Custom x64 Backend Implementation Resources

Overview: Jai-Style Fast Debug Compilation

Two compiler backends exist: an x64 and an LLVM backend. The x64 backend was developed from scratch by J. Blow and his compiler team. It converts the internal byte-code to x64 machine code. It does fast but naive code generation, without any code optimization. It is intended to be used during the development phase

Target Architecture:

```
Your Language Source → Internal IR bytecode → **x64 machine code directly**
```

Key Goals:

- Fast compilation (1 million+ lines per second)
- Naive but functional code generation
- No optimization (debug builds only)
- Direct machine code emission (no C backend)

Core Resources

1. Tsoding's Projects (Essential Examples)

Porth Compiler

- **Repository:** https://github.com/tsoding/porth
- **Key Features:** Compilation generates assembly code, compiles it with nasm, and then links it with GNU ld
- **Performance:** Building a 263 byte Hello World program in 3 milliseconds: The compilation speed in this mode is about 1 million lines per second
- **Approach:** Python-based stack language that generates x64 assembly

B Compiler in Crust

- **Repository:** https://github.com/bext-lang/b
- **Key Features:** the x86_64 and aarch64 targets generate assembly and pass it to cc to assemble and link
- **Rust-like syntax with direct assembly generation

Jai Projects

- **jaibreak:** https://github.com/tsoding/jaibreak - Game showcasing Jai compilation
- **jai-wasm:** https://github.com/tsoding/jai-wasm - WebAssembly compilation proof-of-concept

2. FASM (Flat Assembler) - Your Primary Tool

Why FASM is Perfect for This:

FASM (flat assembler) is an assembler for x86 processors. It supports Intel-style assembly language on the IA-32 and x86-64 computer architectures. It claims high speed, size optimizations, operating system (OS) portability, and macro abilities

It can produce output in plain binary, MZ, PE, COFF or ELF format. It includes a powerful but simple macroinstruction system and does multiple passes to optimize the size of instruction codes

FASM Resources:

- **Official Site:** https://flatassembler.net/
- **Documentation:** https://flatassembler.net/docs.php
- **GitHub Mirror:** https://github.com/tgrysztar/fasm
- **Linux Examples:** https://github.com/hnwfs/lin-Fasm

FASM Tutorial (Essential Read):

"Let's Learn x86-64 Assembly! Part 1 - Metaprogramming in Flat Assembler"

- **URL:** https://gpfault.net/posts/asm-tut-1.txt.html
- **Key Topics:**
  - Macro system for high-level constructs
  - Assembly-time variables and conditional assembly
  - Windows API calling conventions
  - PE file generation

Key FASM Features for Code Generation:

```asm
; Assembly-time variables
arg_count = 0
macro count_args [arg] {
    arg_count = arg_count + 1
}

; Conditional assembly
if arg_count > 9
    display "Too many arguments!"
    err
end if

; Win64 calling convention macro
macro call64 fn*, [arg] {
    ; Automatic register assignment and stack alignment
    ; Implementation handles RCX, RDX, R8, R9, then stack
}
```

3. X64 Instruction Encoding References

Intel Manuals (Essential)

- **Intel® 64 and IA-32 Architectures Software Developer Manuals**
- **URL:** https://software.intel.com/en-us/articles/intel-sdm
- **Volume 2:** Instruction Set Reference (most important)

Practical Encoding Guide

"Notes on x86-64 Assembly and Machine Code"

- **URL:** https://gist.github.com/mikesmullin/6259449
- **Covers:** Complete instruction encoding, ModR/M bytes, SIB bytes, displacement

Key Instruction Structure:

```
┌─────────┬─────────┬─────────┬─────────┬─────────┬─────────┐
│ Prefix  │ Opcode  │ModR/M   │  SIB  │  Disp   │   Imm   │
│ 0-4 B   │ 1-3 B   │ 0-1 B   │ 0-1 B  │0,1,2,4B │0,1,2,4,8│
└─────────┴─────────┴─────────┴─────────┴─────────┴─────────┘
```

Online Tools for Learning:

- **x86 Instruction Assembler:** https://defuse.ca/online-x86-assembler.htm
- **Instruction Reference:** http://www.felixcloutier.com/x86/
- **Opcode Reference:** http://ref.x86asm.net/

4. Machine Code Generation Libraries

AsmJit (C++ Reference Implementation)

- **Repository:** https://github.com/asmjit/asmjit
- **Website:** https://asmjit.com/
- **Key Features:** Low-latency machine code generation and execution. However, AsmJit evolved and now contains features that are far beyond the initial scope

Example AsmJit Usage:

```cpp
// High-level approach similar to what you want
CodeHolder code;
x86::Assembler a(&code);

a.mov(x86::eax, 1);          // mov eax, 1
a.mov(x86::ebx, 2);          // mov ebx, 2
a.add(x86::eax, x86::ebx);   // add eax, ebx
a.ret();                     // ret
```

5. Implementation Strategy

Phase 1: Basic Code Generation

```c
// Your IR to x64 mapping
typedef struct {
    uint8_t* code_buffer;
    size_t code_size;
    size_t capacity;
} X64Generator;

// Example: Generate "mov rax, immediate"
void emit_mov_rax_imm64(X64Generator* gen, uint64_t value) {
    // REX.W prefix for 64-bit operand
    emit_byte(gen, 0x48);
    // MOV r64, imm64 opcode
    emit_byte(gen, 0xB8);
    // 64-bit immediate value (little-endian)
    emit_qword(gen, value);
}
```

Phase 2: FASM Integration

```c
// Generate FASM assembly text
void generate_fasm_output(AST_Node* program, FILE* output) {
    fprintf(output, "format PE64 NX GUI 6.0\n");
    fprintf(output, "entry start\n\n");

    // Your IR to FASM translation
    generate_procedures(program, output);
    generate_data_section(output);
    generate_import_section(output);
}

// Compile with FASM
int compile_with_fasm(const char* asm_file, const char* exe_file) {
    char cmd[1024];
    snprintf(cmd, sizeof(cmd), "fasm \"%s\" \"%s\"", asm_file, exe_file);
    return system(cmd);
}
```

6. Real-World Examples and Tutorials

Simple JIT Implementation Examples:

- **V Language x64 Backend:** Referenced in https://github.com/vlang/v/issues/2849
- **Writing x86-64 Tutorial:** https://nickdesaulniers.github.io/blog/2014/04/18/lets-write-some-x86-64/

Educational Resources:

- **x86-64 Cheat Sheet:** https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf
- **Assembly Programming Tutorial:** Key registers, calling conventions, stack frame management

7. Development Workflow

Recommended Approach:

1. **Start with FASM text generation** (easier debugging)
2. **Parse your IR and emit FASM assembly**
3. **Use FASM to generate executable**
4. **Later optimize with direct machine code emission**

Testing Strategy:

```bash
# Compile your test program
./your_compiler test.your_lang --target=x64-debug

# Verify output
./test.exe

# Debug with objdump if needed
objdump -d test.exe
```

8. Essential Code Examples

Basic Function Prologue/Epilogue:

```asm
; Function entry
push rbp
mov rbp, rsp
sub rsp, 32        ; Local variables space

; Function exit
mov rsp, rbp
pop rbp
ret
```

Win64 Calling Convention:

```asm
; First 4 args: RCX, RDX, R8, R9
; Additional args: stack (right-to-left)
; Return value: RAX
; Caller cleanup, 16-byte stack alignment
```

9. Advanced Features (Later Implementation)

Debug Information:

- Generate .pdb files for Visual Studio debugging
- Line number mapping from source to assembly
- Variable name preservation

Optimization Opportunities:

- Register allocation (even naive)
- Dead code elimination
- Constant folding in code generation

10. Platform-Specific Considerations

Windows (PE Format):

- Import tables for API calls
- Section organization (.text, .data, .rdata)
- Exception handling setup

Linux (ELF Format):

- Symbol tables
- Relocation entries
- Dynamic linking considerations

Quick Start Implementation Plan

1. **Day 1-2:** Study FASM tutorial and x64 encoding guides
2. **Day 3-4:** Implement basic IR to FASM text translation
3. **Day 5-6:** Add function calls, stack management
4. **Day 7:** Test with real programs, debug output
5. **Week 2:** Optimize and add more language features

Key Takeaways

- **Start simple:** Text-based FASM generation first
- **Focus on correctness:** No optimization, just working code
- **Test incrementally:** Small programs to full compiler self-hosting
- **Use references:** Intel manuals and existing implementations
- **Leverage FASM:** Its macro system can handle complex patterns

This approach mirrors Jai's philosophy: fast compilation for development cycles, with the option to switch to optimized LLVM backend for release builds.