Validation

The WASM module validation phase checks that a module is well-formed and safe to execute. The WASM specification defines a type system [25, Ch. 3] as a set of typing rules that specify constraints a module must satisfy. Validation is performed before any code generation takes place. If a WASM module violates any of the typing rules defined by the specification, it is rejected and compilation is aborted. Only valid modules can be compiled by `wasmc`. The WASM typing rules perform the following checks:

- **Ensures type and stack consistency.** WASM is a strongly typed stack-based language. Every instruction has a precisely defined effect

on the WASM stack: it consumes a fixed number of values of specific types and produces a fixed number of results. During validation, the stack behavior of each instruction is simulated to ensure that the required operands are present on top of the WASM stack and that their types match the required types. For example, the instruction `i32.add` expects two 32-bit integers on the top of the stack and produces a single 32-bit integer as its result.

- **Verifies control-flow integrity**: WebAssembly uses structured control-flow constructs such as `block`, `loop`, and `if`. The validator ensures that these constructs are properly nested and that branch instructions only target valid enclosing blocks. It also checks that the values produced by control-flow constructs match the expected result types of the corresponding blocks.

- **Validates module structure.** In addition to instruction-level checks, the validator verifies the consistency of the module as a whole. This includes checking that function signatures are well defined, that indices referring to functions, globals, memories, or tables are within bounds, and that imported and exported entities have compatible types.

These typing rules are declarative, they specify constraints but do not define an algorithm. A skeleton of a sound and complete type-checking algorithm is provided in the WASM specification appendix [25, Sec. 7.3]. The `wasmc` validation phase is a complete implementation of this algorithm.

3.3.3 Compilation from WebAssembly to IR

The diagram in Figure 3.2 abstracts away a detail about the interaction between the back end and the "Compilation from WASM to IR" phase. The output of `wasmc` is an AOT executable. An AOT executable contains several sections including the text section, which stores the machine code of each compiled WASM function. Sections must follow a precise order: some sections precede the text section (e.g., target info and init data sections), while others follow it (e.g., function, export, and relocation sections). Immediately after validation, `wasmc` behaves as illustrated in Figure 3.3. It first outputs the sections before the text section, then fills the text section incrementally, function by function. For each function in the WASM module, a CFG in SSA form is created, passed to the back end to generate and output assembly code, and then freed. Thus, both the compilation algorithm and the back end are invoked n times, where n is the number of functions in the WASM module. Finally, `wasmc` outputs the sections after the text section. At any

given time, at most one CFG resides in memory, which significantly reduces peak heap usage. `wasmc` uses a variant of the algorithm by Braun et al. [3], adapted to WASM, to compiler a WASM function into a CFG in SSA form. The Braun et al. algorithm is a simple and efficient algorithm to construct a CFG in SSA form directly from a bytecode in a single pass, without requiring any preliminary dataflow analysis on the bytecode and without the construction complex intermediate data structures. This significantly reduces the peak heap memory usage during compilation. The Braun et al. algorithm constructs the CFG in SSA form incrementally and places ϕ-functions on the fly, as instructions are processed one by one. The algorithm that compiles a WASM function into a CFG in SSA form is described in detail in Chapter 4.

```
# outputs the AOT sections before the text section
emit_target_info_section()
emit_init_data_section()
for i in range(wasm_module.function_count):
    # create the CFG in SSA form of the i-th WASM function
    IRFunction *fn = compile(wasm_module.functions[i])
    # pass the CFG to the backend that outputs the assembly code
    emit_text(fn)
    # free the CFG; at most one CFG is in memory
    free(fn)
# outputs the AOT sections after the text section
emit_function_section()
emit_export_section()
emit_relocation_section()
```

Figure 3.3: Interaction between the back end and the "Compilation from WASM to IR" phase of the front end.

3.3.4 The `wasmc` Back End

In general the main duties of a compiler back end are: (*i*) lowering the intermediate representation to machine instruction while respecting the ABI constraints, (*ii*) composing the stack frames, (*iii*) allocating variable to architectural registers and (*iv*) producing binary code. A common choice for implementing a compiler back end is LLVM. LLVM is a powerful and widely used framework for compiler back ends, but its large memory footprint—both statically at compile time and dynamically at run time—makes it unsuitable for resource-constrained devices. For this reason, `wasmc` adopts a lightweight back end built from scratch to minimize RAM usage. The back end of `wasmc`

implements these steps as a four-stage pipeline consisting of these phases, in order:

1. **Register allocation and assignment**: register allocation involves deciding at each point in a program which variables should reside in registers and which variables should reside on the stack, instead register assignment is about deciding which variables to keep in which registers among those should be present in the registers. Register allocation and assignment are usually refer only as register allocation. `wasmc` implements the *linear scan* algorithm of Poletto and Sarkar [19] adapted to work on a CFG in SSA form. The implementation is explained in Section 5.1.

2. **ABI constraints enforcement**: it ensures that generated code follows the platform's calling conventions and register usage rules so that components compiled separately and with different compilers can interoperate correctly. The implementation is explained in Section 5.2.

3. **SSA Deconstruction**: Since no hardware instruction set directly implements ϕ-functions, SSA deconstruction translates the semantics of ϕ-functions into commonly supported instructions (i.e., copy instructions). SSA deconstruction is sometimes called out-of-SSA translation. `wasmc` implements SSA deconstruction using *parallel move serialization* [23] and insertion of move instructions with *critical edge splitting* [4, pp. 20–24]. The implementation is described in Section 5.3.

4. **Instruction Selection** and **Stack Frame Construction**: Instruction selection maps IR instructions to target-machine instructions. `wasmc` adopts the *naive macro expansion* strategy [2, Secs. 2.1 and 2.2]. Stack frame construction determines the layout of the stack frame, including space for local variables and temporaries, preservation of callee-saved registers. During this phase, the function prologue and epilogue are generated according to the target architecture's calling convention. The implementation is described in Section 5.4.

The Use of the SSA Form in the Back End

The use of the SSA-form in the back end may seem unusual, since traditionally SSA deconstruction is not performed in the back end of a compiler; typically, the back end directly receives a non-SSA CFG as input. Traditionally, in an optimizing compiler, a CFG in SSA form is generated by the frontend, the SSA form of the CFG is widely exploited by the IR optimizer,

and then, after all IR optimizations have been performed and before the back end, the SSA deconstruction step takes place. The advantage of using a CFG in SSA form during the IR optimizations phase is that the SSA form speed up, simplify and improve a lot these optimizations. For the same reasons that the SSA form is useful in the IR optimizer, it has also proved to be very useful in the back ends of more modern compilers [22, Sec. 1.4]. Modern compilers back end tasks have significantly evolved, they performs a lot of machine dependent code analysis and optimizations. The sophistication of modern compiler back end is one of the reasons behind the introduction of the SSA form in the compiler back ends, in order to simplify certain analyses and optimizations [22, pp. 243-244]. In addition, register allocation can be refactored to take advantage of the SSA form and make it more efficient and effective. There are modern and complex algorithms for register allocation designed ad hoc to work best on a CFG in SSA form [22, Ch. 22]. For these reasons the SSA form was more recently introduced also in the back end of modern compilers. As outline in Section 3.3.5, `wasmc` does not have a IR optimizer but the CFG in SSA form is anyway exploited in the back end. For the sake of truth, the `wasmc` back end exploits the CFG in SSA form in a very limited extent, the SSA form is only exploited when constructing live intervals during register allocation phase, see Section 5.1.3.

3.3.5 An IR Optimizer Is Not Needed

One of the most important design choice is that `wasmc` does not include an IR optimizer[1], there is no IR optimization in Figure 3.2. This choice is justified by these two reasons. The first is that optimizations on the intermediate representation are generally too costly in terms of time and memory for a resource-constrained device. The second reason is that WASM modules are usually not written by hand, but are generated by a high-level language. For example, any language with an LLVM-based compiler, such as C, Rust, or Go, can be compiled into a WASM module. Compiling from a program in a high-level language to a WASM module takes place on a powerful machine and with an optimizing compiler, so the WASM module taken as input from `wasmc` has already been heavily optimized. Of course, one could manually write or generate an unoptimized WASM module and pass it to `wasmc`, but this is not the intended use of `wasmc`. Since the input WASM module has already been heavily optimized, `wasmc` does not gain significant benefits from further optimizing the intermediate representation.

[1]The IR optimizer is also often called machine independent optimizer.

Optimization Considerations When Compiling to WASM

Now a technical point about the compilation from a C program (the same considerations apply to other languages) to a WASM module. The `clang` compiler can compile a C program into a WASM module. During this compilation, if an optimization flag is enabled (e.g., `-O1`, `-O2`, `-O3`), `clang` performs optimizations that improve the execution speed of the generated WASM module at the cost of increased code size. Examples of such optimizations include function inlining and loop unrolling, which can significantly increase the size of the resulting WASM module. The peak heap usage of `wasmc` grows with the size of the input WASM module. Consequently, very large modules may cause memory-related issues when `wasmc` is executed on a resource-constrained device. In addition to the `-O1`, `-O2`, and `-O3` optimization levels, `clang` also provides the `-Os` optimization flag. The `-Os` flag instructs the compiler to optimize for code size: it is similar to `-O2`, but disables optimizations that would significantly increase the size of the generated code. For this reason, `-Os` is the most suitable optimization flag when compiling a C program to a WASM module intended to be processed by `wasmc`.

Chapter 4

Compilation Algorithm from WASM to CFG in SSA Form

This chapter presents the skeleton of `wasmc` algorithm for compiling the body of a WASM function into a CFG in SSA form. The body of a WASM function consists of a sequence of WASM instructions, each beginning with an opcode that identifies the operation to be performed. The compilation algorithm takes as input a WASM function and processes the flat sequence of WASM instructions contained in the code section of the WASM binary module. Instructions are translated in the order in which they appear, and the algorithm performs a single pass over the instruction sequence while incrementally constructing the CFG in SSA form. One of the main challenges is handling WASM local variables. Because they are mutable and may be written at any point in the function body, converting them to SSA form requires additional processing. WASM stack values do not have the same problem as local variables when converting to SSA form because stack values are single-use by design and cannot be reassigned. Therefore, in order to correctly process local variables, a variant of the algorithm by Braun et al. [3], adapted to WASM, is used. The Braun et al. algorithm is a simple and efficient algorithm to construct a CFG in SSA directly from a bytecode. The algorithm is simple because it does not require code analysis on the bytecode or complex data structures during the construction of the CFG in SSA form. This simplicity also makes the algorithm memory efficient. The algorithm by Braun et al. constructs a CFG in SSA form on the fly by tracking the current definition of each WASM local variable per block and lazily inserting ϕ-functions when a block is *sealed*, that is, when all of its predecessors in the CFG are known. The `wasmc` compilation algorithm is presented using a Python-inspired pseudocode syntax.

4.1 Data Structures

This section describes the data structures used during the compilation phase to transform a WASM function into a CFG in SSA form. The IR instruction `IRInstr` and IR basic block `IRBlock` data structures are used to represent the CFG produced by the compilation algorithm. In contrast, the compilation context `CompileCtx` stores temporary values and data structures that are relevant only during the compilation process.

4.1.1 IR Instructions

`IRInstr` represents a single three-address code instruction of the IR as a tuple of four elements: an opcode and up to three operands.

```
class IRInstr:
    def __init__(self, ir_opcode, arg0, arg1, arg2):
        self.opcode = ir_opcode
        self.arg0 = arg0
        self.arg1 = arg1
        self.arg2 = arg2
```

4.1.2 IR Blocks

The `IRBlock` data structure represents a basic block in the CFG. Each basic block has at most two successors: `succ0` and `succ1`. Two successors are sufficient to construct the CFG of every WASM function, since all control flow can be expressed using unconditional and conditional jumps. Each `IRBlock` contains a list of IR instructions (`instr_list`) that stores the three-address code instructions belonging to the basic block. The terminating jump of each basic block is not stored in `instr_list`. Instead, it is represented using two separate fields: `jump_type` and `jump_arg`. The `jmp` method sets the terminating jump to an unconditional jump from the current block (`self`) to `target_block`. The `jnz` (jump-not-zero) method sets the terminating jump to a conditional jump: if `arg` is non-zero, control transfers to `then_block`, otherwise it transfers to `else_block`.

```
class IRBlock:
    def __init__(self, sealed=False):
        self.instr_list = list()
        self.succ0 = None
        self.succ1 = None
        self.jump_type = IR_JUMP_TYPE_NONE
        self.jump_arg = None
        self.sealed = sealed
        self.locals = [None] * locals_count
        if not sealed:
            self.incomplete_phi = [None] * locals_count

    def jmp(self, target_block):
        self.succ0 = target_block
        self.succ1 = None
        self.jump_type = IR_JUMP_TYPE_JMP
        self.jump_arg = None

    def jnz(self, arg, then_block, else_block):
        self.succ0 = then_block
        self.succ1 = else_block
        self.jump_type = IR_JUMP_TYPE_JNZ
        self.jump_arg = arg
```

Beyond the CFG structure described above, `IRBlock` carries two additional arrays that support the on-the-fly SSA construction following the algorithm of Braun et al. [3].

The `locals` array. A WASM function has a fixed number of local variables, comprising its parameters and any additional locals declared in the function body; each is identified by a numeric index in the range from 0 inclusive to `locals_count` excluded. At runtime, these variables are mutable: `local.get` and `local.set` instructions read and overwrite them, and these instructions may appear anywhere in the function body. To construct a CFG in SSA form, the compiler tracks, for every basic block, the most recent definition of each local variable seen so far while filling that block. The `locals` array serves this purpose: `locals[i]` stores the IR variable or the left-hand side of a ϕ-function that represents the current value of local variable `i` within this block. When a `local.set` instruction is compiled, the IR variable of the new value is written into `locals[localidx]` of the current block. When a `local.get` instruction is compiled, `locals[localidx]`

is checked first; if a definition is present it is returned immediately (local value numbering). If the entry is absent, the algorithm searches backwards through the predecessor blocks until a definition is found or a ϕ-function is inserted to merge definitions from multiple paths, the result is then written back into `locals[localidx]` to memoize future look-ups in the same block (global value numbering). All entries are initialised to `None` at block creation time; an entry remains `None` as long as no definition for that variable has been written or looked up in the block.

The `incomplete_phi` array. The global value numbering step can only construct a correct ϕ-function for a block once all of its predecessors are known. A block is *sealed* when when all of its predecessors in the CFG are known, only sealed blocks can be queried for a variable definition without risk of producing a ϕ-function that is missing operands from as-yet-unregistered predecessors. The one case where a block must be queried before it is sealed is the loop header: it becomes the current block immediately after creation — so `local.get` instructions inside the loop body may look up variables in it — but its back-edge predecessors are only registered later, when `br` and `br_if` instructions inside the body are compiled. When a variable look-up reaches an unsealed block that has no current definition for the variable, the algorithm cannot yet fill the ϕ-function's operands. Instead, it creates an operandless *proxy* ϕ-function, writes it into `locals[localidx]` to serve as a placeholder and cycle-breaker, and records it in `incomplete_phi[localidx]`. The `incomplete_phi` array therefore holds at most one proxy per local variable index: a non-`None` entry at index `i` means that a proxy ϕ-function was placed for variable `i` and is waiting to have its operands filled in. When the block is eventually sealed, by calling `seal_block()`, each non-`None` entry in `incomplete_phi` is completed: the predecessor blocks are iterated, the reaching definition of the variable is looked up in each one, and the operand is appended to the proxy ϕ-function. Sealed blocks do not allocate the `incomplete_phi` array at all, since no proxy ϕ-functions can arise in a block whose predecessors are all known before any variable look-up is attempted.

4.1.3 Compile Context

The compilation context, `CompileCtx`, has these fields:

- `curr_block` — a pointer to the basic block currently being populated with IR instructions. It changes whenever a control instruction (`block`, `loop`, `if`, `else`, `end`, or a branch) is compiled.

- *Operand stack* — mirrors the WASM value stack, but stores IR variables rather than runtime values. During the compilation, every WASM instruction that consumes or produces a value pops from or pushes to this stack.
- *Control stack* — tracks the nesting of structured control constructs (`block`, `loop`, `if`). Each entry records the information needed to compile branch instructions targeting that construct and to correctly finalise it when the matching `end` is encountered.

Each element on the control stack is an instance of a subclass of `ControlStackEntry`: `BlockControlStackEntry`, `LoopControlStackEntry`, `IfControlStackEntry`, `ElseControlStackEntry`, or `DummyControlStackEntry`. Using distinct subclasses allows the type of a popped entry to be determined at runtime via `isinstance()`.

```
class CompileCtx:
    def __init__(self, curr_block):
        self.opd_stack = list()    # operand stack
        self.ctrl_stack = list()   # control stack
        self.curr_block = curr_block

    def push_opd(self, result):
        self.opd_stack.append(result)

    def pop_opd(self):
        assert(len(self.opd_stack) > 0)
        return self.opd_stack.pop()

    def top_opd(self):
        assert(len(self.opd_stack) > 0)
        return self.opd_stack[-1]

    def push_ctrl_block(self, block_start, block_end,
    ↪  wasm_blocktype):
        ctrl = BlockControlStackEntry(block_start, block_end,
                                      wasm_blocktype,
                                      ↪  len(self.opd_stack))
        self.ctrl_stack.append(ctrl)
```

```
def push_ctrl_loop(self, loop_start, wasm_blocktype):
    ctrl = LoopControlStackEntry(loop_start, wasm_blocktype,
                                 len(self.opd_stack))
    self.ctrl_stack.append(ctrl)

def push_ctrl_if(self, if_start, if_end, wasm_blocktype):
    ctrl = IfControlStackEntry(if_start, if_end,
                               wasm_blocktype,
                               ↪ len(self.opd_stack))
    self.ctrl_stack.append(ctrl)

def push_ctrl_else(self, if_ctrl):
    ctrl = ElseControlStackEntry(if_ctrl)
    self.ctrl_stack.append(ctrl)

def push_ctrl_dummy(self):
    ctrl = DummyControlStackEntry(len(self.opd_stack))
    self.ctrl_stack.append(ctrl)

def top_ctrl(self):
    n = len(self.ctrl_stack)
    assert(n > 0)
    return self.ctrl_stack[n-1]

def pop_ctrl(self):
    assert(len(self.ctrl_stack) > 0)
    ctrl = self.ctrl_stack.pop()
    if ctrl.end_type != WASM_BLOCKTYPE_NONE and (not
    ↪ ctrl.unreachable):
        opd = self.pop_opd()
        ctrl.end_results.append((opd, self.curr_block))
    return ctrl

def get_ctrl(self, index):
    n = len(self.ctrl_stack)
    assert(index < n)
    return self.ctrl_stack[n - (index + 1)]

def is_unreachable(self):
    return self.top_ctrl().unreachable
```

```
def unreachable(self):
    ctrl = self.top_ctrl()
    while len(self.opd_stack) != ctrl.height:
        self.pop_opd()
    ctrl.unreachable = True
    self.curr_block = None
```

The `pop_ctrl` method deserves a brief note. When a control entry is popped, if the construct declares a result type (`end_type` $\neq$ `WASM_BLOCKTYPE_NONE`) and the current region is reachable, the top operand is recorded together with the current block into the entry's `end_results` list. This list collects one (*value*, *predecessor*) pair per control-flow path that falls through to the merge point of the construct; it is consumed later by `get_block_result()` to produce a ϕ-function or a direct reference (Section 4.2.10). The control stack entry classes are defined below. The base class `ControlStackEntry` holds all fields shared by every kind of entry. Two block-type fields are maintained separately:

- `label_type` — the value type expected when *branching to* this construct via `br` or `br_if`.
- `end_type` — the value type produced when execution *falls through* to the matching `end`.

The two fields differ for loops: a branch to a loop transfers control to its *start*, not its exit, so no value is carried on a back-edge and `label_type` is always `WASM_BLOCKTYPE_NONE` for loop entries regardless of the declared block type.

```
class ControlStackEntry:
    def __init__(self, start_block, label_block,
                 label_type, end_type, unreachable, height,
                 end_results=None):
        # basic block where the structured control construct begins
        self.start_block = start_block
        # target basic block for branches to this control construct
        # (i.e., the block reached by br/br_if instructions)
        self.label_block = label_block
        # value type expected when branching to this construct
        self.label_type = label_type
        # value type produced when falling through to end
        self.end_type = end_type
        # indicates whether the current code region is unreachable
        # (e.g., after an unconditional branch)
        self.unreachable = unreachable
        # height of the operand stack at the time the control entry
        # was pushed, used to restore the stack on exit
        self.height = height
        # values contributed by each path that reaches the merge
        ↪ point
        self.end_results = [] if end_results is None else
        ↪ end_results

class BlockControlStackEntry(ControlStackEntry):
    def __init__(self, block_start, block_end, wasm_blocktype,
    ↪ height):
        super().__init__(block_start, block_end,
                         wasm_blocktype, wasm_blocktype,
                         False, height)

class LoopControlStackEntry(ControlStackEntry):
    def __init__(self, loop_start, wasm_blocktype, height):
        # label_type is NONE: br to a loop carries no value
        super().__init__(loop_start, loop_start,
                         WASM_BLOCKTYPE_NONE, wasm_blocktype,
                         False, height)

class IfControlStackEntry(ControlStackEntry):
    def __init__(self, if_start, if_end, wasm_blocktype, height):
        super().__init__(if_start, if_end,
                         wasm_blocktype, wasm_blocktype,
                         False, height)
```

```
class ElseControlStackEntry(ControlStackEntry):
    def __init__(self, if_ctrl):
        # inherit the merge block, types, height, and accumulated
        ↪  results
        # from the matching if entry
        super().__init__(if_ctrl.start_block, if_ctrl.label_block,
                         if_ctrl.label_type, if_ctrl.end_type,
                         False, if_ctrl.height,
                         end_results=if_ctrl.end_results)

class DummyControlStackEntry(ControlStackEntry):
    def __init__(self, height):
        super().__init__(None, None,
                         WASM_BLOCKTYPE_NONE, WASM_BLOCKTYPE_NONE,
                         True, height)
```

4.2 Compilation of WASM Instructions

The body of a WASM function consists of a sequence of WASM instructions. Each WASM instruction begins with an opcode that uniquely identifies the instruction. The remaining bytes of the instruction depend on the specific opcode and encode its immediates. The compilation algorithm reads the WASM instructions directly from the code section of the WASM binary module. For each instruction, the opcode is read first and then passed to the `compile_instr()` function, which dispatches the compilation logic based on the opcode. The `compile_instr()` function below illustrates the compilation of several representative WASM instructions; the remaining instructions are compiled in a similar manner.

```
def compile_instr(ctx, opcode):
    match opcode:
        case WASM_OPCODE_I32_ADD:
            compile_binop(ctx, IR_OPCODE_ADD)
        case WASM_OPCODE_DROP:
            compile_drop(ctx)
        case WASM_OPCODE_SELECT:
            compile_select(ctx)
        case WASM_OPCODE_BLOCK:
            compile_block(ctx)
        case WASM_OPCODE_LOOP:
            compile_loop(ctx)
        case WASM_OPCODE_IF:
            compile_if(ctx)
        case WASM_OPCODE_ELSE:
            compile_else(ctx)
        case WASM_OPCODE_END:
            compile_end(ctx)
        case WASM_OPCODE_BRANCH:
            compile_br(ctx)
        case WASM_OPCODE_BRANCH_IF:
            compile_br_if(ctx)
        case WASM_OPCODE_LOCAL_GET:
            compile_local_get(ctx)
        case WASM_OPCODE_LOCAL_SET:
            compile_local_set(ctx)
        case WASM_OPCODE_I32_LOAD:
            compile_load(ctx, IR_OPCODE_LOAD)
        case WASM_OPCODE_I32_STORE:
            compile_store(ctx, IR_OPCODE_STORE)
        case WASM_OPCODE_I32_CONST:
                compile_i32const(ctx)
        # The remaining cases are omitted
```

4.2.1 Binary Operations

The function `compile_binop()` compile a WASM binary operation (e.g. `i32.add`) into a single IR instruction. If the current region is unreachable, it returns immediately. Otherwise, it pops the two operands from the operand stack, allocates a fresh IR temporary `result` using `IRTmpReference()`, appends the instruction `result = op(opd1, opd2)` to the current block's instruction list, and pushes `result` onto the operand stack. Allocating a fresh

temporary for every result is what keeps the IR in SSA form.

```
def compile_binop(ctx, ir_opcode):
    if ctx.is_unreachable(): return
    opd2 = ctx.pop_opd()
    opd1 = ctx.pop_opd()
    result = IRTmpReference()
    binop = IRInstr(ir_opcode, result, opd1, opd2)
    ctx.curr_block.instr_list.append(binop)
    ctx.push_opd(result)
```

4.2.2 Drop

The `drop` instruction discards the value at the top of the WASM stack without producing a result. `compile_drop()` models this by popping the top IR variable from the operand stack and emitting no IR instruction. If the current region is unreachable, it returns immediately.

```
def compile_drop(ctx):
    if ctx.is_unreachable(): return
    ctx.pop_opd()
```

4.2.3 Select

In WASM, the `select` instruction pops three values from the operand stack: an `i32` condition and two candidate values v_2 and v_1 (in that order, with the condition on top). The instruction pushes back v_1 if the condition is non-zero, and v_2 otherwise. The function `compile_select()` implements the compilation of the WASM `select` instruction. First, it checks whether the current context is unreachable; if so, the instruction is ignored. Otherwise, three values are popped from the operand stack, producing IR variables `cond`, `opd2`, and `opd1`. The `select` instruction is lowered by introducing two auxiliary IR blocks and a ϕ-function that merges the two candidate values based on which control-flow path was taken. Specifically, a block named `select_arg1` is created to represent the path on which `opd1` is selected (condition non-zero), and a block named `merge` serves as the convergence point. The current block (`select_arg2`) is terminated with a conditional jump: if `cond` is non-zero, control goes to `select_arg1`; otherwise, it falls directly to `merge`. The `select_arg1` block ends with an unconditional jump to `merge`. A fresh IR variable `result` is created as the left-hand side of a ϕ-function

inserted at the beginning of `merge`. The ϕ-function assigns `opd1` if control arrives from `select_arg1`, and `opd2` if control arrives from `select_arg2`. Finally, `result` is pushed onto the operand stack. Both `select_arg1` and `merge` have a complete, known predecessor set at the point of creation and are therefore sealed immediately.

```
def compile_select(ctx):
    if ctx.is_unreachable(): return
    cond = ctx.pop_opd()
    opd2 = ctx.pop_opd()
    opd1 = ctx.pop_opd()
    select_arg1 = IRBlock(sealed=True)
    select_arg2 = ctx.curr_block
    merge = IRBlock(sealed=True)
    ctx.curr_block.jnz(cond, select_arg1, merge)
    select_arg1.jmp(merge)
    result = IRTmpReference()
    phi = IRPhi(result)
    phi.args.append(opd1, select_arg1)
    phi.args.append(opd2, select_arg2)
    merge.phi_list.append(phi)
    ctx.curr_block = merge
    ctx.push_opd(result)
```

4.2.4 Block

In WebAssembly, a `block` defines a structured region of instructions that are executed sequentially, from the `block` instruction to the corresponding `end` instruction. The general form is:

`block`
  $instr_1$
  $instr_2$
  ...
  $instr_n$
`end`

One of the most important uses of a `block` is as a target for branch instructions such as `br` and `br_if`. A branch instruction does not jump to an arbitrary program location, but rather to a surrounding structured construct. When a branch instruction targets a `block`, it transfers control to the *end* of that block. A block may optionally declare a result type (e.g.,

(result i32)), in which case the block produces a value that is left on the WASM stack when the block terminates. During compilation of the WASM block instruction, an entry describing the block is pushed onto the control stack. The instructions that follow correspond to the body of the block and are compiled while this entry remains on the stack. The entry provides the information needed to compile branch instructions that target the block, and is popped when the matching end instruction is encountered. The function compile_block() implements this logic. First, the block type is read using read_u8(ctx). If the current region is unreachable, a dummy control entry is pushed to track the structural nesting for the matching end; no CFG nodes are created. If the region is reachable, a new IR block block_end is created as the merge point and a BlockControlStackEntry recording the current block, block_end, and the block type is pushed onto the control stack. block_end is sealed immediately at creation. According to the algorithm of Braun et al. [3], a block may be sealed as soon as no further predecessors will be added to it. Although the CFG edges into block_end are not wired until the body of the block construct has been fully compiled, the invariant required for safe sealing is weaker: it is sufficient that the local variable lookup function read_local() is never called with block_end as input before all of its predecessors have been registered. This invariant holds because read_local() is only called with the current block (ctx.curr_block) as input and block_end never becomes the current block until the matching end instruction is compiled, at which point all predecessors (the fall-through path and any br/br_if instructions targeting the block) have already been recorded. Sealing block_end eagerly avoids allocating the incomplete_phi array.

```
def compile_block(ctx):
    wasm_blocktype = read_u8(ctx)
    if ctx.is_unreachable():
        ctx.push_ctrl_dummy()
        return
    block_end = IRBlock(sealed=True)
    ctx.push_ctrl_block(ctx.curr_block, block_end, wasm_blocktype)
```

4.2.5 Loop

In WebAssembly, a loop defines a structured construct that may be executed repeatedly. The instructions inside the loop are executed sequentially, from the loop instruction to the corresponding end instruction. Unlike traditional

loops in many programming languages, a WebAssembly `loop` does not have an explicit entry condition. Execution always enters the loop body, and repetition is achieved through branch instructions that jump back to the beginning of the loop. The general form is:

```
loop
  instr_1
  instr_2
  ...
  instr_n
end
```

Unlike a `block`, the primary purpose of a `loop` is to serve as a target for backward branching. When a branch instruction such as `br` or `br_if` targets a loop, control is transferred to the *beginning* of the loop rather than to its end. It is worth noting that a `loop` construct does not itself guarantee a cycle in the CFG: a cycle only arises if the loop body contains at least one `br` or `br_if` that explicitly branches back to the loop start. A loop may optionally declare a result type (e.g., `(result i32)`), indicating a value produced when execution falls through to the `end` without branching back. However, since `br` to a loop always targets its start (not its exit), no value is passed on a back-edge; the `label_type` of a loop control entry is therefore always `WASM_BLOCKTYPE_NONE`. The function `compile_loop()` implements the compilation of the WASM `loop` instruction. First, the block type is read using `read_u8(ctx)`. If the current region is unreachable, a dummy control entry is pushed and the function returns. Otherwise, a new IR block `loop_start` is created, an unconditional jump from the current block to `loop_start` is emitted, `loop_start` becomes the current block, and a `LoopControlStackEntry` is pushed. Crucially, `loop_start` is *not* sealed at creation time. Because it becomes `ctx.curr_block` immediately, `read_local()` may be called on it while the loop body is being compiled — before the back-edges from any `br`/`br_if` instructions inside the body have been registered. Sealing `loop_start` at this point would violate the invariant of Braun et al. [3]: a variable look-up on a block with an incomplete predecessor set would miss definitions flowing in along back-edges, producing incorrect ϕ-functions. The block is therefore left unsealed (allocating an `incomplete_phi` array for proxy ϕ-functions) and is sealed by `compile_end_loop()` once all back-edges are known. It is sealed by calling `seal_block()` in `compile_end_loop()`, once all back-edges have been recorded.

```
def compile_loop(ctx):
    wasm_blocktype = read_u8(ctx)
    if ctx.is_unreachable():
        ctx.push_ctrl_dummy()
        return
    loop_start = IRBlock()   # not sealed: back-edges not yet known
    ctx.curr_block.jmp(loop_start)
    ctx.curr_block = loop_start
    ctx.push_ctrl_loop(loop_start, wasm_blocktype)
```

4.2.6 If

In WebAssembly, the `if` instruction defines a structured conditional construct. The condition value is taken from the top of the WASM stack: if it is non-zero, execution continues with the *then* branch, otherwise execution continues with the *else* branch (or falls through to `end` if no `else` is present). The general form is:

`if`
  $instr_1$
  …
  $instr_n$
`else`
  $instr_{n+1}$
  …
  $instr_{n+m}$
`end`

The `else` branch is optional. If it is absent, execution continues after the `end` instruction when the condition is zero. A branch instruction such as `br` or `br_if` targeting an `if` construct transfers control to the *end* of the construct. Like other structured constructs, `if` may declare a result type (e.g., `(result i32)`), indicating a value left on the stack at the corresponding `end`. The function `compile_if()` implements the compilation of the WASM `if` instruction. First, the block type is read using `read_u8(ctx)`. If the current region is unreachable, a dummy control entry is pushed and the function returns. Otherwise, the condition is popped from the operand stack. Two new sealed IR blocks are created: `then_branch` (the entry of the *then* branch) and `if_end` (the merge point). A conditional jump `jnz(cond, then_branch, None)` is emitted from the current block; the false successor is set to `None` for now, because it is unknown whether an `else` will

follow — it is patched to `else_branch` by `compile_else()`, or directly to `if_end` by `compile_end_if()` if there is no `else`. An `IfControlStackEntry` recording the current block, `if_end`, and the block type is pushed, and `then_branch` becomes the current block. Both new blocks are sealed immediately. `then_branch` has exactly one predecessor — the current block, wired via the conditional jump just emitted — so its predecessor set is complete at creation time. `if_end` satisfies the same invariant as `block_end`: it never becomes the current block until the matching `end` is compiled, at which point all predecessors have already been registered.

```
def compile_if(ctx):
    wasm_blocktype = read_u8(ctx)
    if ctx.is_unreachable():
        ctx.push_ctrl_dummy()
        return
    cond = ctx.pop_opd()
    then_branch = IRBlock(sealed=True)
    if_end = IRBlock(sealed=True)
    ctx.curr_block.jnz(cond, then_branch, None)  # false successor
    ↪ patched later
    ctx.push_ctrl_if(ctx.curr_block, if_end, wasm_blocktype)
    ctx.curr_block = then_branch
```

4.2.7 Else

The `else` instruction marks the boundary between the *then* and *else* branches of a previously opened `if` construct. When it is encountered, the compiler has finished emitting code for the *then* branch and must now begin the *else* branch. The function `compile_else()` implements this. If the top control entry is a `DummyControlStackEntry`, the matching `if` was in an unreachable region and no CFG work is needed; the function returns immediately. Otherwise, the `if` control entry is popped. If the *then* branch is reachable, an unconditional jump to the merge block (`if_ctrl.label_block`) is emitted so that the *then* branch does not fall through into the *else* branch. A new block `else_branch` is created and sealed immediately: its sole predecessor is the `if`-condition block, which is wired to it on the very next line by patching `if_ctrl.start_block.succ1`. Because that predecessor is registered before `else_branch` ever becomes the current block, the Braun et al. invariant holds and no `incomplete_phi` array is needed. Finally, an `ElseControlStackEntry` is pushed — inheriting the merge block, types,

stack height, and accumulated `end_results` from the `if` entry — and the current block is set to `else_branch`.

```
def compile_else(ctx):
    if isinstance(ctx.top_ctrl(), DummyControlStackEntry): return
    if_ctrl = ctx.pop_ctrl()
    if not if_ctrl.unreachable:
        ctx.curr_block.jmp(if_ctrl.label_block)
    else_branch = IRBlock(sealed=True)
    if_ctrl.start_block.succ1 = else_branch  # patch the false
    ↪  successor
    ctx.push_ctrl_else(if_ctrl)
    ctx.curr_block = else_branch
```

4.2.8 Branch

In WebAssembly, the `br` instruction performs an unconditional branch to a surrounding structured control construct. Instead of jumping to an arbitrary program location, WebAssembly restricts branches to target enclosing constructs such as `block`, `loop`, or `if`. The branch target is identified by a label index specifying how many nested constructs must be exited to reach the destination: index `0` refers to the innermost enclosing construct, `1` to the next outer, and so on. The exact control-flow behavior depends on the type of the targeted construct: for `block` and `if`, the branch transfers control to the *end* of the construct; for `loop`, it transfers control to the *beginning* (the loop start). If the targeted construct declares a result type, the value at the top of the operand stack is consumed and passed to the construct's merge point. The function `compile_br()` implements this. First, the label index is read using `readULEB128_u32()`. If the current context is unreachable, the function returns immediately. Otherwise, the control entry for the target label is retrieved using `ctx.get_ctrl(labelidx)`. If the target construct declares a result type (`label_type` $\neq$ `WASM_BLOCKTYPE_NONE`), the top operand is popped and the pair (*value*, *current block*) is appended to the control entry's `end_results` list. This record is later consumed by `get_block_result()` to build the ϕ-function at the merge point. An unconditional jump to the target label block is emitted. Since `br` is unconditional, no further instructions can execute in the current block; the context is therefore marked unreachable via `ctx.unreachable()`.

```
def compile_br(ctx):
    labelidx = readULEB128_u32()
    if ctx.is_unreachable(): return
    ctrl = ctx.get_ctrl(labelidx)
    if ctrl.label_type != WASM_BLOCKTYPE_NONE:
        opd = ctx.pop_opd()
        ctrl.end_results.append((opd, ctx.curr_block))
    ctx.curr_block.jmp(ctrl.label_block)
    ctx.unreachable()
```

4.2.9 Branch If

The `br_if` instruction implements a conditional branch. Like `br`, it targets a surrounding structured construct identified by a label index. However, the branch is only taken if a condition value on the top of the WASM stack is non-zero; if the condition is zero, execution continues with the next instruction. If the targeted construct declares a result type, the value below the condition on the stack is passed to the construct's merge point when the branch is taken. The function `compile_br_if()` implements this. First, the label index is read using `readULEB128_u32()`. If the current context is unreachable, the function returns immediately. Otherwise, the condition is popped from the operand stack, and the control entry for the target label is retrieved using `ctx.get_ctrl(labelidx)`. If the target construct declares a result type, the value currently on top of the operand stack (the potential branch result) is *peeked* using `ctx.top_opd()` rather than popped, because it must remain on the stack for the fall-through path. It is recorded together with the current block in the control entry's `end_results` list. A new sealed IR block named `continue_block` is created to represent the fall-through path when the condition is zero. A conditional jump is emitted from the current block: if the condition is non-zero, control goes to the target label block; otherwise, execution continues at `continue_block`. The compilation context is updated so that `continue_block` becomes the current block.

```
def compile_br_if(ctx):
    labelidx = readULEB128_u32()
    if ctx.is_unreachable(): return
    cond = ctx.pop_opd()
    ctrl = ctx.get_ctrl(labelidx)
    if ctrl.label_type != WASM_BLOCKTYPE_NONE:
        opd = ctx.top_opd()
        ctrl.end_results.append((opd, ctx.curr_block))
    continue_block = IRBlock(sealed=True)
    ctx.curr_block.jnz(cond, ctrl.label_block, continue_block)
    ctx.curr_block = continue_block
```

4.2.10 End

The `end` instruction in WebAssembly marks the termination of a structured control construct such as `block`, `loop`, or `if`. It signals the compiler that the current control entry is complete and that execution should continue at the merge point of the construct. The function `compile_end()` pops the top control entry using `ctx.pop_ctrl()` and delegates to a specialized helper based on the entry type:

- `DummyControlStackEntry` — the matching construct was skipped because the region was unreachable. No CFG updates are performed.
- `IfControlStackEntry` — end of an `if` without a matching `else`. Delegates to `compile_end_if()`.
- `ElseControlStackEntry` — end of an `else` branch. Delegates to `compile_end_block()`.
- `BlockControlStackEntry` — end of a `block` construct. Delegates to `compile_end_block()`.
- `LoopControlStackEntry` — end of a `loop` construct. Delegates to `compile_end_loop()`.

Each helper connects the current block to the merge block and pushes any result value onto the operand stack.

```
def compile_end(ctx):
    ctrl = ctx.pop_ctrl()
    match ctrl:
        case DummyControlStackEntry:
            return
        case IfControlStackEntry:
            compile_end_if(ctx, ctrl)
        case ElseControlStackEntry:
            # compile_end_else() is equivalent to
            ↪  compile_end_block()
            compile_end_block(ctx, ctrl)
        case BlockControlStackEntry:
            compile_end_block(ctx, ctrl)
        case LoopControlStackEntry:
            compile_end_loop(ctx, ctrl)
```

End of If

`compile_end_if()` handles the termination of an `if` construct without a matching `else`. If the end of the *then* branch is reachable, an unconditional jump is emitted to the merge block `if_end`. The compilation context is updated so that `if_end` becomes the current block. The false successor of the conditional jump emitted during `compile_if()` — which was left as `None` because no `else` was present — is now patched to point directly to `if_end`. If the construct declares a result type, `get_block_result()` is called to obtain the merged result, which is then pushed onto the operand stack.

```
def compile_end_if(ctx, ctrl):
    if_start = ctrl.start_block
    if_end = ctrl.label_block
    wasm_blocktype = ctrl.end_type
    if not ctrl.unreachable:
        ctx.curr_block.jmp(if_end)
    ctx.curr_block = if_end
    if_start.succ1 = if_end  # patch the false successor
    if wasm_blocktype != WASM_BLOCKTYPE_NONE:
        result = get_block_result(ctx, ctrl)
        ctx.push_opd(result)
```

End of Block

`compile_end_block()` handles the termination of a `block` construct or the end of an `else` branch. If the current region is reachable, an unconditional jump is emitted to the merge block `block_end`. The compilation context is updated so that `block_end` becomes the current block. If the construct declares a result type, the merged result is obtained via `get_block_result()` and pushed onto the operand stack.

```
def compile_end_block(ctx, ctrl):
    block_end = ctrl.label_block
    wasm_blocktype = ctrl.end_type
    if not ctrl.unreachable:
        ctx.curr_block.jmp(block_end)
    ctx.curr_block = block_end
    if wasm_blocktype != WASM_BLOCKTYPE_NONE:
        result = get_block_result(ctx, ctrl)
        ctx.push_opd(result)
```

End of Loop

`compile_end_loop()` finalises a loop construct. By this point all predecessors of `loop_start` — including all back-edges from `br` and `br_if` instructions inside the body — have been registered, so `seal_block()` is called to complete any proxy ϕ-functions left in `incomplete_phi`. If the construct declares a result type, the merged result is obtained via `get_block_result()` and pushed onto the operand stack. If the loop body ended with an unconditional branch (i.e., `ctrl.unreachable` is true), execution never fell through to the `end`; the code following the loop is therefore dead, and `ctx.unreachable()` is called accordingly.

```
def compile_end_loop(ctx, ctrl):
    loop_header = ctrl.label_block
    wasm_blocktype = ctrl.end_type
    seal_block(loop_header)  # all predecessors now known
    if wasm_blocktype != WASM_BLOCKTYPE_NONE:
        result = get_block_result(ctx, ctrl)
        ctx.push_opd(result)
    if ctrl.unreachable:
        ctx.unreachable()
```

Handling Block Results

The function `get_block_result()` consolidates the values contributed by each control-flow path that reaches a construct's merge block. The `end_results` list holds one (*value*, *predecessor block*) pair per contributing path, accumulated by `pop_ctrl()`, `compile_br()`, and `compile_br_if()`. Three cases are distinguished:

- **One entry**: the single value is returned directly; no ϕ-function is needed.
- **More than one entry**: a fresh IR variable and an `IRPhi` node are created; each pair becomes an argument of the ϕ-function, which is prepended to the merge block.
- **Empty**: every path through the construct exited via an unconditional branch, so the fall-through is dead code. A placeholder operand is returned; its value is irrelevant, but it must be pushed so that compilation of the surrounding (unreachable) code can continue without error.

```
def get_block_result(ctx, ctrl):
    n = len(ctrl.end_results)
    if n == 1:
        opd, block = ctrl.end_results[0]
        return opd
    elif n > 1:
        result = IRTmpReference()
        phi = IRPhi(result)
        for opd, block in ctrl.end_results:
            phi.args.append(opd, block)
        ctx.curr_block.phi_list.append(phi)
        return result
    else:
        # No values were contributed: all paths exited via
        ↪  unconditional
        # branches, so the code after this construct is dead.
        # Return a placeholder; its value is irrelevant.
        return default_opd()
```

4.2.11 Local Get

The `local.get` instruction reads the current value of a local variable and pushes it onto the WASM stack. Its immediate operand is a *local index* that identifies which local variable to read. WASM local variables include both the function parameters and any additional locals declared at the top of the function body. The function `compile_local_get()` implements the compilation of this instruction. The local index is read using `readULEB128_u32()`. If the current region is unreachable, the instruction is ignored. Otherwise, `read_local()` is called on the current block and the local index. As described in Section 4.1.2 (`locals` and `incomplete_phi`), `read_local()` first checks `curr_block.locals[localidx]` (local value numbering). If no definition is present, it searches backward through the predecessor graph, inserting a ϕ-function if multiple predecessors carry different definitions, and leaving an operandless proxy in `incomplete_phi[localidx]` if the block is not yet sealed. The result is memoised in `locals[localidx]` and pushed onto the operand stack.

```
def compile_local_get(ctx):
    localidx = readULEB128_u32()
    if ctx.is_unreachable(): return
    local = read_local(ctx.curr_block, localidx)
    ctx.push_opd(local)
```

4.2.12 Local Set

The `local.set` instruction pops the top value from the WASM stack and stores it into a local variable. Its immediate operand is the local index identifying the variable to be written. The function `compile_local_set()` implements this. Unlike `local.get`, which may trigger a backward search through the CFG, `local.set` is a purely local operation: it records a new definition for the variable in the current block only. First, the local index is read using `readULEB128_u32()`. If the current region is unreachable, the instruction is ignored. Otherwise, the top value is popped from the operand stack. `write_local()` is then called to record this IR reference as the current definition of local variable `localidx` in the current block, by storing it in `curr_block.locals[localidx]`. This overwrites any previous definition held in the `locals` array for this variable in this block; subsequent `local.get` instructions for the same variable in the same block will find this new definition directly via local value numbering, without any backward search. No

IR instruction is emitted: `local.set` is a bookkeeping operation from the compiler's perspective, not a computational one.

```
def compile_local_set(ctx):
    localidx = readULEB128_u32()
    if ctx.is_unreachable(): return
    opd = ctx.pop_opd()
    write_local(curr_block, localidx, opd)
```

4.2.13 Load

WASM provides a family of load instructions that read a value from linear memory and push it onto the WASM stack. For example, the instruction `i32.load` reads a 32-bit integer from the linear memory. All load instructions share the same structure: they encode two immediates — an *alignment hint* and a *static offset* — and consume one `i32` value from the WASM stack that serves as the dynamic base address. The effective byte address is the sum of the static offset and the dynamic base address. After reading the two immediates and returning early if the region is unreachable, the dynamic base address is popped from the operand stack. An IR `add` instruction is emitted to compute the effective pointer `ptr = mem0 + base` at runtime, followed by the load instruction that reads from `ptr` at the static `offset`. A fresh IR temporary `destination` receives the loaded value and is pushed onto the operand stack.

```
def compile_load(ctx, ir_opcode):
    align  = readULEB128_u32()   # alignment hint, currently unused
    offset = readULEB128_u32()   # static offset immediate
    if ctx.is_unreachable(): return
    base = ctx.pop_opd()
    destination = IRTmpReference()
    ptr = IRTmpReference()
    emit(IR_ADD, ptr, mem0, base)
    emit(IR_LOAD, destination, IR_I32(offset), ptr)
    ctx.push_opd(destination)
```

4.2.14 Store

WASM provides a symmetric family of store instructions that pop a value from the WASM stack and write it to linear memory. For example, the

instruction `i32.store` writes a 32-bit integer. Like load instructions, store instructions encode an alignment hint and a static offset as immediates. They consume two values from the WASM stack: first the *value* to be written (on top), and below it the *dynamic base address*. The effective byte address is computed the same way as for loads. The function `compile_store()` implements the compilation of all store instructions. The alignment hint and the static offset are read first. If the current region is unreachable, the instruction is ignored. Otherwise, the value to be stored is popped first, followed by the dynamic base address. If the base address is an IR `add` is emitted to compute `ptr = mem0 + base` at runtime, and the IR store uses `ptr` as the pointer and the static `offset` as the offset. Unlike load instructions, no result is pushed onto the operand stack: store instructions are side-effecting operations that produce no value.

```
def compile_store(ctx, ir_opcode):
    align  = readULEB128_u32()   # alignment hint, currently unused
    offset = readULEB128_u32()   # static offset immediate
    if ctx.is_unreachable(): return
    value = ctx.pop_opd()
    base  = ctx.pop_opd()
    ptr = IRTmpReference()
    emit(IR_ADD, ptr, ctx.mem0, base)
    emit(IR_STORE, value, IR_I32(offset), ptr)
```

4.2.15 Integer Constant

The `i32.const` instruction pushes a 32-bit integer literal onto the WASM stack. Its single immediate operand is a signed 32-bit integer encoded as a signed LEB128 value in the WASM binary. The function `compile_i32const()` implements the compilation of this instruction. The immediate value is read using `readILEB128_i32()`. If the current region is unreachable, the instruction is ignored. Otherwise, no IR instruction is emitted: the constant is pushed directly onto the operand stack. To support this behavior, the operand stack must be generalized so that it can store both IR variables and constant values.

```
def compile_i32const(ctx):
    n = readILEB128_i32()
    if ctx.is_unreachable():
        return
    ctx.push_opd(n)
```

Chapter 5

Implementation Details of the `wasmc` Back End

This chapter describes the implementation of the `wasmc` back end, which takes a CFG in SSA form as input and produces native 32-bit RISC-V machine code. The back end is organized as a pipeline of four phases, each described in a dedicated section. Section 5.1 covers **register allocation**. `wasmc` implements the *linear scan* algorithm of Poletto and Sarkar [19], adapted to operate on a CFG in SSA form. Section 5.2 covers **ABI constraint enforcement**, which ensures that the generated code follows the 32-bit RISC-V calling conventions and register-usage rules. Section 5.3 covers **SSA deconstruction**. `wasmc` eliminates ϕ-functions using *parallel move serialization* [23] and inserts the resulting moves along CFG edges with *critical-edge splitting* [4, pp. 20–24]. Section 5.4 covers **instruction selection**. `wasmc` uses the *naive macro expansion* approach [2, Secs. 2.1 and 2.2].

5.1 Register Allocation

The role of a register allocator is to take an IR that uses an arbitrary number of variables and produce a map that assigns to each variable either a physical register or a position on the stack frame. This map is used later to rewrite the IR to use a finite number of physical registers and stack slots. `wasmc` implements the *linear scan* register allocator algorithm directly from the article that first introduced it, by Poletto and Sarkar [19] [1], adapted to work on a

[1]There are many variants of the linear scan algorithm created after the Poletto and Sarkar article. In the literature, the name "linear scan" usually refers to the entire family of register allocation algorithms inspired by that original work.

CFG in SSA form. The Poletto and Sarkar linear scan is a global[2] register allocation algorithm that is not based on graph coloring[3]. Rather, given the live intervals of variables in the IR of a function, the algorithm scans all live intervals in a single pass, allocating registers to variables greedily and respecting this rule: two variables cannot be assigned to the same register if their live intervals overlap. The name "linear scan" derives from this single scan over the live intervals. A *live interval* of a variable is a pair of integers $[start, end)$ (with *end* exclusive) that begins when the variable is first defined and ends when it is last used. This requires that the order of instructions in the IR has already been fixed into a single linear sequence. To fix this sequence, the CFG is first linearized into an ordered list of basic blocks; then the list is scanned and an increasing sequence number is assigned to each instruction. So the `wasmc` register allocator takes a CFG in SSA form as input, linearizes it into a list of basic blocks, numbers the instructions, constructs the live intervals, applies the Poletto and Sarkar linear scan algorithm, and finally produces a map assigning to each variable a register or stack slot such that no two variables with overlapping live ranges share the same register. The register hints step assigns preferred registers to live intervals to guide the linear scan allocator and reduce move instructions introduced by ABI constraints and SSA deconstruction. The `wasmc` register allocator can be broken down into the steps shown in Figure 5.1. The following sections explain the implementation details of each step.

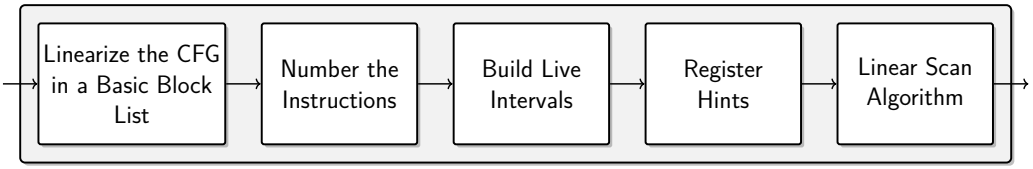


Figure 5.1: The `wasmc` register allocator steps.

[2]Register allocation algorithms are divided into local and global. Global register allocation algorithms analyze the function's CFG, extract information that spans multiple basic blocks, and use it to perform register allocation. Local algorithms instead allocate registers one basic block at a time, making locally optimal choices. Local algorithms are largely historical at this point. See [7, Ch. 13] for a longer explanation.

[3]Graph coloring is a register allocation framework in which registers are the colors and the variables of the function's IR are the nodes of a graph. Two nodes are connected if the respective variables cannot be assigned to the same register; the goal is then to color the graph. Graph coloring register allocation produces better register assignments than linear scan but requires more memory and is more computationally expensive. Since one of the goals of `wasmc` is to run on resource-constrained devices, a linear scan algorithm is used instead. See [7, Ch. 13] for a longer explanation.

5.1.1 Linearize the CFG in a Basic Block List

The CFG in SSA form is linearized into a ordered list of basic blocks. The ordering must satisfy the following two properties:

- All predecessors of a block are located before it, with the exception of backward edges from loop ends to loop starts.
- All blocks that are part of the same loop are contiguous, i.e., no non-loop block appears between two loop blocks.

In general, the choice of basic block ordering is very important in a compiler back end: it has a high impact on the quality of linear scan and, later, on instruction selection. Basic block ordering is a heuristic: a good or bad ordering can greatly increase or decrease the performance of the final generated code. As indicated in [42, Sec. 4], these two properties guarantee a good heuristic. Additionally, they are necessary for the correctness of the live interval construction algorithm [42, Sec. 4]. Figure 5.2 shows an example: the list of basic blocks obtained by linearizing the CFG in SSA form of figure 2.2.

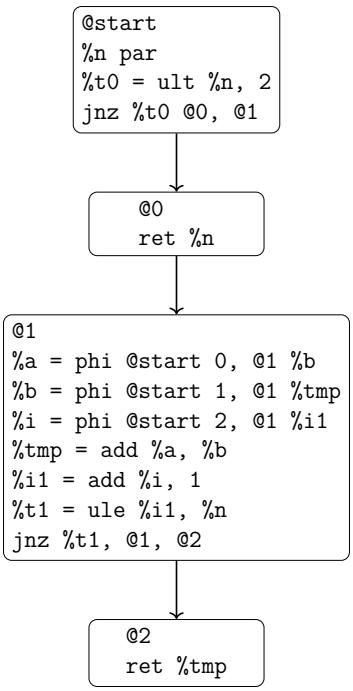


Figure 5.2: List of basic blocks obtained by linearizing the CFG of figure 2.2.

5.1.2 Numbering the Instructions

The ordered list of basic blocks produced by the previous step is traversed to assign a sequence number to every instruction. Each IR instruction stores its sequence number in the `seqnum` field. The numbering procedure, shown in Figure 5.3, is a straightforward iteration over the block list and the instructions within each block. Note that the jump instruction terminating each block also receives a sequence number, which is important for correctly computing live intervals at block boundaries.

```
def number_instructions(f: IRFunction):
    next_seqnum = 0
    # For each basic block
    for block in f.block_list:
        block.seqnum = next_seqnum
        next_seqnum += 1
        # For each instruction inside the basic block
        for ins in block.instr_list:
            ins.seqnum = next_seqnum
            next_seqnum += 1
        # The jump terminating the block also gets a sequence
        ↪ number
        block.jump.seqnum = next_seqnum
        next_seqnum += 1
```

Figure 5.3: Numbering of instructions.

Sequence numbers are not associated with ϕ-functions individually; instead, all ϕ-functions within a block b share the sequence number of block b itself. This choice is also made in [42, Sec. 3] and is justified by the following reasons. The sequence number associated with an instruction is used to build live intervals, and the live interval for a variable defined by a ϕ-function must start directly at the beginning of the block in which the ϕ-function resides, even when multiple ϕ-functions appear at the start of a block (this follows from the parallel move semantics of ϕ-functions, see section 2.5). In addition, the live intervals for variables *used* by a ϕ-function end at the end of the corresponding predecessor block (as long as the variable is not used by any other instruction after the ϕ-function). As an example, Figure 5.4 shows the instruction linear sequence obtained by numbering the basic block list of Figure 5.2; all ϕ-functions inside block `@1` share sequence number 6.

```
 0: @start
 1: %n par
 2: %t0 = ult %n, 2
 3: jnz %t0 @0, @1
 4: @0
 5: ret %n
 6: @1
    %a = phi @start 0, @1 %b
    %b = phi @start 1, @1 %tmp
    %i = phi @start 2, @1 %i1
 7: %tmp = add %a, %b
 8: %i1 = add %i, 1
 9: %t1 = ule %i1, %n
10: jnz %t1, @1, @2
11: @2
12: ret %tmp
```

Figure 5.4: Instruction linear sequence obtained by numbering the basic block list of figure 5.2.

5.1.3 Build Live Intervals

`wasmc` constructs live intervals following the `BuildIntervals` algorithm of Wimmer and Franz [42, Sec. 4], with one simplification: whereas Wimmer and Franz represent each live interval as a union of multiple disjoint ranges (e.g. $I = [a_1, b_1) \cup [a_2, b_2) \cup \cdots \cup [a_n, b_n)$), `wasmc` uses a single contiguous range $I = [a, b)$ per variable. Single-range intervals were used in the original linear scan article by Poletto and Sarkar. The field later moved to multiple disjoint ranges [28], which can free up registers in the context of branches and achieve better code quality. The only changes relative to the `BuildIntervals` function of [42, Sec. 4] are the implementations of the `addRange` and `setFrom` helper functions.

```
class LiveInterval:
    def __init__(self):
        self.start = UINT_MAX
        self.end = 0

    def addRange(self, a: int, b: int):
        # Extend the interval to be the smallest range containing
        # the union of [self.start, self.end) and [a, b).
        assert a < b
        if a < self.start:
            self.start = a
        if b > self.end:
            self.end = b

    def setFrom(self, a: int):
        if self.end == 0:
            # This happens when a variable is defined but never
            ↪ used.
            self.start = a
            self.end = a
        else:
            # If a valid range already exists, this may shrink it.
            assert a < self.end
            self.start = a
```

Figure 5.5 continues the example of Figure 5.4. It shows the live intervals computed by the `BuildIntervals` function applied to the linear sequence of Figure 5.4.

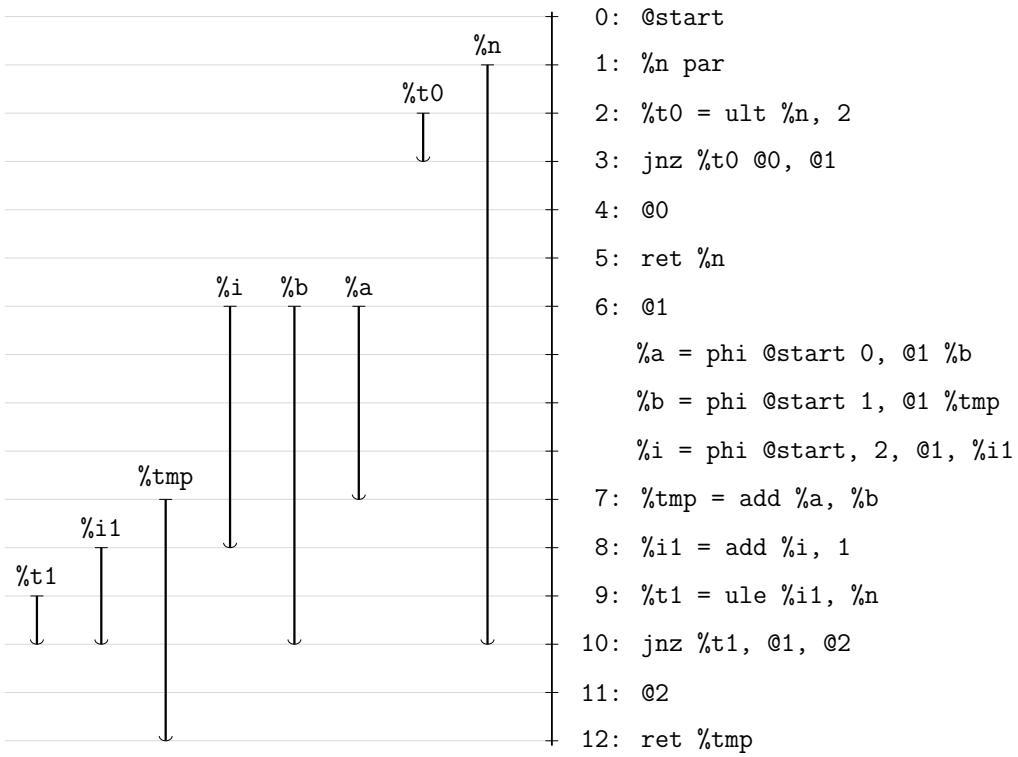


Figure 5.5: Live intervals computed by the `BuildIntervals` function applied to the code of figure 5.2.

This section concludes with a note on how SSA form is exploited inside `BuildIntervals`. The standard way of constructing live intervals on a CFG, not necessarily in SSA form, requires liveness analysis as a subroutine, which is traditionally performed using the iterative data flow analysis framework. By exploiting properties guaranteed by SSA form in combination with the basic block ordering, `BuildIntervals` eliminates the need for iterative data flow analysis entirely. Live interval construction is the only part of the back end that exploits the SSA form, and the `BuildIntervals` function is more time-efficient than interval construction based on iterative liveness analysis.

5.1.4 Register Hints

The register hints step populates an optional `register_hint` field on each live interval. When the linear scan algorithm must assign a register to a live interval, it first checks whether the hinted register is available (i.e., not currently in use); if so, it assigns that register directly. If the hint is not

available, the algorithm falls back to picking the next free register from the registers pool. Register hints are a simple but effective optimization: they reduce the number of move instructions inserted later by ABI constraint enforcement and SSA deconstruction, because variables that are ultimately required to reside in a specific register are more likely to already be there. `wasmc` computes the following hints:

- **Function parameters:** Each `par` instruction at the start of a function is hinted toward the corresponding ABI argument register (`a0` for the first parameter, `a1` for the second, and so on). This reduces the moves needed to satisfy the calling convention when the function is entered.
- **Call arguments:** Each `arg` instruction preceding a function call is hinted toward the ABI argument register that the corresponding call argument must eventually occupy (`a0` for the first argument, `a1` for the second, etc.).
- **Call return values:** If a `call` instruction has an output variable, that variable is hinted toward `a0`, since the callee places its return value there.
- **Function return values:** Any variable used by a `ret` instruction is hinted toward `a0`, since the return value must be placed in that register.
- **Phi inputs:** If a ϕ-function's output variable already has a hint, that same hint is propagated to the input variables of the ϕ-function. This helps coalesce the live intervals involved in a ϕ-function into the same register, reducing the moves generated during SSA deconstruction.

5.1.5 Linear Scan Algorithm

`wasmc` implements the linear scan algorithm of Poletto and Sarkar [19] with one small change regarding live interval endpoints. In the Poletto and Sarkar original work, live intervals are treated as closed intervals $[from, to]$; many later implementations and papers instead use half-open intervals $[start, end)$. The reason for this change concerns endpoint semantics: the real question is, "do two intervals interfere if one ends exactly where the other begins?" Suppose closed intervals are used, and variable v_1 is live from instruction 5 to 10 ($v_1 = [5, 10]$) while v_2 is live from instruction 10 to 15 ($v_2 = [10, 15]$). They overlap at point 10, so there is a conflict, and v_1 and v_2 must be assigned different registers. If instruction 10 is v_2 `= addi` v_1`, 10`, we cannot rewrite it as `reg1 = addi reg1, 10` because closed intervals force the algorithm to

assign different registers to v_1 and v_2. Yet semantically this is wrong: at instruction 10, v_1 is used and v_2 is defined, and after instruction 10 finishes, v_1 is dead and v_2 becomes live. They never need a register at the same time. Closed intervals therefore introduce an artificial interference. Half-open intervals fix this problem. With $v_1 = [5, 10)$ and $v_2 = [10, 15)$, v_1 is live up to but not including instruction 10, and v_2 starts at instruction 10. They do not overlap, so the algorithm is free to assign the same register `reg1` to both. Additionally, when using half-open intervals the correct expiry condition in linear scan is: expire interval `i` if `i.end` $\leq$ `current.start`. Another important implementation detail is the register assignment heuristic. Poletto and Sarkar do not specify how to choose a register from the pool of free registers when assigning one to a live interval. `wasmc` uses the heuristic shown in Figure 5.6: if the interval carries a register hint and that register is free, it is used; otherwise the first free register in registers pool is chosen.

```
def assign_register(live_int: LiveInterval, unused: RegisterPool):
    # The 'register_hint' field is set during the Register Hints
    ↪  step.
    hint = live_int.register_hint
    if hint is not None and unused.contains(hint):
        register = hint
    else:
        # Pick the first free register.
        register = unused.get()
    live_int.assigned_location = register
    unused.remove(register)
```

Figure 5.6: Register assignment heuristic.

Figure 5.7 continues the example of Figure 5.5. It shows the register assigned to each variable by the `wasmc` implementation of the Poletto and Sarkar linear scan algorithm. Variables with overlapping intervals are not assigned to the same register. In this specific case there are no spilled[4] variables.

[4]"Spill" in the context of register allocation means moving a variable out of a register and into a stack slot because there are not enough registers available.

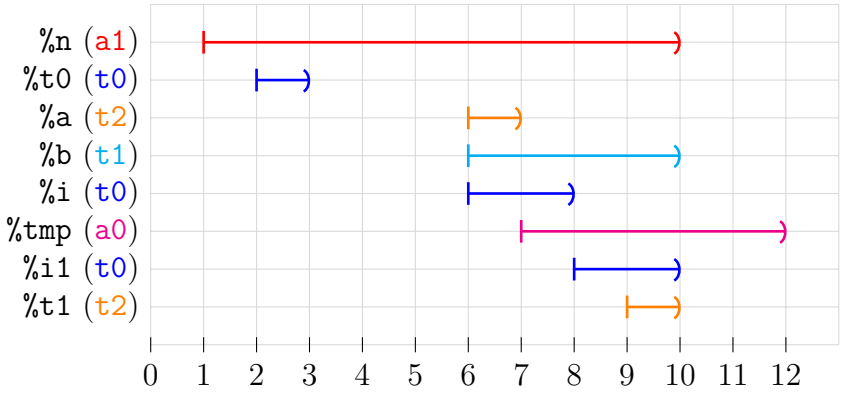


Figure 5.7: Output of the `wasmc` linear scan algorithm on the live intervals of Figure 5.5. On the y-axis, each variable is shown alongside its assigned register in parentheses.

This section concludes with a list of important implementation details of the Poletto and Sarkar linear scan algorithm that are not prominently highlighted in [19]:

- The algorithm cannot deal with lifetime holes and does not split intervals. Therefore, a variable either holds a register for its entire lifetime or is spilled entirely. Poletto and Sarkar linear scan assigns exactly one location (register or stack slot) per variable for the whole program: if a variable is spilled, it lives on the stack from the start to the end of its interval and is never placed in a register.
- It is not possible to implement the algorithm without reserving two scratch registers. When a spilled variable is used by an instruction that requires its operand to be in a register, the variable must be temporarily reloaded into a scratch register.
- ABI constraints must be handled separately, after register allocation.

5.2 ABI Constraints Enforcement

The ABI constraints for 32-bit `RISC-V` [12] can be summarized as follows (floating-point register constraints are ignored because `wasmc` does not currently implement WebAssembly `f32` and `f64`; see appendix A for the implementation status of `wasmc` relative to the WebAssembly specification):

- **Function arguments:** the caller must place function arguments in `a0`–`a7`, so the callee receives them through those registers. `a0` holds

the first argument and `a7` holds the eighth; arguments beyond eight are passed on the stack.

- **Return value:** the callee must place the return value in `a0`, so the caller receives it there. For a 64-bit return value, the pair `a0`+`a1` is used.
- **Special registers:** `ra` (also called `x1`), `sp` (also called `x2`), and `fp` (also called `s0` or `x8`). `ra` holds the return address after a function call. `sp` is the stack pointer. `fp` is the frame pointer.
- **Caller-saved registers**[5]: the value of a caller-saved register does not need to be preserved across function calls; the caller must save and restore it if it wishes to keep the value across a call. The 32-bit RISC-V caller-saved registers are `ra`, `t0`–`t6`, and `a0`–`a7`.
- **Callee-saved registers**[6]: the value of a callee-saved register must be preserved across function calls; if a callee-saved register is modified during a function, its original value must be restored before the function returns. The 32-bit RISC-V callee-saved registers are `s0`–`s11`.
- The stack must be 16-byte aligned.

The handling of special registers, callee-saved registers, and stack alignment takes place during instruction selection; see section 5.4. The remaining ABI constraints are handled during this ABI constraint enforcement phase. The goals of this phase are:

- Ensure that the function arguments are correctly received from the ABI argument registers (this is trickier than it might seem, as explained below in Section 5.2.1).
- Insert push and pop instructions around call instructions to preserve variables assigned to caller-saved registers that are live across the call. Only variables stored in physical registers need to be preserved; variables already on the stack do not. Variables assigned to `t0`–`t6` or `a0`–`a7` are saved and restored using push/pop around calls, while `ra` is saved in the function prologue (see instruction selection, Section 5.4).

[5]Caller-saved registers are also called temporary, scratch, volatile, or clobbered registers.

[6]Callee-saved registers are also called non-volatile, preserved, call-preserved, or saved registers.

- Populate call arguments into the ABI argument registers (again, trickier than it might seem, see Section 5.2.2).
- Ensure that the return value of a function call, which the callee places in `a0`, is moved to the location assigned by the register allocator to the output variable of the call instruction. This can be done with a simple move or load instruction.
- Ensure that every return instruction places the return value into the ABI return register `a0`. This can be done with a simple move or load instruction.

5.2.1 Receiving Function Arguments

A function must receive its arguments from the caller through argument registers `a0`–`a7`. In an IR function, function parameters are IR variables like any other, so the register allocator is free to assign them to any register—not necessarily the argument registers. A naive approach for satisfying this ABI constraint is: for each IR function parameter in order, emit a move from `a0` to the register assigned to the first parameter, from `a1` to the register assigned to the second, and so on. This sounds reasonable but is incorrect in general. Consider an IR function with two parameters `%x par` and `%y par`, where the register allocator has assigned `a1` to `%x` and `a0` to `%y`. The naive approach would generate:

```
mv a1, a0
mv a0, a1
```

This is wrong: after these two moves, both `a0` and `a1` hold the original value of `a0` (i.e., `x`), and the value of `y` has been lost. The correct approach is to recognize that copying function parameters from the argument registers has *parallel move* semantics. To correctly receive function arguments, a parallel move is constructed with the ABI argument registers (`a0`, …, `a7`) on the right-hand side and the locations assigned by the register allocator to the IR function parameters on the left-hand side. The parallel move is then serialized and the resulting sequence of single assignments is inserted at the start of the function.

5.2.2 Populating Argument Registers for Calls

The same issue arises when preparing arguments for a function call. Before a call, the ABI requires that the arguments be placed in registers `a0`–`a7`.

Since the register allocator may assign the argument variables to arbitrary locations, moving them sequentially into the argument registers may again overwrite values that are still needed. For instance, when compiling the call `f(a, b)`, suppose the register allocator has assigned `a1` to `a` and `a0` to `b`. A naive implementation would generate:

```
mv a0, a1
mv a1, a0
call f
```

This again loses one of the values for the same reason as before. As with receiving arguments, the operation must therefore be treated as a *parallel move*. In this case, the ABI argument registers (`a0`, …, `a7`) are the left-hand side of the parallel move, and the locations assigned to the call arguments are the right-hand side. The parallel move is serialized and the resulting sequence of moves is inserted immediately before the call instruction.

5.3 SSA Deconstruction

The SSA deconstruction algorithm converts a CFG in SSA form into a CFG not in SSA form by eliminating ϕ-functions. Each ϕ-function is replaced with move (copy) instructions that preserve the original program semantics. Figure 5.8 shows the pseudo-code of the SSA deconstruction algorithm used by `wasmc`. `wasmc` implements SSA deconstruction using parallel move serialization [23] combined with move insertion along CFG edges with critical-edge splitting [4, pp. 20-24]. The function `ssa_deconstruction()` of Figure 5.8 iterates over every edge $(from, to)$ in the CFG and, for each edge, the `pmove_from_phis()` function constructs a parallel move from the ϕ-functions at the start of block *to*. Suppose block *to* has the following ϕ-functions at its start:

$$\begin{aligned} v_1 &= \phi(v_1^1, \ldots, v_{from}^1, \ldots, v_n^1) \\ v_2 &= \phi(v_1^2, \ldots, v_{from}^2, \ldots, v_n^2) \\ &\vdots \\ v_m &= \phi(v_1^m, \ldots, v_{from}^m, \ldots, v_n^m) \end{aligned}$$

Since there is an edge $(from, to)$ in the CFG, $from$ is a predecessor of *to*, so each ϕ-function at the start of *to* has an argument associated with $from$.

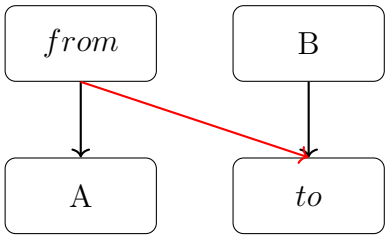
Then `pmove_from_phis()` creates the following parallel move:

$$(v_1, v_2, \ldots, v_m) := (v^1_{from}, v^2_{from}, \ldots, v^m_{from})$$

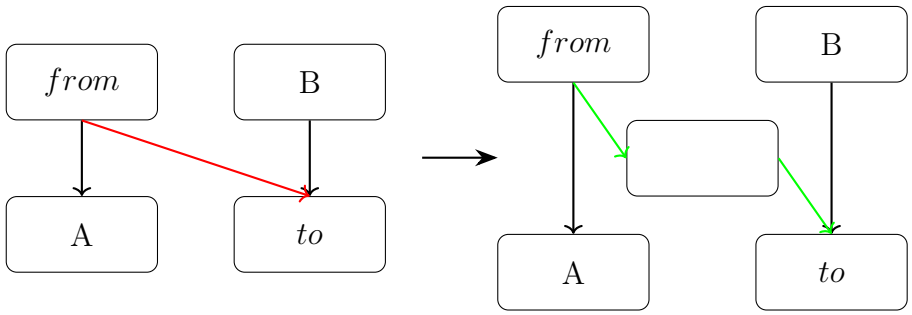
This parallel move is then serialized using `pmove_serialize()`. `wasmc` implements the Leroy parallel move serialization algorithm [23]. `pmove_serialize()` returns a list of move instructions whose effect on $v_1, v_2, \ldots, v_m$ is identical to that of the parallel move. Because the serializer has access to the register allocator's assignment map, it can eliminate redundant moves when both the source and destination variables are mapped to the same location. After serialization, the `insert_smove()` function inserts the sequence of move instructions into a basic block of the CFG along the edge $(from, to)$.

5.3.1 Moves Insertion and Critical Edges

Inserting moves along a CFG edge is not always straightforward. An edge $(from, to)$ is called a **critical edge** if $from$ has multiple successors, and to has multiple predecessors. The following figure shows an example of a critical edge.



If moves must be inserted along the critical edge in this example, neither placing them at the end of $from$ nor at the start of to is correct. Because $from$ has multiple successors, inserting the moves at the end of $from$ would execute them even when control flows to block A. Conversely, because to has multiple predecessors, inserting the moves at the beginning of to would execute them even when control reaches to from block B. The standard solution is therefore to *split* the critical edge by inserting a new empty basic block between $from$ and to. The moves are then placed in this new block.



The function `insert_smove()` implements this logic. Given a sequence of serialized move instructions for the edge $(from, to)$, it inserts them according to the following rules:

- If the edge is critical ($from$ has multiple successors and to has multiple predecessors), the edge is split by creating a new block b. The original edge $(from, to)$ is replaced with the edges $(from, b)$ and (b, to), and the move instructions are inserted into b.
- If $from$ has multiple successors but to has only one predecessor (i.e. $from$), the moves are inserted at the beginning of block to. Because to has no other incoming edges, the moves will only execute when control flows from $from$.
- If $from$ has only one successor (i.e. to), the moves are inserted at the end of block $from$. In this case the moves are executed unconditionally before control transfers to to, which is correct because to is the only successor.

```
def ssa_deconstruction(cfg):
    for (from, to) in cfg.edges:
        pmove_lhs, pmove_rhs = pmove_from_phis(from, to)
        if len(pmove_lhs) == 0: continue
        smove = pmove_serialize(pmove_lhs, pmove_rhs)
        if len(smove) == 0: continue
        insert_smove(cfg, from, to, smove)

def pmove_from_phis(from, to):
    pmove_lhs = list()
    pmove_rhs = list()
    for phi in to.phis:
        pmove_lhs.append(phi.lhs)
        pmove_rhs.append(phi.args[from])
    return pmove_lhs, pmove_rhs

def insert_smove(cfg, from, to, smove):
    if len(from.succ) > 1 and len(to.preds) > 1:
        # Critical edge: split the edge and insert moves in the new
        ↪ block.
        b = cfg.new_block()
        cfg.rm_edge(from, to)
        cfg.add_edge(from, b)
        cfg.add_edge(b, to)
        for mv in smove:
            b.instr_list.append(mv)
    elif len(from.succ) > 1 and len(to.preds) == 1:
        # 'to' has only one predecessor: insert moves at the start
        ↪ of 'to'.
        for mv in reversed(smove):
            to.instr_list.insert(0, mv)
    elif len(from.succ) == 1:
        # 'from' has only one successor: insert moves at the end of
        ↪ 'from'.
        for mv in smove:
            from.instr_list.append(mv)
```

Figure 5.8: Pseudo-code of SSA deconstruction.

5.4 Instruction Selection

Instruction selection is the compiler phase that translates each IR instruction into one or more target machine instructions. `wasmc` implements the simplest possible approach, known as *naive macro expansion* [2, Secs. 2.1 and 2.2]. Each IR instruction is translated independently into a fixed, predetermined sequence of RISC-V instructions. No global analysis is performed, and no attempt is made to combine or optimize across multiple IR instructions. although macro expansion generally produces less efficient code than approaches based on tree-pattern matching or dynamic programming, it is straightforward to implement and resource-effective [2, p. 29]. In `wasmc`, the instruction selection phase is also responsible for the following tasks:

- **Emitting the function prologue and epilogue.** These sequences set up and tear down the stack frame. At the start of each function, the prologue saves the caller's frame pointer and return address on the stack, establishes a new frame pointer, and allocates space in the stack frame for spill slots and saved registers. The epilogue, emitted at every `ret` instruction, performs the inverse operations: it restores all saved registers, deallocates the stack frame, restores the caller's frame pointer and return address, and finally executes the `ret` instruction.
- **Saving and restoring callee-saved registers.** During register allocation, whenever the allocator assigns a callee-saved register to a variable, it records that the register is used. The prologue then saves each such register to a dedicated slot in the stack frame using store instructions, and the epilogue restores them with the corresponding load instructions before returning. Callee-saved registers that are never assigned to any variable are neither saved nor restored, keeping the prologue and epilogue as small as possible.

Chapter 6

The Ahead Of Time File Format

The AOT format is file format for executable, it is not standard, it is used and developed by the WAMR project [6]. The AOT format is designed to facilitate compilation from a WASM module to machine code. The AOT file format is not officially documented anywhere. One of the main contributions of this work is the documentation of the AOT file format, done by reverse engineering the open-source WAMR source code (version 2.4.4, git commit `8c18e3f68b16c4bcaf05996b2636f6ed2b4cf629`). As this description is based on source code analysis rather than official documentation, some details may be incomplete or imprecise. The AOT file format in this chapter is specified using a mix of Extended Backus–Naur Form (EBNF) [41] and C struct syntax. Constraints that cannot be conveniently expressed in EBNF are described in accompanying prose.

6.1 AOT File Format and Its Sections

The AOT file format decodes in file header and sequence of sections following the grammar shown in Figure 6.1. Each section consists of a 32-bit unsigned integer section id, a 32-bit unsigned int section length and the actual section contents. The start of each section must be 4-byte aligned.

```
AOTFile =
    FileHeader,
    TargetInfoSection,
    { CustomSection },
    InitDataSection,
    { CustomSection },
    TextSection,
    { CustomSection },
    FunctionSection,
    { CustomSection },
    ExportSection,
    { CustomSection },
    RelocationSection,
    { CustomSection }
    ;
```

Figure 6.1: The grammar of an AOT file format and its sections.

6.2 WAMR Compiler AOT File Generation

Before proceeding, it is helpful to understand how the WAMR compiler generates an AOT file, as this context clarifies some of the design choices in the AOT format. The WAMR compiler, called `wamrc`, takes a WASM module as input and produces an AOT file as output. During this process, `wamrc` first compiles the WASM bytecode (i.e., the contents of the code section in the WASM module) into LLVM IR, which is then compiled into a relocatable object file. On GNU/Linux systems, LLVM produces a relocatable ELF object file, whereas on Windows it produces a relocatable COFF object file. The relocatable object file (ELF or COFF) is then combined by `wamrc` with the original WASM module to generate the final AOT file. Specifically, the content of the target info section, text section, and relocation section is taken from the relocatable object file, while the content of the init data section, function section, and export section is taken from the original WASM module. As a result, the contents of the target info, text, and relocation sections in the final AOT file may differ depending on whether an ELF or COFF relocatable object file is used.

Important: This documentation describes only the AOT file format obtained from a WASM module combined with a relocatable ELF object file.

6.3 Data Representation

This chapter describes the binary layout of the AOT file format, using an EBNF grammar to define its structure. All data in the AOT file is encoded in *little-endian* byte order.

Important: This documentation focuses exclusively on AOT files targeting little-endian machines. Changes required for big-endian targets are not covered here.

All multi-byte integer values in the AOT file format are encoded in little-endian byte order. Table 6.1 defines the integer types used throughout this document to describe the binary layout of the AOT file format.

Type	Size (bytes)	Description
`u8`	1	Unsigned 8-bit integer
`u16`	2	Unsigned 16-bit integer
`u32`	4	Unsigned 32-bit integer
`u64`	8	Unsigned 64-bit integer

Table 6.1: Integer types used in this document.

The EBNF grammars in this chapter use these integer type names as terminals representing the corresponding sequence of bytes in the file.

6.4 File Header

```
typedef struct {
        uint8_t    magic[4];
        uint32_t   version;
} FileHeader;
```

Figure 6.2: AOT file header

- `magic[4]`: A 4 bytes array holding a ”magic number”, identifying the file as an AOT file.

Name	Value	Position
MAGIC0	'\0'	magic[0]
MAGIC1	'a'	magic[1]
MAGIC2	'o'	magic[2]
MAGIC3	't'	magic[3]

Table 6.2: AOT magic number

- `version`: This member identifies the AOT file version, the current version is 5. The current version is incompatible with the previous versions and it has breaking changes.

  Important: This documentation documents only the AOT file version 5.

6.5 Target Info Section

The target info section has id 0. It decodes into the C struct of Figure 6.3. The start of the target info section must be 4-byte aligned.

```
typedef struct {
        uint16_t   bin_type;
        uint16_t   abi_type;
        uint16_t   e_type;
        uint16_t   e_machine;
        uint32_t   e_version;
        uint32_t   e_flags;
        uint64_t   feature_flags;
        uint64_t   reserved;
        char       arch[16];
} TargetInfo;
```

Figure 6.3: AOT Target Info Section

- `bin_type`: This member identifies from which relocatable object file, ELF or COFF, the content of the target info section, text section and relocation section are taken, see Section 6.2. This member identifies also if the AOT file is targeting a 32-bit or 64-bit architecture and also if the AOT file is targeting a little endian or bit endian machine.

Name	Value	Meaning
BIN_TYPE_ELF32L	0	ELF, 32-bit, little endian
BIN_TYPE_ELF32B	1	ELF, 32-bit, big endian
BIN_TYPE_ELF64L	2	ELF, 64-bit, little endian
BIN_TYPE_ELF64B	3	ELF, 64-bit, big endian
BIN_TYPE_COFF32	4	COFF, 32-bit, little endian
BIN_TYPE_COFF64	6	COFF, 64-bit, little endian

Table 6.3: Binary type legal values

In the Table 6.3 all the little endian entry have the first least significant bit set to 0 and all the big endian entry have the first least significant bit set to 1. So with `bin_type & 1` you can know if the AOT file is targeting a little endian or big endian machine. In the Table 6.3 all the 32-bit entry have the second least significant bit set to 0 and all the 64-bit entry have the second least significant bit set to 1. So with `bin_type & 2` you can know if the AOT file is targeting a 32-bit or 64-bit machine. The value 5 in Table 6.3 is missing for these two reasons.

- `abi_type`: This member is never used by WAMR, the WAMR compiler set it always to 0.
- `e_type`: Table 6.4 contains the possible values for the `e_type` member.
  1. `E_TYPE_REL` simply means that the target info section, text section and relocation section are taken from a relocatable object file, see Section 6.2.
  2. `E_TYPE_XIP` has the same meaning of `E_TYPE_REL` but it also enable the WAMR execution in place feature[1].

Name	Value	Meaning
E_TYPE_REL	1	Relocatable object file
E_TYPE_XIP	4	Enable execution in place

Table 6.4: e_type legal values

- `e_machine`: This member specifies the machine instruction set for which the AOT file is intended. It indicating the type of machine

[1]See `wasm-micro-runtime/doc/xip.md` in the WAMR repository.

instruction set that the executable code in the text section is designed to run on. `e_machine` in the AOT target info section has the same meaning of `e_machine` in the ELF header [43, Ch. 2].

Name	Value	Meaning
`E_MACHINE_X86_64`	62	AMD x86-64 architecture
`E_MACHINE_WIN_X86_64`	0x8664	Windows x86-64 architecture
`E_MACHINE_386`	3	Intel 80386
`E_MACHINE_WIN_I386`	0x14c	Windows i386 architecture
`E_MACHINE_ARM`	40	ARM 32-bit architecture (AARCH32)
`E_MACHINE_AARCH64`	183	ARM 64-bit architecture (AARCH64)
`E_MACHINE_MIPS`	8	MIPS I Architecture
`E_MACHINE_XTENSA`	94	Tensilica Xtensa Architecture
`E_MACHINE_RISCV`	243	RISC-V 32/64 bit
`E_MACHINE_ARC_COMPACT`	93	ARC International ARCompact
`E_MACHINE_ARC_COMPACT2`	195	Synopsys ARCompact V2

Table 6.5: e_machine values

The difference between `E_MACHINE_X86_64` and `E_MACHINE_WIN_X86_64`, `E_MACHINE_386` and `E_MACHINE_WIN_I386` is due to the difference between ELF and COFF format, see Section 6.2. `E_MACHINE_X86_64` and `E_MACHINE_386` are taken from the `e_machine` field of an ELF header, instead `E_MACHINE_WIN_X86_64` and `E_MACHINE_WIN_I386` have the same meaning of the non Windows version but they are taken from a COFF header.

- `e_version`: This member identifies the object file version from witch the target info section, text section and relocation section are taken, see Section 6.2.

Name	Value	Meaning
`E_VERSION_CURRENT`	1	Current version

Table 6.6: e_version legal values.

- `e_flags`: This member holds processor-specific flags. `e_flags` in the AOT target info section has the same meaning of `e_flags` in the ELF header [43, Ch. 2]. The legal values of `e_flags` are defined in ABI specification of the machine instruction set. For example see "Chapter

8. ELF Object Files" in RISC-V ABI specification [12]. The `e_flags` member is never used by WAMR, so you can just put 0[2].

- `feature_flags`: This member enable WASM features, each bit of `feature_flags` enable a WASM feature if set to 1. If you don't use any WASM feature just put 0. The WASM features supported by WAMR are not documented here.

  Important: The activation of some feature flag can change the content of some part of the AOT file format. This documentation documents only the AOT file format with `feature_flags` equal to 0.

- `reserved`: Reserved for future use, just set it to 0.

- `arch[16]`: Machine architecture name. Each machine architecture name must match with the corresponding `e_machine` value. Table 6.7 contains the supported `arch` values and the corresponding `e_machine` value.

arch	e_machine
`"x86_64"`	`E_MACHINE_X86_64`, `E_MACHINE_WIN_X86_64`
`"i386"`	`E_MACHINE_386`, `E_MACHINE_WIN_I386`
`???`	`E_MACHINE_ARM`
`???`	`E_MACHINE_AARCH64`
`"mips"`	`E_MACHINE_MIPS`
`"xtensa"`	`E_MACHINE_XTENSA`
`"riscv"`	`E_MACHINE_RISCV`
`"arc"`	`E_MACHINE_ARC_COMPACT`, `E_MACHINE_ARC_COMPACT2`

Table 6.7: arch values.

  The remaining bytes between the end of the architecture name and the end of the `arch` buffer must be zero.

6.6 Init Data Section

The init data section has id 1. The start of the init data section must be 4-byte aligned. The init data section decodes to an id, a 32-bit unsigned integer indicating the section length in bytes, a 32-bit unsigned int section length

[2] `wasmc` sets `e_flags` to 4 simply because `wamrc` does so, there is no specific rationale behind this choice.

and sequence of subsections following the grammar shown in Figure 6.4. The start of each subsections must be 4-byte aligned.

```
(* The start of InitDataSection
   must be 4-byte aligned. *)
InitDataSection =
    InitDataId,
    u32, (* length of the section in bytes *)
    MemorySubsection,
    TableSubsection,
    TypeSubsection,
    ImportGlobalSubsection,
    GlobalSubsection,
    ImportFunctionSubsection,
    AuxiliarySubsection,
    ObjectDataSubsection
    ;
InitDataId = "0x01", "0x00", "0x00", "0x00" ;
```

Figure 6.4: Init data section grammar

The AOT format is designed to facilitate the compilation of a WASM module. The AOT format is able to facilitate the compilation of a WASM module because many parts of an AOT file can be taken directly from the WASM module with few modifications. Table 6.8 summarizes from which section of the WASM form the information of each init data subsection is taken.

Init Data Subsection	WASM Sections
Memory	Memory, Data
Table	Table, Import
Type	Type
Import Global	Import
Global	Global
Import Function	Import
Auxiliary	Start, Export, Global
Object Data	- not taken from the WASM module -

Table 6.8: Mapping between init data subsections and WASM sections.

6.6.1 Memory Subsection

The content and the structure of the memory subsection are very similar to the memory and data section of a WASM module [25, Secs. 5.5.8 and 5.5.14]. The start of the memory subsection must be 4-byte aligned. The memory subsection decodes in a vector of WASM memory and in a vector of WASM data segments following the grammar shown in Figure 6.5. A vector of WASM memory is encoded with the 32-bit unsigned int specifying the number of elements, followed by a sequence of WASM memories. The vector of WASM memory has always 1 element. A WASM memory is composed of four members in the following order.

1. A four byte boolean flag indicating whether a maximum number of memory pages is present, zero means unlimited.
2. The number of bytes in a memory page.
3. The initial number of memory pages. The initial number of pages must be always less then or equal to the maximum number of pages.
4. The maximum number of memory pages. If the four byte boolean flag is 0, the maximum number of pages can be set to 65536.

If the WASM module doesn't have a memory (a WASM module can only have one or zero memory), the WASM Memory members in the AOT module must be filled with these default values, respectively: 0, 65536, 0, 0. A vector of WASM data segments is encoded with the 32-bit unsigned int specifying the number of elements, followed by a sequence of WASM data segments. The initial contents of a WASM memory are zero-valued bytes. A WASM data segments initialize a range of a WASM memory, at a given offset, with a static buffer of bytes. A WASM data segment is composed of five members in the following order.

1. The first member is always zero.
2. A memory index identifying which WASM memory this data segment initialize. Since there can only be one WASM memory, the memory index is always 0.
3. The four bytes `0x41 0x00 0x00 0x00`.
4. Offset in the WASM linear memory where this data segment starts.
5. A buffer of bytes.

If a WASM module has been successfully validated and has no WASM memory, it cannot have WASM data segments. As a result, this must also apply in an AOT file.

```
(* The start of MemorySubsection
   must be 4-byte aligned *)
MemorySubsection =
    u32, (* Always zero *)
    u32, (* Number of WASMMemory elements *)
    WASMMemory,
    u32, (* Number of WASMDataSegment elements *)
    { WASMDataSegment }
    ;

WASMMemory =
    u32, (* Boolean flag indicating
            whether a maximum is present *)
    u32, (* Bytes per page *)
    u32, (* Initial page count *)
    u32  (* Maximum page count *)
    ;

(* The start of each WASMDataSegment
   must be 4-byte aligned *)
WASMDataSegment =
    u32, (* Always 0 *)
    u32, (* Memory index, always 0 *)
    "0x41", "0x00", "0x00", "0x00",
    u32, (* Index in the WASM linear memory
            where this data segment starts *)
    Buffer
    ;
(* u32 is the number of bytes *)
Buffer = u32, { u8 } ;
```

Figure 6.5: Memory subsection grammar.

6.6.2 Table Subsection

The content and structure of the table subsection are very similar to the table section and the import section of a WASM module [25, Secs. 5.5.7 and 5.5.5]. The start of the table subsection must be 4-byte aligned. The table subsection decodes to a vector of imported WASM tables followed by a

vector of WASM tables, as defined by the grammar in Figure 6.6. A vector of imported WASM tables is encoded as a 32-bit unsigned integer specifying the number of elements, followed by a sequence of imported WASM table elements. A vector of WASM tables is encoded as a 32-bit unsigned integer specifying the number of elements, followed by a sequence of WASM table elements. This document does not specify the structure of an imported WASM table nor the structure of a WASM table. It is assumed that the number of elements of the vector of imported WASM tables is zero and that the number of elements of the vector of WASM tables is zero. Therefore, the table subsection consists only of two 32-bit unsigned integers, both equal to zero.

```
(* The start of TableSubsection
   must be 4-byte aligned *)
TableSubsection =
    u32, (* Number of WASMImportTable elements *)
    { WASMImportTable },
    u32, (* Number of WASMTable elements *)
    { WASMTable }
    ;

WASMImportTable = ; (* Not specified *)
WASMTable = ;       (* Not specified *)
```

Figure 6.6: Grammar of the table subsection.

6.6.3 Type Subsection

The content and structure of the type subsection are very similar to the type section of a WASM module [25, Sec. 5.5.4]. The start of the type subsection must be 4-byte aligned. The type subsection decodes to a vector of WASM function types following the grammar shown in Figure 6.7. A vector of WASM function types is encoded as a 32-bit unsigned integer specifying the number of elements, followed by a sequence of WASM function types. The start of each WASM function type in the vector must also be 4-byte aligned. A WASM function type is composed of four members in the following order:

- `TypeFlag`: This member is 2 bytes long and must be zero. If some feature flags are enabled (see Section 6.5), the type subsection may contain types other than WASM function types. In that case, this field is used to distinguish between different type kinds.

- A 2-byte unsigned integer specifying the number of parameters of the WASM function type.
- A 2-byte unsigned integer specifying the number of return values of the WASM function type.
- An array of WASM value types whose length is equal to the sum of the number of parameters and the number of return values. The array first contains the value types of the function parameters, followed by the value types of the return values. Each WASM value type is encoded as a single byte, as shown in Table 6.9.

```
(* The start of TypeSubsection
   must be 4-byte aligned *)
TypeSubsection =
    u32, (* Number of WASMFuncType elements *)
    { WASMFuncType }
    ;

(* The start of each WASMFuncType
   must be 4-byte aligned *)
WASMFuncType =
    TypeFlag,
    u16, (* Number of parameters *)
    u16, (* Number of return values *)
    { WASMValtype }
    ;

TypeFlag = "0x00", "0x00" ;
WASMValtype = "0x7F" | "0x7E" | "0x7D" | "0x7C" ;
```

Figure 6.7: Grammar of the type subsection.

WASM value type	Meaning
"0x7F"	WASM 32-bit integer
"0x7E"	WASM 64-bit integer
"0x7D"	WASM 32-bit floating-point number
"0x7C"	WASM 64-bit floating-point number

Table 6.9: WASM value types [25, Secs. 2.3.1 and 5.3.1].

6.6.4 Import Global Subsection

The content and structure of the import global subsection are very similar to the import section of a WASM module [25, Sec. 5.5.5]. The start of the import global subsection must be 4-byte aligned. The import global subsection decodes to a vector of imported WASM global variables, following the grammar shown in Figure 6.8. A vector of imported WASM global variables is encoded as a 32-bit unsigned integer specifying the number of elements, followed by a sequence of imported WASM global variable entries. This document does not specify the structure of an imported WASM global variable. It is assumed that the number of elements of the vector of imported WASM global variables is zero. Therefore, the import global subsection consists only of a single 32-bit unsigned integer equal to zero.

```
(* The start of ImportGlobalSubsection
   must be 4-byte aligned *)
ImportGlobalSubsection =
    u32, (* Number of WASMImportGlobal elements *)
    { WASMImportGlobal }
    ;

WASMImportGlobal = ; (* Not specified *)
```

Figure 6.8: Grammar of the import global subsection.

6.6.5 Global Subsection

The content and structure of the global subsection are very similar to the global section of a WASM module [25, Sec. 5.5.9]. The start of the global subsection must be 4-byte aligned. The global subsection decodes to a vector of WASM globals following the grammar shown in Figure 6.9. A vector of WASM globals is encoded as a 32-bit unsigned integer specifying the number of elements, followed by a sequence of WASM globals. The start of each WASM global in the vector must be 4-byte aligned. A WASM global consists of three members in the following order:

1. A WASM value type. WASM value types are encoded as a single byte, as shown in Table 6.9.
2. A one-byte boolean mutability flag.
3. A WASM initial expression. The start of the WASM initial expression within a WASM global must be 4-byte aligned. A WASM initial

expression is a triple consisting of: initial expression type, three zero bytes, and the actual initial expression payload. The initial expression type is encoded as a single byte, as shown in Table 6.10.

```
(* The start of GlobalSubsection
   must be 4-byte aligned *)
GlobalSubsection =
    u32, (* Number of WASMGlobal elements *)
    { WASMGlobal }
    ;

(* The start of each WASMGlobal must be 4-byte aligned *)
WASMGlobal = WASMValtype, IsMutable, InitExpr ;

WASMValtype = "0x7F" | "0x7E" | "0x7D" | "0x7C" ;
IsMutable   = "0x00" | "0x01" ;

(* The start of each InitExpr must be 4-byte aligned *)
InitExpr =
      "0x41", "0x00", "0x00", "0x00", u32
    | "0x42", "0x00", "0x00", "0x00", i64
    | "0x43", "0x00", "0x00", "0x00", f32
    | "0x44", "0x00", "0x00", "0x00", f64
    | "0x23", "0x00", "0x00", "0x00", u32
    ;
```

Figure 6.9: Global subsection grammar.

Initial expression type	Meaning
"0x41"	WASM 32-bit integer constant
"0x42"	WASM 64-bit integer constant
"0x43"	WASM 32-bit float constant
"0x44"	WASM 64-bit float constant
"0x23"	index of another WASM globals

Table 6.10: Initial expression type.

6.6.6 Import Function Subsection

The content and structure of the import function subsection are very similar to the import section of a WASM module [25, Sec. 5.5.5]. The start of

the import function subsection must be 4-byte aligned. The import function subsection decodes to a vector of imported WASM functions following the grammar shown in Figure 6.10. A vector of imported WASM functions is encoded as a 32-bit unsigned integer specifying the number of elements, followed by a sequence of imported WASM functions. This documentation does not specify the structure of an imported WASM function. It is assumed that the number of elements of the vector of imported WASM functions is zero. Therefore, the import function subsection consists only of a zero 32-bit unsigned integer.

```
(* The start of ImportFunctionSubsection
   must be 4-byte aligned *)
ImportFunctionSubsection =
    u32, (* Number of WASMImportFunction elements *)
    { WASMImportFunction }
    ;

WASMImportFunction = ; (* not specified *)
```

Figure 6.10: Import function subsection grammar.

6.6.7 Auxiliary Subsection

The start of the auxiliary subsection must be 4-byte aligned. The subsection decodes into a tuple following the grammar shown in Figure 6.11.

```
(* The start of AuxiliarySubsection
   must be 4-byte aligned *)
AuxiliarySubsection =
    u32, (* Function count *)
    u32, (* Start function index *)
    u32, (* "__data_end" global variable index *)
    u64, (* data end linear memory index *)
    u32, (* "__heap_base" global variable index *)
    u64, (* heap base linear memory index *)
    u32, (* stack pointer global variable index *)
    u64, (* stack pointer linear memory index *)
    u32  (* stack size *)
```

Figure 6.11: Auxiliary subsection grammar

Basic fields

- Function count: the number of functions in the module.
- Start function index: the index of the start function [25, Sec. 5.5.11].

The remaining fields describe the *layout of the WASM linear memory.*

Linear memory layout WASM modules are typically generated by compiling a high-level language rather than being written manually. For example, `clang` can compile a C program into a WASM module. During this compilation process, the C memory layout (initialized data, uninitialized data, stack, and heap) is mapped onto the WASM linear memory. Initialized and uninitialized data are mapped in a single data segment inside the WASM linear memory. `clang` stores in the WASM module in output where the data segment starts and ends in the WASM linear memory, the stack starts and ends in the WASM linear memory, the heap starts and ends in the WASM linear memory. These linear memory layout information are store partly in the *global section* of the WASM module and partly in the *export section* of the WASM module. Figure 6.12 summarizes the resulting linear memory layout for a WASM module generated from a C program.

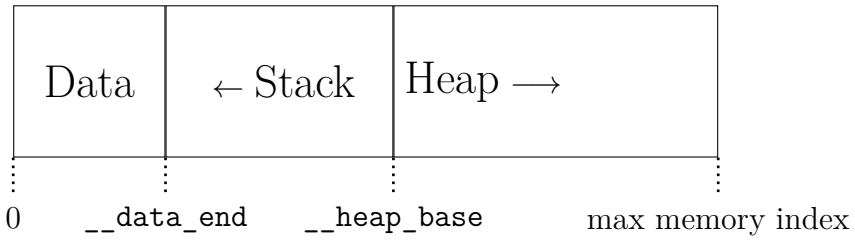


Figure 6.12: Linear memory layout of a WASM module compiled from a C program.

Detecting C-generated WASM modules When generating the auxiliary subsection, it is first necessary to determine whether the WASM module was compiled from a C program. [3] A module is considered to be compiled from C if the following conditions are satisfied:

[3] A programmer can of course write a WASM module by imitating the behavior of a compiler by hand. Let's suppose, for simplicity, that that WASM module is still generated by a compiler.

1. The module exports a global variable named `"__data_end"`. The global must have type `i32`, must be immutable, and must be initialized with a constant expression.

2. The module exports a global variable named `"__heap_base"`. This global must also have type `i32`, must be immutable, and must be initialized with a constant expression.

3. The initial value of `"__data_end"` is less than or equal to the initial value of `"__heap_base"`.

Default values If the module is not generated from C, the auxiliary subsection fields are filled with the following default values:

- `"__data_end"` global variable index: $2^{32} - 1$
- Data end linear memory index: 0
- `"__heap_base"` global variable index: $2^{32} - 1$
- Heap base linear memory index: 0
- Stack pointer global variable index: $2^{32} - 1$
- Stack pointer linear memory index: 0
- Stack size: 0

Extracting memory layout information If the module was compiled from a C program, the memory layout can be reconstructed as follows:

- Data starts at linear memory index 0.
- The data end index is stored in the global variable `"__data_end"`. The data segment corresponds to the interval $[start, end)$.
- The stack starts at the data end index and grows toward the heap.
- The heap base index is stored in the global variable `"__heap_base"`.
- The heap extends from the heap base index to the maximum memory index.

The auxiliary subsection fields are therefore filled as follows:

- `"__data_end"` global variable index: index of the exported global `"__data_end"`.
- Data end linear memory index: initial value of the `"__data_end"` global.
- `"__heap_base"` global variable index: index of the exported global `"__heap_base"`.
- Heap base linear memory index: initial value of the `"__heap_base"` global.

Stack pointer detection The remaining fields describe the stack pointer. The index of the global variable holding the stack pointer can be determined using the algorithm shown in Figure 6.13.

```
# heap_base_idx is the heap base linear memory index
def get_stack_ptr_global_index(wasm_module, heap_base_idx):
    for i in range(wasm_module.global_count):
        g = wasm_module.globals[i]
        if (g.is_mutable and g.type == WASM_VALTYPE_I32 and
            g.init_value <= heap_base_idx):
            return i
    return -1
```

Figure 6.13: Get the index of the global variable that holds the stack pointer.

Let `sp_index` be the value returned by the function of Figure 6.13.
If `sp_index` is not `-1`:

- Stack pointer global variable index: `sp_index`
- Stack pointer linear memory index: set to initial value of the global variable of index `sp_index`.
- Stack size: (stack pointer index) - (data end index)

Otherwise:

- Stack pointer global variable index: set to `"__heap_base"` global variable index.
- Stack pointer linear memory index: heap base index
- Stack size: 0

6.6.8 Memory shrink optimization

The WAMR compiler, `wamrc`, also performs the optimization shown in the code snippet of Figure 6.14 after initializing the WASM memory members in the memory subsection, as explained in Section 6.6.1. This optimization can only be applied if the WASM module has a single memory and does not contain any `memory.grow` or `memory.size` operations in any function (see Section [25, Sec. 2.4.4] for details on `memory.grow` and `memory.size`). The members of the WASM memory in the memory subsection are updated using the optimization shown in Figure 6.14:

```
bytes_per_page *= initial_page_count
if initial_page_count > 0:
    initial_page_count = 1
    maximum_page_count = 1
    # Can fail if the WASM module is not compiled
    # from a C program, see Auxiliary Subsection
    ok, heap_base = get_heap_base_linear_memory_index()
    if ok:
        bytes_per_page = heap_base
```

Figure 6.14: Memory shrink optimization

Typically, a WASM module has a memory of two pages of 64 KB each, so the runtime must allocate a contiguous array of 128 KB in the heap for the linear memory. On devices with very limited memory, external fragmentation may prevent the allocation of a contiguous 128 KB region in the heap. The solution to this problem is to shrink the linear memory. Using the optimization in Figure 6.14, the WAMR compiler instructs the WAMR runtime to allocate only enough contiguous linear memory to hold the data section and the stack (see Section 6.6.7 and Figure 6.12). If the WASM module code calls `malloc`, the call is intercepted, and the required memory is allocated in the heap of the WAMR runtime rather than in the WASM linear memory dedicated to the heap.

6.6.9 Object Data Subsection

The start of the object data subsection must be 4-byte aligned. The object data subsection decodes to a vector of object data elements following the grammar shown in Figure 6.15. A vector of object data elements is encoded as a 32-bit unsigned integer specifying the number of elements, followed by

a sequence of object data elements. This documentation does not specify the structure of an object data element. It is assumed that the number of elements of the vector of object data elements is zero. Therefore, the object data subsection consists only of a zero 32-bit unsigned integer.

```
(* The start of ObjectDataSubsection
   must be 4-byte aligned *)
ObjectDataSubsection =
    u32, (* Number of ObjectData elements *)
    { ObjectData }
    ;

(* The start of each ObjectData
   must be 2-byte aligned *)
ObjectData = ; (* not specified *)
```

Figure 6.15: Object data subsection grammar.

6.7 Text Section

The function section has id 2. The start of the function section must be 4-byte aligned. The function section decodes to an id, a 32-bit unsigned integer indicating the section length in bytes and the machine code of each compiled WASM function following the grammar shown in Figure 6.16.

```
(* The start of TextSection
   must be 4-byte aligned *)
TextSection =
    TextId,
    (* Length of the section in bytes *)
    u32,
    (* One u32 set to zero *)
    "0x00", "0x00", "0x00", "0x00",
    (* The machine code of each
       compiled WASM function *)
    { u8 }
    ;

TextId = "0x02", "0x00", "0x00", "0x00" ;
```

Figure 6.16: Text section grammar.

6.8 Function Section

The function section has id 3. The content and structure of the function section are very similar to the function section of a WASM module [25, Sec. 5.5.6]. The start of the function section must be 4-byte aligned. The function section decodes to an id, a 32-bit unsigned integer indicating the section length in bytes and four 32-bit integer vectors following the grammar shown in Figure 6.17. Each vector has a number of elements equal to the number of functions (see "function count" in Section 6.6.7). The first vector contains offsets into the text section. The i-th element of this vector specifies the beginning of the code of the i-th function. The second vector contains type indices. The i-th element of this vector specifies that the i-th function has the type indexed by that value in the type vector defined in the type subsection of the init data. The elements of the third and fourth vectors are always zero.

```
(* The start of FunctionSection
   must be 4-byte aligned *)
FunctionSection =
    FunctionId,
    u32, (* Length of the section in bytes *)
    { u32 }, (* Offset in the text section *)
    { u32 }, (* Type index *)
    { u32 }, (* Always zero *)
    { u32 }, (* Always zero *)
    ;

FunctionId = "0x03", "0x00", "0x00", "0x00" ;
```

Figure 6.17: Function section grammar.

6.9 Export Section

The export section has id 4. The content and structure of this section are very similar to the export section of a WASM module [25, Sec. 5.5.10]. The start of the export section must be 4-byte aligned. The export section is decoded as an id, a 32-bit unsigned integer indicating the section length in bytes, and a vector of WASM export elements following the grammar shown in Figure 6.18. A vector of WASM exports is encoded as a 32-bit unsigned integer specifying the number of elements, followed by a sequence of WASM

exports. The start of each WASM export in the vector must be 4-byte aligned. A WASM export consists of three members in the following order:

- An export index [25, Sec. 5.5.10].
- An export description [25, Sec. 5.5.10].
- A string. The start of each string must be 2-byte aligned (not 4-byte aligned). A string is encoded as a 16-bit unsigned integer length followed by a sequence of bytes terminated by a zero byte. The length of the string also includes the final zero byte.

```
(* The start of ExportSection
   must be 4-byte aligned *)
ExportSection =
    ExportId,
    u32, (* Length of the section in bytes *)
    u32, (* Number of WASMExport elements *)
    { WASMExport }
    ;

ExportId = "0x04", "0x00", "0x00", "0x00" ;

(* The start of each WASMExport
   must be 4-byte aligned *)
WASMExport =
    u32, (* Export index *)
    u8,  (* Export description *)
    String
    ;

(* The start of each String
   must be 2-byte aligned *)
String =
    u16, (* Number of bytes of the string
            including the final zero byte *)
    { u8 },
    "0x00"
    ;
```

Figure 6.18: Export section grammar.

6.10 Relocation Section

The relocation section has id 5. The start of the relocation section must be 4-byte aligned. The relocation section is decoded as an id, a 32-bit unsigned integer indicating the section length in bytes and other fields. This documentation does not specify the content and the structure of relocation section. The minimal and empty relocation section is shown in Figure 6.19.

```
(* The start of RelocationSection
   must be 4-byte aligned *)
RelocationSection =
    RelocationId,
    (* Length of the section in bytes (12 bytes) *)
    "0x0c", "0x00", "0x00", "0x00",
    (* three u32 set to zero *)
    "0x00", "0x00", "0x00", "0x00",
    "0x00", "0x00", "0x00", "0x00",
    "0x00", "0x00", "0x00", "0x00"
    ;

RelocationId = "0x05", "0x00", "0x00", "0x00" ;
```

Figure 6.19: Relocation section grammar.

6.11 Custom Section

The custom section has id 100. The start of the custom section must be 4-byte aligned. The custom section is decoded as an id, a 32-bit unsigned integer indicating the section length in bytes and other bytes as shown in Figure 6.20. I don't know the purpose of the custom section.

```
(* The start of CustomSection
   must be 4-byte aligned *)
CustomSection =
    CustomId,
    u32, (* Length of the section in bytes *)
    { u8 }
    ;

CustomId = "0x64", "0x00", "0x00", "0x00" ;
```

Figure 6.20: Custom section grammar.

Chapter 7

Benchmarks and Evaluation

In this chapter we evaluate `wasmc` to answer the following questions:

- **Q1:** Can `wasmc` effectively compile non-trivial WASM modules on a resource-constrained device, and what are the resulting heap memory consumption and execution time during compilation?
- **Q2:** Since the only reliable way to execute WASM modules on resource-constrained devices is currently through interpretation, does executing a WASM module compiled by `wasmc` achieve better performance than executing the same module using an interpreter?
- **Q3:** Since `wasmc` performs compilation directly on the target device, how does the performance of the generated code compare with that produced by an off-device compiler?

7.1 Methodology

Benchmarks. All benchmarks were executed on an `ESP32-C3` device using the `esp-idf` framework. To evaluate the performance of `wasmc`, the following benchmarks were used:

- `sieve`: computes prime numbers up to 2^{16} using the Sieve of Eratosthenes algorithm.
- `fibonacci`: recursively computes the 10th Fibonacci number.
- `matrix-mult`: multiplies two integer matrices of size 50×50.

- `lu-decomp`: performs LU decomposition of an integer matrix of size 50×50. The matrix is constructed so that all division operations produce integer results with no remainder, avoiding truncation.
- `0-1-knapsack`: solves the 0–1 knapsack problem using dynamic programming with 5000 objects and a knapsack capacity of 5000.
- `crc32`: computes CRC32 checksum over a buffer of 15000 bytes.
- `edit-distance`: computes the Levenshtein distance between two strings of length 100 using dynamic programming.
- `gauss-elim`: solves a system of linear equations using Gaussian elimination with 100 equations, 100 variables, and integer coefficients. The matrix associated with the system is constructed so that all division operations produce integer results with no remainder, avoiding truncation.

All benchmarks are written in C and compiled to WASM using `clang` version 19.1.7 at optimization level `-Os`. The AOT executables produced by `wasmc` are executed using the WAMR [6] AOT runtime (version 2.4.4). All WASM and AOT modules are statically allocated; therefore, they do not contribute to the peak heap usage reported in any benchmark. The source code of the benchmarks is available at:

`https://github.com/lllllluca/wasmc/tree/main/tests/wasm/src`

Baselines. The following existing systems are used for comparison:

- `iwasm`: the WASM interpreter of the WAMR [6] project (version 2.4.4).
- `wamrc`: the off-device AoT compiler of the WAMR [6] project (version 2.4.4).

For a fair comparison with `wamrc`, the safety checks of `wamrc` were disabled, since `wasmc` does not implement these safety checks. `wamrc` is run with the following flags:

```
wamrc \
    --target=riscv32 \
    --format=aot \
    --stack-bounds-checks=0 \
    --bounds-checks=0 \
    -o <name>.aot <name>.wasm
```

7.2 Compilation Time and Memory Usage

Table 7.1 reports the compilation time and peak heap memory usage of `wasmc` for each benchmark, measured directly on the `ESP32-C3` microcontroller.

Benchmark	**Time** (ms)	**Heap Peak** (KB)
`sieve`	6.6	4.3
`fibonacci`	4.6	3.3
`heapsort`	10.5	10.9
`matrix-mult`	12.0	12.8
`lu-decomp`	26.6	14.8
`0-1-knapsack`	11.5	5.5
`floyd-warshall`	13.1	6.8
`crc32`	11.7	6.2
`edit-distance`	16.5	16.8
`gauss-elim`	25.1	14.3

Table 7.1: Compilation time and heap peak memory usage of `wasmc`.

The results in Table 7.1 give a clear affirmative answer to **Q1**. `wasmc` successfully compiled all ten benchmarks directly on the `ESP32-C3`, which has only 400 KB of RAM. No benchmark required more than 16.8 KB of peak heap memory during compilation, and no benchmark took longer than 26.6 ms to compile. These figures confirm that `wasmc` can compile non-trivial WASM modules entirely on-device within the memory and time constraints of a microcontroller.

7.3 Comparison with Interpretation

Execution Time Table 7.2, Figure 7.1 and Figure 7.2 compare the execution time of the AOT executables produced by `wasmc` against the WAMR interpreter running the same WASM modules on the `ESP32-C3`.

Benchmark	**AOT (`wasmc` + WAMR) Exec. Time** (ms)	**WAMR Interp. Exec. Time** (ms)	**Speedup of AOT (wasmc + WAMR) over Interpreter**
`sieve`	20.8	383.2	18.4
`fibonacci`	8.4	118.7	14.1
`heapsort`	56.3	1125.0	20.0
`matrix-mult`	13.6	297.1	21.8
`lu-decomp`	18.8	396.5	21.1
`0-1-knapsack`	2815.0	68 075.1	24.2
`floyd-warshall`	124.8	2190.0	17.5
`crc32`	3.3	60.9	18.5
`edit-distance`	1.7	37.3	21.9
`gauss-elim`	228.3	2231.4	9.8

Table 7.2: Execution time of benchmarks between the AOT executable generated by `wasmc` running on the WAMR AOT runtime and the WAMR WASM interpreter.

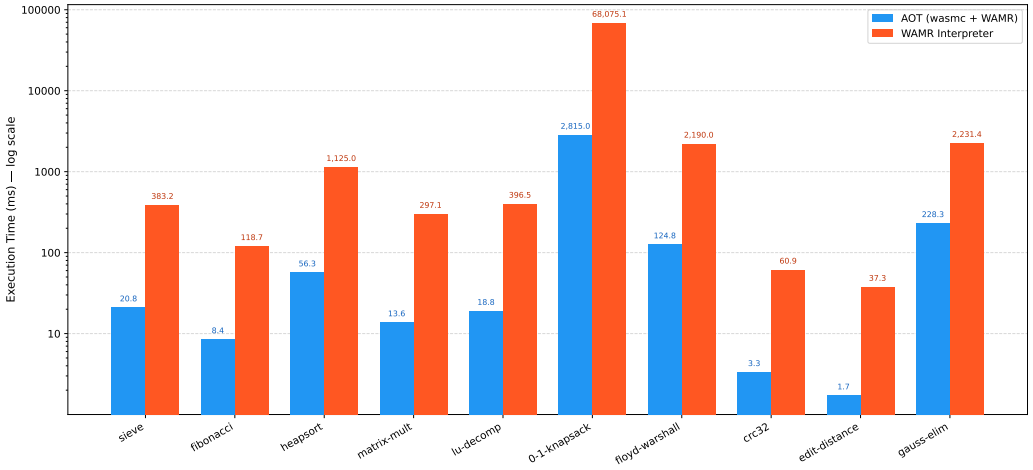


Figure 7.1: Execution time of AOT (`wasmc` + WAMR) vs WAMR interpreter (log scale).

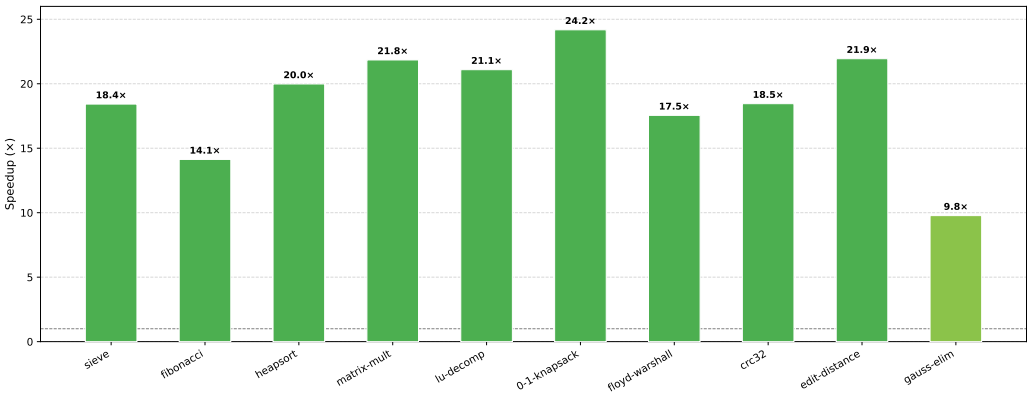


Figure 7.2: Speedup of AOT (`wasmc` + WAMR) over the WAMR interpreter.

The AOT executables produced by `wasmc` consistently and substantially outperform the WAMR interpreter across all ten benchmarks. The speedup ranges from a minimum of approximately 9.8× on `gauss-elim` to a maximum of approximately 24.2× on `0-1-knapsack`. The consistently high speedup across all benchmarks demonstrates the advantage of `wasmc` AoT compilation over interpretation. The performance gap between AOT and interpretation is expected and well understood: an interpreter must decode and dispatch each WASM bytecode instruction at runtime, incurring overhead on every operation, whereas the AOT executable produced by `wasmc` consists of native 32-bit RISC-V instructions that execute directly on the hardware with no runtime decoding overhead. These results confirm that `wasmc` achieves its goal of producing efficient native code, demonstrating that AoT compilation of a WASM module is a practical and substantially faster alternative to interpretation on resource-constrained devices.

Heap Peak Table 7.3 reports the peak heap memory usage at runtime for both the AOT executables produced by `wasmc` and the WAMR interpreter, measured on the `ESP32-C3`.

Benchmark	**AOT (`wasmc` + WAMR) Heap Peak** (KB)	**WAMR Interp. Heap Peak** (KB)
`sieve`	149.1	149.2
`fibonacci`	79.4	79.4
`heapsort`	137.2	137.4
`matrix-mult`	108.6	108.9
`lu-decomp`	109.3	110.3
`0-1-knapsack`	137.7	138.2
`floyd-warshall`	120.9	121.3
`crc32`	96.3	96.9
`edit-distance`	121.1	121.6
`gauss-elim`	121.7	122.8

Table 7.3: Runtime heap peak memory usage of benchmarks between the AOT executable generated by `wasmc` running on the WAMR AOT runtime and the WAMR WASM interpreter.

The two runtimes exhibit nearly identical peak heap usage across all benchmarks. The difference between AOT and interpreter is at most a few hundred bytes in every case. This near-parity is expected: the dominant contribution to runtime heap usage is the WASM linear memory allocated for the module itself, which is identical in both cases since both runtimes execute the same WASM module with the same memory layout. The overhead specific to the runtime, the WAMR AOT loader and executor on one side, and the WAMR interpreter and its bytecode dispatch tables on the other, is small and comparable in size. These results show that switching from interpretation to AoT compilation with `wasmc` comes at no meaningful cost in runtime memory: the executed programs consume essentially the same amount of heap whether they run as native AOT code or as interpreted WASM bytecode.

Answer to Q2. Yes, executing WASM modules compiled by `wasmc` is significantly faster than interpretation. Across all benchmarks, speedups range from approximately 9.8× to 24.2×, while runtime memory usage remains nearly identical. Therefore, `wasmc` clearly outperforms the WAMR interpreter on resource-constrained devices.

7.4 Comparison with Off-Device Compilation

Tables 7.4 and 7.5 compare the execution time and runtime heap peak of AOT executables produced by `wasmc` against those produced by `wamrc`. Both executables are run on the same WAMR AOT runtime on the `ESP32-C3`. `wamrc` uses LLVM as its backend and applies a full suite of compiler optimizations. `wasmc`, by contrast, runs entirely on the device, uses no external framework, and applies no IR optimizations (see Section 3). Only two benchmarks have `wamrc` results available: `fibonacci` and `crc32`. The remaining benchmarks could not be compiled by `wamrc` and executed by the AOT runtime for the 32-bit RISC-V target of the `ESP32-C3`, and are therefore marked with "-" in the tables. The `wamrc` compiler does not fully support the 32-bit RISC-V target; this limitation is inherent to `wamrc` and is not related to `wasmc`. Despite this limitation, `wamrc` is currently the only WASM off-device AoT compiler targeting 32-bit RISC-V resource-constrained devices, and was therefore used as a baseline in the benchmark.

Benchmark	**AOT (`wasmc` + WAMR) Exec. Time** (ms)	**AOT (`wamrc` + WAMR) Exec. Time** (ms)
`sieve`	20.8	-
`fibonacci`	8.4	6.1
`heapsort`	56.3	-
`matrix-mult`	13.6	-
`lu-decomp`	18.8	-
`0-1-knapsack`	2815.0	-
`floyd-warshall`	124.8	-
`crc32`	3.3	2.9
`edit-distance`	1.7	-
`gauss-elim`	228.3	-

Table 7.4: Execution time of benchmarks between the AOT executable generated by `wasmc` and the AOT executable generated by `wamrc`, both running on the WAMR AOT runtime.

Benchmark	AOT (`wasmc` + WAMR) **Heap Peak** (KB)	AOT (`wamrc` + WAMR) **Heap Peak** (KB)
`sieve`	149.1	-
`fibonacci`	79.4	79.2
`heapsort`	137.2	-
`matrix-mult`	108.6	-
`lu-decomp`	109.3	-
`0-1-knapsack`	137.7	-
`floyd-warshall`	120.9	-
`crc32`	96.3	96.7
`edit-distance`	121.1	-
`gauss-elim`	121.7	-

Table 7.5: Runtime heap peak memory usage of benchmarks between the AOT executable generated by `wasmc` and the AOT executable generated by `wamrc`, both running on the WAMR AOT runtime.

Execution Time For `fibonacci`, `wasmc` produces an executable that runs in 8.4 ms, compared to 6.1 ms for `wamrc`. For `crc32`, `wasmc` takes 3.3 ms against 2.9 ms for `wamrc`. These differences are consistent with the expectation that an optimizing LLVM-based compiler produces faster code than a lightweight compiler that applies no IR optimizations and uses a naive macro expansion instruction selection strategy [2, Secs. 2.1 and 2.2]. The performance gap is modest; however, there is insufficient data to fully evaluate the execution time of AOT executables produced by `wasmc` relative to `wamrc`.

Heap Peak Runtime heap peak at execution time is essentially identical between the two compilers. As discussed in Section 7.3, runtime heap is dominated by the WASM linear memory allocated for the module, which is the same regardless of how the module was compiled.

Answer to Q3 Overall, these results suggest that `wasmc` produces code of reasonable quality relative to a fully optimizing off-device compiler. However, due to the limited support of `wamrc` for the 32-bit RISC-V target, only two benchmarks could be compared, and therefore there is insufficient data to fully evaluate the performance of `wasmc` against the only available off-device WASM AOT compiler targeting this architecture.

Chapter 8

Conclusion

This thesis presented `wasmc`, a lightweight AoT compiler that translates WASM modules into native 32-bit RISC-V machine code directly on a resource-constrained device. `wasmc` is implemented for the `ESP32-C3` microcontroller and demonstrates that AoT compilation of WASM modules entirely on-device is practically feasible within the severe memory constraints of microcontrollers, requiring no off-device toolchains and no heavyweight compiler back end frameworks such as LLVM [16] or Cranelift [5]. The key design decisions that make `wasmc` viable on a resource-constrained device are: function-at-a-time compilation, which bounds peak heap usage to the size of the largest single function rather than the entire module; a memory-efficient single-pass algorithm to construct the CFG in SSA form, adapted from the algorithm of Braun et al. [3]; input-proportional heap allocation that avoids large fixed-size buffers; zero-copy WASM decoding; deliberate omission of an IR optimizer (as WASM modules are typically already produced by optimizing compilers); and a lightweight back end built from scratch implementing linear scan register allocation [19], ABI constraint enforcement, SSA deconstruction via parallel move serialization [23] with critical edge splitting [4, pp. 20-24], and instruction selection via naive macro expansion [2, Secs. 2.1 and 2.2]. A second major contribution of this work is the documentation of the WAMR AOT executable file format. This format is used by the WAMR project [6] for its off-device AoT compiler `wamrc`, but had never been officially documented. The description presented in Chapter 6 was obtained by reverse engineering the open-source WAMR source code and constitutes a self-contained reference for anyone wishing to produce or consume AOT executables compatible with the WAMR AOT runtime. As a practical complement, the `readaot` utility was developed to inspect and debug AOT binaries in a manner analogous to `readelf` or `objdump` for ELF files.

The benchmarks presented in Chapter 7 give affirmative answers to the

three evaluation questions. Regarding compilation feasibility (**Q1**), `wasmc` successfully compiled all ten benchmark programs directly on the `ESP32-C3`, which has only 400 KB of RAM [8, p. 4], with peak heap memory usage never exceeding 16.8 KB and compilation time never exceeding 26.6 ms. Regarding execution performance relative to interpretation (**Q2**), the AOT executables produced by `wasmc` consistently and substantially outperformed the WAMR WASM interpreter, with speedups ranging from approximately 9.8× to 24.2× across all benchmarks, while maintaining essentially identical runtime memory usage. Regarding execution performance relative to off-device compilation (**Q3**), the two benchmarks for which `wamrc` results were available showed a modest performance gap in favor of `wamrc`, consistent with the expectation that an optimizing LLVM-based compiler produces faster code than a lightweight compiler that applies no IR optimizations. However, the limited support of `wamrc` for the 32-bit RISC-V target prevented a full evaluation across all benchmarks.

Several directions remain open for future work. The most immediate is expanding the set of supported WASM instructions: `wasmc` currently implements only the 32-bit integer subset of WASM 1.0 [25], and adding support for 64-bit integers (`i64`) and floating-point types (`f32`, `f64`) would significantly increase the range of WASM modules that `wasmc` can compile. On the code quality side, selective lightweight optimizations, such as constant folding [3, Sec. 3.1] or simple peephole optimization [2, Sec. 2.3], could reduce the performance gap with optimizing off-device compilers without incurring prohibitive memory or time overhead. Furthermore, porting `wasmc` to additional resource-constrained architectures beyond 32-bit RISC-V would broaden its applicability to the wider IoT and embedded systems ecosystem. Another interesting direction for future work is to further reduce the peak heap usage of `wasmc` during the compilation of a WASM module. The data structure that contributes most to the peak memory usage is the CFG of the largest WASM function. Currently, `wasmc` represents the CFG as an explicit graph data structure in memory. An alternative approach would be to eliminate the explicit CFG and store the intermediate representation in a different form that preserves the same information, such as a register-based bytecode representation similar to LLVM Bitcode [15]. LLVM Bitcode is the binary serialization of LLVM IR; similarly, the register-based bytecode would represent a binary serialization of the CFG. This register-based bytecode could be stored in a contiguous buffer, reducing the need for graph-based data structures and potentially lowering peak heap usage. Adopting such a representation would require modifying the back end to operate without an explicit CFG. Currently, the CFG in the back end is primarily used to perform liveness analysis in order to build the live intervals required by the linear-

scan register allocator. However, alternative approaches exist for performing liveness analysis without constructing a CFG [17].

In summary, `wasmc` demonstrates that full AoT compilation of WASM modules on a microcontroller is achievable without sacrificing correctness or execution efficiency, making on-device compilation of WebAssembly feasible even on resource-constrained systems.

Appendix A

`wasmc` WASM 1.0 Implementation Status

This appendix summarizes which instructions and which WASM module sections of the WASM 1.0 specification [25] are currently supported by `wasmc`. The symbol ✓ indicates full support and ✗ indicates that the feature is not yet implemented.

WASM Module Sections

Unimplemented sections are ignored when decoding the WASM binary module.

Section	Supported
`Custom`	✗
`Type`	✓
`Import`	✗
`Function`	✓
`Table`	✓
`Memory`	✓
`Global`	✓
`Export`	✓
`Start`	✗
`Element`	✗
`Code`	✓
`Data`	✓

Types

`wasmc` operates on 32-bit integer values (`i32`) only. The types `i64`, `f32`, and `f64` are not currently supported. As a consequence, any WASM module that uses 64-bit integers or floating-point values cannot be compiled by `wasmc`.

Type	Supported
`i32`	✓
`i64`	✗
`f32`	✗
`f64`	✗

Control Flow Instructions

Instruction	Supported
`unreachable`	✓
`nop`	✓
`block`	✓
`loop`	✓
`if / else`	✓
`end`	✓
`br`	✓
`br_if`	✓
`br_table`	✗
`return`	✓
`call`	✓
`call_indirect`	✗

Parametric Instructions

Instruction	Supported
`drop`	✓
`select`	✓

Variable Instructions

Instruction	Supported
`local.get`	✓
`local.set`	✓
`local.tee`	✓
`global.get`	✓
`global.set`	✓

Memory Instructions

Memory instructions are supported only for `i32` values. The `memory.grow` and `memory.size` instructions are not supported.

Instruction	Supported
`i32.load`	✓
`i64.load`	✗
`f32.load`	✗
`f64.load`	✗
`i32.load8_s`	✓
`i32.load8_u`	✓
`i32.load16_s`	✗
`i32.load16_u`	✗
`i32.load32_s`	✗
`i32.load32_u`	✗
`i32.store`	✓
`i64.store`	✗
`f32.store`	✗
`f64.store`	✗
`i32.store8`	✓
`i32.store16`	✗
`i64.store8`	✗
`i64.store16`	✗
`i64.store32`	✗
`memory.size`	✗
`memory.grow`	✗

Numeric Instructions

Only `i32` numeric instructions are implemented. All the remaining instructions not in the following table are not implemented.

Instruction	Supported
`i32.const`	✓
`i32.eqz`	✓
`i32.eq`	✓
`i32.ne`	✓
`i32.lt_s`	✓
`i32.lt_u`	✓
`i32.gt_s`	✓
`i32.gt_u`	✓
`i32.le_s`	✓
`i32.le_u`	✓
`i32.ge_s`	✓
`i32.ge_u`	✓
`i32.clz`	✗
`i32.ctz`	✗
`i32.popcnt`	✗
`i32.add`	✓
`i32.sub`	✓
`i32.mul`	✓
`i32.div_s`	✓
`i32.div_u`	✓
`i32.rem_s`	✓
`i32.rem_u`	✓
`i32.and`	✓
`i32.or`	✓
`i32.xor`	✓
`i32.shl`	✓
`i32.shr_s`	✓
`i32.shr_u`	✓
`i32.rotl`	✗
`i32.rotr`	✗

Appendix B

`readaot`: A Readelf/Objdump-Like Tool for AOT

`readaot` is a command-line inspection tool for AOT binary files. Analogous to `readelf` or `objdump` for ELF binaries, it parses and displays the structured contents of `.aot` files in a human-readable form. `readaot` was developed to understand the structure and fields of the AOT format described in Chapter 6, and to test and debug `wasmc`. The `readaot` source code is at: `https://github.com/llllluca/wasmc`.

```
readaot <option(s)> path/to/file.aot
```

If no option is provided, `--all` is assumed by default.

`-H, --help`	Display the help message
`-a, --all`	Display all sections (equivalent to `-h -i -f -e -r`)
`-x, --hexdump`	Set display format to raw hexdump
`-p, --print`	Set display format to printable text
`-h, --target-info`	Display the Target Info section
`-i, --init-data`	Display the Init Data section
`-t, --text`	Display the Text section
`-f, --function`	Display the Function section
`-e, --export`	Display the Export section
`-r, --relocation`	Display the Relocation section

The `-x` and `-p` flags act as modifiers: they set the display format for all subsequent section flags on the command line. By default, the Text section is shown as a hexdump and all other sections as printable text.

Bibliography

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Boston, MA: Addison-Wesley, 2006. ISBN: 978-0321486813.

[2] John Blindell. *Instruction Selection: Principles, Methods, and Applications*. New York, NY: Academic Press, 2020.

[3] Matthias Braun et al. "Simple and Efficient Construction of Static Single Assignment Form". In: *Proceedings of the 22nd International Conference on Compiler Construction (CC 2013)*. Vol. 7791. Lecture Notes in Computer Science. Springer, 2013, pp. 102–122. DOI: 10.1007/978-3-642-37051-9_6.

[4] P. Briggs et al. "Practical improvements to the construction and destruction of static single assignment form". In: *ACM SIGPLAN Notices* 33.6 (1998), pp. 1–12.

[5] Bytecode Alliance. *Cranelift: A Fast, Secure Code Generator for WebAssembly*. https://cranelift.dev/. Part of the Wasmtime project. 2024.

[6] Bytecode Alliance. *WebAssembly Micro Runtime (WAMR)*. https://github.com/bytecodealliance/wasm-micro-runtime/. Version v2.4.4. Accessed: 2026-02-20. Bytecode Alliance.

[7] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. 2nd ed. Boston, MA: Morgan Kaufmann, 2012. ISBN: 978-0120884780.

[8] Espressif Systems. *ESP32-C3 Series Datasheet*. https://documentation.espressif.com/esp32-c3_datasheet_en.pdf. Version v2.3. Accessed: 2026-03-13. 2023.

[9] Robbert Gurdeep Singh and Christophe Scholliers. "WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. MPLR 2019. Athens, Greece: Association for Computing Machinery, 2019,

pp. 27–36. ISBN: 9781450369770. DOI: 10.1145/3357390.3361029. URL: https://doi.org/10.1145/3357390.3361029.

[10] Andreas Haas et al. “Bringing the Web up to Speed with WebAssembly”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’17. ACM, 2017, pp. 185–200.

[11] Martin Jacobsson and Jonas Willén. “Virtual Machine Execution for Wearables Based on WebAssembly”. In: *13th EAI International Conference on Body Area Networks*. Ed. by Chika Sugimoto, Hamed Farhadi, and Matti Hämäläinen. Cham: Springer International Publishing, 2020, pp. 381–389. ISBN: 978-3-030-29897-5.

[12] Kito Cheng and Jessica Clarke. *RISC-V ABIs Specification, Document Version 1.0*. Technical Specification Version 1.0. Processor-specific ELF psABI for RISC-V. RISC-V International, Nov. 2022. URL: https://github.com/riscv-non-isa/riscv-elf-psabi-doc/releases/download/v1.0/riscv-abi.pdf.

[13] Borui Li et al. “Bringing WebAssembly to Resource-Constrained IoT Devices for Seamless Device-Cloud Integration”. In: *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys ’22)*. Portland, OR, USA: ACM, 2022, pp. 261–272. DOI: 10.1145/3498361.3538922. URL: https://doi.org/10.1145/3498361.3538922.

[14] Borui Li et al. “ThingSpire OS: a WebAssembly-based IoT operating system for cloud-edge integration”. In: *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys ’21. Virtual Event, Wisconsin: Association for Computing Machinery, 2021, pp. 487–488. ISBN: 9781450384438. DOI: 10.1145/3458864.3466910. URL: https://doi.org/10.1145/3458864.3466910.

[15] LLVM Project. *LLVM Bitcode File Format*. https://llvm.org/docs/BitCodeFormat.html. Online documentation, accessed March 15, 2026.

[16] LLVM Project. *The LLVM Compiler Infrastructure*. https://llvm.org/. Accessed: 2026-03-12. 2024.

[17] Markus Mohnen. “A Graph-Free Approach to Data-Flow Analysis”. In: *Proceedings of the 11th International Conference on Compiler Construction (CC 2002)*. Vol. 2304. Lecture Notes in Computer Science. Springer, 2002, pp. 46–61. DOI: 10.1007/3-540-45937-5_5.

[18] Gregor Peach et al. “eWASM: Practical Software Fault Isolation for Reliable Embedded Devices”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (2020), pp. 3492–3505. DOI: 10.1109/TCAD.2020.3012647.

[19] Massimiliano Poletto and Vivek Sarkar. “Linear Scan Register Allocation”. In: *ACM Trans. Program. Lang. Syst.* 21.5 (1999), pp. 895–913. DOI: 10.1145/330249.330250. URL: https://doi.org/10.1145/330249.330250.

[20] Python Software Foundation. *Python 3 Documentation*. https://docs.python.org/3/. Accessed: 2026-03-12. 2024.

[21] Ricardo Quesada. *Spidermonkey*. https://github.com/ricardoquesada/Spidermonkey. GitHub repository, accessed: 2026-03-12. 2015.

[22] Fabrice Rastello and Florent Bouchez Tichadou, eds. *SSA-based Compiler Design*. PDF/ebook version; accessed 2026-02-20. Cham, Switzerland: Springer International Publishing, 2022. ISBN: 978-3-030-80515-9. DOI: 10.1007/978-3-030-80515-9.

[23] Laurence Rideau, Bernard P. Serpette, and Xavier Leroy. “Tilting at windmills with Coq: Formal verification of a compilation algorithm for parallel moves”. In: *Journal of Automated Reasoning* 40.4 (2008), pp. 307–326. DOI: 10.1007/s10817-007-9096-8. URL: https://xavierleroy.org/publi/parallel-move.pdf.

[24] *RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. https://docs.riscv.org/reference/isa/_attachments/riscv-unprivileged.pdf. Version 20250508; accessed 2026-02-20. RISC-V International, 2025.

[25] Andreas Rossberg, ed. *WebAssembly 1.0 Core Specification*. https://webassembly.github.io/spec/versions/core/WebAssembly-1.0.pdf. Accessed 2026-02-20. WebAssembly Community Group and W3C, 2019.

[26] Tetrate Labs. *wazero: the zero dependency WebAssembly runtime for Go*. https://github.com/tetratelabs/wazero. GitHub repository, accessed: 2026-03-12. 2026.

[27] Benjamin L. Titzer. *Wizard Engine*. https://github.com/titzer/wizard-engine. GitHub repository, accessed: 2026-03-12. 2026.

[28] Omri Traub, Glenn Holloway, and Michael D. Smith. “Quality and Speed in Linear-scan Register Allocation”. In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)*. New York, NY, USA: ACM, 1998, pp. 142–151. DOI: 10.1145/277650.277714.

[29] V8 Project Authors. *V8 JavaScript Engine*. https://github.com/v8/v8. GitHub repository, accessed: 2026-03-12. 2026.

[30] Stefan Wallentowitz, Bastian Kersting, and Dan Mihai Dumitriu. “Potential of WebAssembly for Embedded Systems”. In: *2022 11th Mediterranean Conference on Embedded Computing (MECO)*. 2022, pp. 1–4. DOI: 10.1109/MECO55406.2022.9797106.

[31] Wasm3 Project Contributors. *Wasm3: the fast, lightweight WebAssembly interpreter*. https://github.com/wasm3/wasm3. GitHub repository, accessed: 2026-03-12. 2026.

[32] WasmEdge Project Contributors. *WasmEdge: a lightweight, high-performance WebAssembly runtime*. https://github.com/WasmEdge/WasmEdge. GitHub repository, accessed: 2026-03-12. 2026.

[33] Wasmer Contributors. *Wasmer: The Universal WebAssembly Runtime*. https://github.com/wasmerio/wasmer. Accessed: 2026-03-10. Wasmer Project, 2019.

[34] wasmi-labs Project Contributors. *wasmi: Efficient and Versatile WebAssembly Interpreter*. https://github.com/wasmi-labs/wasmi. GitHub repository, accessed: 2026-03-12. 2026.

[35] Wasmtime Contributors. *Wasmtime: A Lightweight WebAssembly Runtime*. https://github.com/bytecodealliance/wasmtime. Accessed: 2026-03-10. Bytecode Alliance, 2019.

[36] WebAssembly Community Group. *WABT: The WebAssembly Binary Toolkit*. https://github.com/WebAssembly/wabt. Accessed: 2026-03-12. 2026.

[37] WebAssembly Community Group. *WebAssembly Core Specification, Release 3.0 (Draft)*. Ed. by Andreas Rossberg. https://webassembly.github.io/spec/versions/core/WebAssembly-3.0-draft.pdf. Draft specification, accessed 2026-03-12. 2025.

[38] WebAssembly Community Group. *WebAssembly Core Specification, Version 2.0*. Ed. by Andreas Rossberg. https://webassembly.github.io/spec/versions/core/WebAssembly-2.0.pdf. Accessed: 2026-03-12. 2025.

[39] WebKit Project Contributors. *WebKit*. https://github.com/WebKit/WebKit. GitHub repository, accessed: 2026-03-12. 2026.

[40] Elliott Wen and Gerald Weber. "Wasmachine: Bring the Edge up to Speed with A WebAssembly OS". In: *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. 2020, pp. 353–360. DOI: 10.1109/CLOUD49709.2020.00056.

[41] Wikipedia contributors. *Extended Backus–Naur form*. URL: https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form (visited on 02/27/2026).

[42] Christian Wimmer and Michael Franz. "Linear Scan Register Allocation on SSA Form". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. CGO '10. ACM, 2010, pp. 170–179. DOI: 10.1145/1772954.1772979. URL: https://doi.org/10.1145/1772954.1772979.

[43] Xinuos, Inc. *ELF Object File Format, Version 4.2*. Technical Specification Version 4.2. Executable and Linking Format (ELF) specification. Xinuos, Inc., 2025. URL: https://gabi.xinuos.com/v42/elf.pdf.

[44] Yixuan Zhang et al. "Research on WebAssembly Runtimes: A Survey". In: *ACM Trans. Softw. Eng. Methodol.* 34.8 (Oct. 2025). ISSN: 1049-331X. DOI: 10.1145/3714465. URL: https://doi.org/10.1145/3714465.