17-350 / 17-650

Lecture 04: Just-In-Time Compilation

Jonathan Aldrich
Sep 3, 2026

Carnegie Mellon University

S3D Software and Societal Systems Department

Lecture Outline

- From evaluating an AST to generating code from it
- A minimal code generation strategy: one register, one machine stack
- x86-64 refresher
- Key Homework 1 constructs, paired with the assembly they compile to
- Functions, calls, and return: tying compiled code together
- From raw bytes to a callable Rust function
- What this JIT leaves out

From Evaluating to Compiling

- Homework 1: an interpreter recursively walks the AST and **computes** an `i32` value for each node
- Today: a compiler recursively walks the *same* AST, with the *same* structure, but **emits instructions** instead of computing a value
- Same cases, same recursion - only the action per case changes
- By the end of this lecture:
  - `(i32.add (i32.const 1) (i32.const 2))` becomes a handful of x86-64 instructions
  - A whole function becomes a callable native function pointer

A Simple Code Generation Strategy

- WebAssembly's abstract machine is a stack machine: instructions pop operands off an operand stack and push their result
- Simplest mapping to a real CPU:
  - Keep the **top of the operand stack in one fixed register**, `eax`
  - Spill everything else onto the **real machine call stack**
- Every binary operator has the same shape:
  i. compile the left operand (result in `eax`)
  ii. `push rax` to save it
  iii. compile the right operand (result in `eax`)
  iv. `pop rcx` to recover the left operand
  v. combine `eax` and `ecx`, result stays in `eax`
- Not a production register allocator - but simple, uniform, and enough

Just Enough x86-64 - A Refresher

- `rax` / `eax` are the *same* physical register; `eax` is its low 32 bits (same for `rdi` / `edi`, `rcx` / `ecx`, `r12` / `r12d`)
- Instruction order: `mov dest, src` (Intel syntax, used by `dynasm`)
  - Opposite of AT&T syntax; same order as ARM, but different mnemonics elsewhere
- The machine stack grows **down**:
  - `push` decrements `rsp`, then stores
  - `pop` loads, then increments `rsp`
  - `call` pushes a return address; `ret` pops it and jumps there
- Calling convention (System V x86-64): first argument arrives in `rdi`
  - Same idea as ARM's AAPCS64 passing the first argument in `x0`
- `r12` is **callee-saved**: a function that uses `r12` must restore it before returning

`i32.const n`

```
(i32.const 5)
```

```
mov eax, 5
```

- A constant loads an immediate directly into the top-of-stack register
- No stack traffic at all - the simplest possible construct

`local.get $name`

```
(local.get $n)
```

```
mov eax, [r12 + offset]
```

- Parameters live in a small array of 8-byte slots
- A pointer to that array is kept in `r12` for the whole function
- `offset` is the parameter's position times 8, computed **once at compile time**
- Same idea as Lecture 2's "name lookup becomes slot indexing" - just baked into the generated code instead of computed at run time

Binary Operators: `i32.add`

```
(i32.add (local.get $x) (i32.const 1))
```

```
mov eax, [r12 + 0]          ; compile left operand
push rax                    ; save it
mov eax, 1                  ; compile right operand
pop rcx                     ; recover left operand
add eax, ecx                ; combine, result in eax
```

- This is a concrete version of the push/pop pattern from the strategy slide

Binary Operators: The Rest

```
; i32.sub: WASM order is (first - second)
sub ecx, eax
mov eax, ecx

; i32.mul
imul eax, ecx

; i32.rem_s
xchg eax, ecx   ; dividend into eax
cdq             ; sign-extend eax into edx:eax
idiv ecx        ; quotient in eax, remainder in edx
mov eax, edx    ; we want the remainder

; i32.lt_s
cmp ecx, eax
setl al         ; set low byte to 1/0 from flags
movzx eax, al   ; zero-extend byte back to 32 bits
```

`if (result i32) cond (then ...) (else ...)`

```
(if (result i32) (i32.lt_s (local.get $n) (i32.const 2))
  (then (local.get $n))
  (else (i32.const 0)))
```

```
; ... compile cond, result in eax ...
test eax, eax
jz else_label
; ... compile then branch ...
jmp end_label
else_label:
; ... compile else branch ...
end_label:
```

- Same control-flow shape as compiled `if` / `else` in C: two forward labels

Function Declaration (Prologue)

```
(func $fib (param $n i32) (result i32) ...)
```

```
fib_entry:
  push r12        ; save caller's r12
  mov r12, rdi    ; adopt incoming argument pointer as our frame base
  ; ... compiled body ...
```

- Every function starts with a label - its entry point
- `rdi` carries a pointer to this call's argument array (System V convention)
- Adopting it into `r12` is exactly what `local.get` later reads from

Return (Epilogue)

```
; ... body left its result in eax ...
pop r12   ; restore caller's r12
ret       ; pop return address (pushed by call), jump to it
```

- No explicit `return` instruction in this WAT subset
- A function's value is simply whatever is left in `eax` when its body finishes
- The epilogue undoes exactly what the prologue did, in reverse

`call $name arg1 arg2 ...`

```
(call $fib (i32.sub (local.get $n) (i32.const 1)))
```

```
; ... compile each argument, push rax for each ...
sub rsp, 8         ; only if needed, to keep 16-byte alignment
mov rdi, rsp        ; pointer to the pushed arguments
call fib_entry
add rsp, 8          ; discard pushed args (+ any padding)
```

- Arguments are pushed in order so they end up contiguous on the stack
- x86-64 requires 16-byte stack alignment at `call`
- This is a **native call instruction** - recursive WAT calls become ordinary recursive machine calls, using the machine's own call stack

From Bytes to a Callable Function

- `dynasm!` (from the `dynasm` crate): write `mov` / `push` / `call` /labels directly in Rust source, assemble to raw machine code bytes

```
let otherwise = assembler.new_dynamic_label();
let end = assembler.new_dynamic_label();
dynasm!(assembler ; test eax, eax ; jz =>otherwise);
// ... compile the `then` branch ...
dynasm!(assembler ; jmp =>end ; =>otherwise);
// ... compile the `else` branch ...
dynasm!(assembler ; =>end);
```

- Labels like `=>otherwise` are exactly the `then` / `else` / `end` /entry labels from the last several slides
- Assembled bytes live in an `ExecutableBuffer`: memory explicitly marked executable by the OS
  - Ordinary heap memory is **not** allowed to run as code - this is the unsafe, OS-level step

`mem::transmute` to a Function Pointer

```
let entry_ptr = code.ptr(entry_offset);
let compiled: extern "C" fn(*const i32) -> i32 =
    unsafe { mem::transmute(entry_ptr) };
let result = compiled(args.as_ptr());
```

- Ties back to Lecture 3's `unsafe` theme:
  - **Hazards**: wrong calling convention, jumping into non-code bytes, a non-executable buffer (OSes enforce W^X - calling into ordinary writable memory faults, or worse, is a code-injection vector)
  - **Invariant**: the bytes at `entry` implement a function matching this signature, following the calling convention we just walked through, and live in memory the OS has marked executable
  - **Establish/Maintain**: `finalize()` is the *only* way to obtain an `ExecutableBuffer` (it `mmap`s a fresh executable page); every transmuted pointer comes from that buffer, never a raw `Vec<u8>`

What This JIT Leaves Out

- No register allocation - only one value lives in a register at a time
- No optimization passes - no constant folding, no removing redundant push/pop
- Single architecture - x86-64 only, no portable intermediate form
- Whole-function compilation every time - no tiering, no inline caching
- Still: this already shows the core idea in full
  - compilation = tree walk that emits instructions instead of computing values
  - generated code runs at native speed, with **no per-instruction dispatch**

Wrap-Up

- Compiling is structurally the same recursive walk as interpreting - only the action per AST case changes
- One register + the machine stack is enough to compile a small stack-based language
- The `unsafe` boundary is small and specific: producing correct bytes, then trusting `mem::transmute`
- In Homework 2 you will build a threaded interpreter or JIT, following these ideas