Hybrid C + Direct FASM Backend (ELF64 Linux)

Table of Contents

Architecture Overview

The Optimal Hybrid Pipeline

```
┌─────────────┐    ┌─────────────┐    ┌─────────────┐    ┌─────────────┐
│ Your Source │ -> │ C Compiler  │ -> │ FASM Macro  │ -> │ ELF64       │
│   Code      │    │  Frontend   │    │  Assembly   │    │  Executable │
│             │    │ (Parse/IR)  │    │  (Direct)   │    │   Binary    │
└─────────────┘    └─────────────┘    └─────────────┘    └─────────────┘
                          ^                  ^                  ^
                          │                  │                  │
                   AST Processing       FASM Magic        Native Code
                   Symbol Analysis      Metaprogramming   OS Integration
                   Register Planning    Code Templates    System Calls
```

Why This Combination is Optimal

C Frontend Advantages:

- Complex parsing and AST manipulation
- Cross-platform logic and data structures
- Integration with existing codebase
- Sophisticated analysis passes

Direct FASM Backend Advantages:

- Incredibly powerful macro system
- Assembly-time computation and code generation
- Direct x64 instruction control
- Built-in ELF64 output support
- No intermediate files or parsing overhead

FASM Metaprogramming for Code Generation

Core FASM Infrastructure

asm

```
; runtime.asm - Core runtime macros for your compiler
format ELF64 executable 3
entry start

; Assembly-time variables for code generation
current_function_locals = 0
stack_depth = 0
label_counter = 0
temp_register = 0

; Register allocation table
rax_used = 0
rbx_used = 0
rcx_used = 0
rdx_used = 0
rsi_used = 0
rdi_used = 0
r8_used = 0
r9_used = 0
r10_used = 0
r11_used = 0
r12_used = 0
r13_used = 0
r14_used = 0
r15_used = 0

; Powerful macro for automatic register allocation
macro allocate_reg {
    if rax_used = 0
        rax_used = 1
        equ allocated_reg rax
    else if rbx_used = 0
        rbx_used = 1
        equ allocated_reg rbx
    else if rcx_used = 0
        rcx_used = 1
        equ allocated_reg rcx
    else if rdx_used = 0
        rdx_used = 1
        equ allocated_reg rdx
    else if rsi_used = 0
        rsi_used = 1
        equ allocated_reg rsi
    else if rdi_used = 0
```

```
        rdi_used = 1
        equ allocated_reg rdi
    else if r8_used = 0
        r8_used = 1
        equ allocated_reg r8
    else if r9_used = 0
        r9_used = 1
        equ allocated_reg r9
    else if r10_used = 0
        r10_used = 1
        equ allocated_reg r10
    else if r11_used = 0
        r11_used = 1
        equ allocated_reg r11
    else
        display 'ERROR: No registers available!'
        err
    end if
}

; Release register
macro release_reg reg {
    if reg eq rax
        rax_used = 0
    else if reg eq rbx
        rbx_used = 0
    else if reg eq rcx
        rcx_used = 0
    else if reg eq rdx
        rdx_used = 0
    else if reg eq rsi
        rsi_used = 0
    else if reg eq rdi
        rdi_used = 0
    else if reg eq r8
        r8_used = 0
    else if reg eq r9
        r9_used = 0
    else if reg eq r10
        r10_used = 0
    else if reg eq r11
        r11_used = 0
    end if
}

; Function prologue with automatic stack calculation
macro func_begin name, [local_vars] {
```

```
    common
    ; Count local variables
    local locals_count, locals_size
    locals_count = 0

    forward
    locals_count = locals_count + 1

    common
    locals_size = locals_count * 8
    ; Align to 16 bytes
    locals_size = (locals_size + 15) and not 15
    current_function_locals = locals_size

    name:
    push rbp
    mov rbp, rsp
    if locals_size > 0
        sub rsp, locals_size
    end if

    ; Clear all register allocation
    rax_used = 0
    rbx_used = 0
    rcx_used = 0
    rdx_used = 0
    rsi_used = 0
    rdi_used = 0
    r8_used = 0
    r9_used = 0
    r10_used = 0
    r11_used = 0
    r12_used = 0
    r13_used = 0
    r14_used = 0
    r15_used = 0
}

macro func_end {
    mov rsp, rbp
    pop rbp
    ret
}

; Smart arithmetic operations with automatic register management
macro smart_add dest, src1, src2 {
```

```
    if dest eq src1
        if src2 eqtype 0
            add dest, src2
        else
            add dest, src2
        end if
    else
        mov dest, src1
        if src2 eqtype 0
            add dest, src2
        else
            add dest, src2
        end if
    end if
}

macro smart_sub dest, src1, src2 {
    if dest eq src1
        if src2 eqtype 0
            sub dest, src2
        else
            sub dest, src2
        end if
    else
        mov dest, src1
        if src2 eqtype 0
            sub dest, src2
        else
            sub dest, src2
        end if
    end if
}

macro smart_mul dest, src1, src2 {
    if dest eq src1
        if src2 eqtype 0
            imul dest, src2
        else
            imul dest, src2
        end if
    else
        mov dest, src1
        if src2 eqtype 0
            imul dest, src2
        else
            imul dest, src2
        end if
```

```
        end if
    end if
}

; Generate unique labels
macro gen_label prefix {
    label_counter = label_counter + 1
    label_name equ prefix#label_counter
}

; Conditional code generation
macro if_compile condition {
    if condition
        macro endif \{ \}
    else
        macro endif \{
        \}
        endif
        macro endif \\{ \\}
    end if
}

; Advanced loop generation with automatic label management
macro compile_loop init_code, condition_code, body_code, increment_code {
    gen_label loop_start
    gen_label loop_end

    init_code

    label_name:
    condition_code
    jz loop_end

    body_code

    increment_code
    jmp label_name

    loop_end:
}
```

Advanced Code Generation Macros

asm

```
; Dynamic function call generation
macro dyn_call func_name, [args] {
    common
    local arg_count, stack_space, current_arg
    arg_count = 0

    ; Count arguments
    forward
    arg_count = arg_count + 1

    common
    ; Calculate stack space (System V ABI for Linux)
    ; First 6 args in registers: RDI, RSI, RDX, RCX, R8, R9
    if arg_count > 6
        stack_space = (arg_count - 6) * 8
        ; Align stack to 16 bytes
        if stack_space mod 16 <> 0
            stack_space = stack_space + (16 - stack_space mod 16)
        end if
        sub rsp, stack_space
    end if

    current_arg = 0

    ; Place arguments in correct locations
    forward
    current_arg = current_arg + 1

    if current_arg = 1
        if args eqtype 0
            mov rdi, args
        else
            mov rdi, args
        end if
    else if current_arg = 2
        if args eqtype 0
            mov rsi, args
        else
            mov rsi, args
        end if
    else if current_arg = 3
        if args eqtype 0
            mov rdx, args
        else
```

```
            mov rdx, args
        end if
    else if current_arg = 4
        if args eqtype 0
            mov rcx, args
        else
            mov rcx, args
        end if
    else if current_arg = 5
        if args eqtype 0
            mov r8, args
        else
            mov r8, args
        end if
    else if current_arg = 6
        if args eqtype 0
            mov r9, args
        else
            mov r9, args
        end if
    else
        ; Stack arguments (in reverse order)
        if args eqtype 0
            mov qword [rsp + (current_arg - 7) * 8], args
        else
            push args
            pop qword [rsp + (current_arg - 7) * 8]
        end if
    end if

    common
    call func_name

    ; Clean up stack
    if arg_count > 6
        add rsp, stack_space
    end if
}

; High-level expression compilation
macro compile_expr dest, expr {
    match a + b, expr \{
        allocate_reg
        compile_expr allocated_reg, a
        push allocated_reg
        release_reg allocated_reg
```

```
        allocate_reg
        compile_expr allocated_reg, b
        pop dest
        add dest, allocated_reg
        release_reg allocated_reg
    \}

    match a - b, expr \{
        allocate_reg
        compile_expr allocated_reg, a
        push allocated_reg
        release_reg allocated_reg

        allocate_reg
        compile_expr allocated_reg, b
        pop dest
        sub dest, allocated_reg
        release_reg allocated_reg
    \}

    match a * b, expr \{
        allocate_reg
        compile_expr allocated_reg, a
        push allocated_reg
        release_reg allocated_reg

        allocate_reg
        compile_expr allocated_reg, b
        pop dest
        imul dest, allocated_reg
        release_reg allocated_reg
    \}

    match =var name, expr \{
        ; Load variable from stack frame
        mov dest, [rbp - name#_offset]
    \}

    match num, expr \{
        if num eqtype 0
            mov dest, num
        end if
    \}
}

; String literal handling with automatic data section
```

```
strlit_count = 0
macro string_literal text {
   strlit_count = strlit_count + 1
   section '.rodata'
   str#strlit_count db text, 0
   section '.text'
   equ string_address str#strlit_count
}

; Pattern matching for complex code generation
macro match_pattern pattern, code, [alternatives] {
   common
   local matched
   matched = 0

   forward
   match pattern, alternatives \{
      if matched = 0
         matched = 1
         code
      end if
   \}

   common
   if matched = 0
      display 'No pattern matched for: ', `alternatives
      err
   end if
}

; Automatic variable allocation on stack
var_offset = 0
macro declare_var name, size {
   var_offset = var_offset + size
   name#_offset = var_offset
   name#_size = size
}

; Smart memory operations
macro load_var dest, var_name {
   mov dest, [rbp - var_name#_offset]
}

macro store_var var_name, src {
   mov [rbp - var_name#_offset], src
}
```

```
; Conditional compilation based on optimization level
OPTIMIZATION_LEVEL = 0  ; Can be set by C frontend

macro opt_code opt_level, code {
    if OPTIMIZATION_LEVEL >= opt_level
        code
    end if
}

; Debug information insertion
DEBUG_MODE = 1

macro debug_info info {
    if DEBUG_MODE = 1
        ; Insert debug comment
        ; In real implementation, this could generate DWARF info
    end if
}

; Loop optimization patterns
macro optimize_loop counter, limit, body {
    ; Check if loop can be unrolled
    if limit - counter <= 4
        ; Unroll small loops
        repeat limit - counter
            body
        end repeat
    else
        ; Use regular loop
        compile_loop \
            mov counter, counter, \
            cmp counter, limit, \
            body, \
            inc counter
    end if
}
```

C-to-FASM Interface

C Frontend that Generates FASM

```c
// fasm_generator.h
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    FILE *output;
    int optimization_level;
    bool debug_mode;
    int current_scope_depth;
    int temp_var_counter;
} FASMCodeGen;

// Initialize FASM code generator
FASMCodeGen* fasm_codegen_create(const char *output_file) {
    FASMCodeGen *gen = malloc(sizeof(FASMCodeGen));
    gen->output = fopen(output_file, "w");
    gen->optimization_level = 0;
    gen->debug_mode = true;
    gen->current_scope_depth = 0;
    gen->temp_var_counter = 0;

    // Include the macro library
    fprintf(gen->output, "include 'runtime.asm'\n\n");

    // Set compilation flags
    fprintf(gen->output, "OPTIMIZATION_LEVEL = %d\n", gen->optimization_level);
    fprintf(gen->output, "DEBUG_MODE = %d\n", gen->debug_mode ? 1 : 0);
    fprintf(gen->output, "\n");

    return gen;
}

// Generate FASM function from AST
void generate_fasm_function(FASMCodeGen *gen, AST_Node *func_node) {
    fprintf(gen->output, "; Function: %s\n", func_node->name);

    // Count local variables
    int local_count = count_local_variables(func_node);

    // Generate function with automatic variable allocation
    fprintf(gen->output, "func_begin %s", func_node->name);

    // Declare local variables
    for (int i = 0; i < local_count; i++) {
```

```c
    for (int i = 0; i < local_count; i++) {
        fprintf(gen->output, ", local_var_%d", i);
    }
    fprintf(gen->output, "\n");

    // Generate function body
    if (func_node->body) {
        generate_fasm_statement_block(gen, func_node->body);
    }

    fprintf(gen->output, "func_end\n\n");
}

// Generate binary operation using smart macros
void generate_fasm_binop(FASMCodeGen *gen, AST_Node *binop) {
    char temp_var[64];
    snprintf(temp_var, sizeof(temp_var), "temp_%d", gen->temp_var_counter++);

    fprintf(gen->output, "    ; Binary operation: %s\n", binop->name);
    fprintf(gen->output, "    allocate_reg\n");

    // Generate left operand
    generate_fasm_expression(gen, binop->left);
    fprintf(gen->output, "    push allocated_reg\n");
    fprintf(gen->output, "    release_reg allocated_reg\n");

    // Generate right operand
    fprintf(gen->output, "    allocate_reg\n");
    generate_fasm_expression(gen, binop->right);

    // Perform operation
    if (strcmp(binop->name, "+") == 0) {
        fprintf(gen->output, "    pop rax\n");
        fprintf(gen->output, "    smart_add allocated_reg, rax, allocated_reg\n");
    } else if (strcmp(binop->name, "-") == 0) {
        fprintf(gen->output, "    pop rax\n");
        fprintf(gen->output, "    smart_sub allocated_reg, rax, allocated_reg\n");
    } else if (strcmp(binop->name, "*") == 0) {
        fprintf(gen->output, "    pop rax\n");
        fprintf(gen->output, "    smart_mul allocated_reg, rax, allocated_reg\n");
    }

    fprintf(gen->output, "    ; Result in allocated_reg\n");
}

// Generate complex expressions using FASM pattern matching
void generate_fasm_expression(FASMCodeGen *gen, AST_Node *expr) {
```

```
    switch (expr->type) {
        case AST_NUM:
            fprintf(gen->output, "    mov allocated_reg, %lld\n", expr->num);
            break;

        case AST_ID:
            fprintf(gen->output, "    load_var allocated_reg, %s\n", expr->name);
            break;

        case AST_BIN_OP:
            generate_fasm_binop(gen, expr);
            break;

        case AST_CALL:
            generate_fasm_function_call(gen, expr);
            break;
    }
}

// Generate function call using dynamic call macro
void generate_fasm_function_call(FASMCodeGen *gen, AST_Node *call_node) {
    fprintf(gen->output, "    ; Function call: %s\n", call_node->name);
    fprintf(gen->output, "    dyn_call %s", call_node->name);

    // Add arguments
    for (size_t i = 0; i < call_node->children.used; i++) {
        fprintf(gen->output, ", ");

        AST_Node *arg = call_node->children.data[i];
        if (arg->type == AST_NUM) {
            fprintf(gen->output, "%lld", arg->num);
        } else if (arg->type == AST_ID) {
            fprintf(gen->output, "[rbp - %s_offset]", arg->name);
        } else {
            // For complex expressions, we'd need temporary storage
            generate_fasm_expression(gen, arg);
            fprintf(gen->output, "allocated_reg");
        }
    }

    fprintf(gen->output, "\n");
    fprintf(gen->output, "    ; Result in rax\n");
}

// Generate assignment with smart variable handling
void generate_fasm_assignment(FASMCodeGen *gen, AST_Node *assign_node) {
```

```
    fprintf(gen->output, "    ; Assignment: %s\n", assign_node->name);

    // Generate right-hand side
    fprintf(gen->output, "    allocate_reg\n");
    generate_fasm_expression(gen, assign_node->right);

    // Store in variable
    fprintf(gen->output, "    store_var %s, allocated_reg\n", assign_node->name);
    fprintf(gen->output, "    release_reg allocated_reg\n");
}

// Main program generation
void generate_fasm_program(FASMCodeGen *gen, AST_Node *program) {
    fprintf(gen->output, "format ELF64 executable 3\n");
    fprintf(gen->output, "entry start\n\n");

    // Generate all functions
    for (size_t i = 0; i < program->children.used; i++) {
        AST_Node *child = program->children.data[i];
        if (child->type == AST_PROC) {
            generate_fasm_function(gen, child);
        }
    }

    // Generate main entry point
    fprintf(gen->output, "start:\n");
    fprintf(gen->output, "    ; Program entry\n");
    fprintf(gen->output, "    dyn_call main\n");
    fprintf(gen->output, "    ; Exit program\n");
    fprintf(gen->output, "    mov rax, 60    ; sys_exit\n");
    fprintf(gen->output, "    mov rdi, 0     ; exit status\n");
    fprintf(gen->output, "    syscall\n");
}

// Complete compilation pipeline
int compile_with_fasm_backend(const char *source_file, const char *output_exe) {
    // Parse source
    AST_Node *program = parse_file(source_file);
    if (!program) return -1;

    // Generate FASM assembly
    char asm_file[256];
    snprintf(asm_file, sizeof(asm_file), "%s.asm", source_file);

    FASMCodeGen *gen = fasm_codegen_create(asm_file);
    generate_fasm_program(gen, program);
    fclose(gen->output);
```

```c
    fclose(gen->output);

    // Compile with FASM
    char cmd[512];
    snprintf(cmd, sizeof(cmd), "fasm %s %s", asm_file, output_exe);

    int result = system(cmd);

    // Cleanup
    if (result == 0) {
        unlink(asm_file);  // Remove intermediate file
        chmod(output_exe, 0755);  // Make executable
        printf("Successfully compiled %s\n", output_exe);
    } else {
        printf("FASM compilation failed. Assembly saved as %s\n", asm_file);
    }

    free(gen);
    ast_destroy(program);
    return result;
}
```

Advanced FASM Macros

Domain-Specific Language Creation

asm

```asm
; Create a high-level language directly in FASM
; fibonacci.lang compiled to fibonacci.asm

; Define language constructs
macro FUNCTION name, [params] {
    func_begin name
    forward
        declare_var params, 8
    common
}

macro END_FUNCTION {
    func_end
}

macro IF condition {
    gen_label if_false
    gen_label if_end

    ; Evaluate condition
    compile_expr rax, condition
    test rax, rax
    jz if_false

    macro ELSE \\{
        jmp if_end
        if_false:
    \\}

    macro END_IF \\{
        if_false:
        if_end:
    \\}
}

macro WHILE condition {
    gen_label while_start
    gen_label while_end

    while_start:
    compile_expr rax, condition
    test rax, rax
    jz while_end

    macro END_WHILE \\{
```

```
    macro END_WHILE \\{
        jmp while_start
        while_end:
    \\}
}

macro RETURN expr {
    compile_expr rax, expr
    jmp func_end  ; Will be resolved by func_begin/func_end
}

macro VAR name, value {
    declare_var name, 8
    if value eqtype 0
        store_var name, value
    else
        allocate_reg
        compile_expr allocated_reg, value
        store_var name, allocated_reg
        release_reg allocated_reg
    end if
}

macro CALL func_name, [args] {
    dyn_call func_name, args
}

; Example usage of the DSL:
FUNCTION fibonacci, n
    IF var n <= 1
        RETURN var n
    END_IF

    VAR temp1, CALL(fibonacci, var n - 1)
    VAR temp2, CALL(fibonacci, var n - 2)
    RETURN var temp1 + var temp2
END_FUNCTION

FUNCTION main
    VAR result, CALL(fibonacci, 10)
    ; Print result somehow...
    RETURN 0
END_FUNCTION
```

Optimization Macros

```asm
; Peephole optimization patterns
macro optimize_mov_sequences [instructions] {
    common
    local opt_done
    opt_done = 0

    forward
    match mov reg1=,val1 mov reg2=,val2, instructions \{
        if reg1 eq reg2 & opt_done = 0
            ; Remove redundant second mov
            mov reg1, val1
            opt_done = 1
        end if
    \}

    match mov reg1=,reg2 add reg1=,val, instructions \{
        if opt_done = 0
            ; Optimize to lea
            lea reg1, [reg2 + val]
            opt_done = 1
        end if
    \}

    if opt_done = 0
        ; No optimization applied, emit original
        instructions
    end if
}

; Loop unrolling
macro unroll_loop count, body {
    if count <= 8
        repeat count
            body
        end repeat
    else
        ; Use regular loop for large counts
        mov rcx, count
        .loop_start:
        body
        loop .loop_start
    end if
}

; Inline function calls for small functions
```

```
; Inline function calls for small functions
inline_threshold = 20  ; Maximum instructions to inline

macro maybe_inline func_name, [args] {
    local func_size
    func_size = func_name#_size  ; Set by func_begin macro

    if func_size <= inline_threshold
        ; Inline the function
        display 'Inlining function: ', `func_name

        ; Set up parameters
        forward
            ; Handle parameter passing inline
        common

        ; Insert function body directly
        include func_name#_body
    else
        ; Regular function call
        dyn_call func_name, args
    end if
}

; Constant folding at assembly time
macro const_fold operation {
    match a + b, operation \{
        if a eqtype 0 & b eqtype 0
            result = a + b
        else
            add a, b
            equ result a
        end if
    \}

    match a * b, operation \{
        if a eqtype 0 & b eqtype 0
            result = a * b
        else
            imul a, b
            equ result a
        end if
    \}
}

; Smart instruction selection
macro smart_multiply dest, value {
```

```
    if value = 1
        ; Multiplication by 1 is identity
        if dest eq value
            ; Already correct
        else
            mov dest, value
        end if
    else if value = 2
        ; Use shift instead
        shl dest, 1
    else if value = 4
        shl dest, 2
    else if value = 8
        shl dest, 3
    else if value and (value - 1) = 0
        ; Power of 2, use shift
        local shift_amount
        shift_amount = 0
        repeat 64
            if (1 shl shift_amount) = value
                shl dest, shift_amount
                break
            end if
            shift_amount = shift_amount + 1
        end repeat
    else
        ; Use regular multiplication
        imul dest, value
    end if
}
```

ELF64 Linux Integration

Complete Program Template

asm

```
; program_template.asm - Generated by your compiler
format ELF64 executable 3
entry start

; Runtime library inclusion
include 'runtime.asm'
include 'linux_syscalls.asm'

; Data section for string literals and globals
section '.data' writeable
; Generated global variables go here

section '.rodata' readable
; Generated string literals go here

section '.bss' writeable
; Generated uninitialized data go here

section '.text' executable

; System call wrappers
macro sys_write fd, buffer, count {
    mov rax, 1      ; sys_write
    mov rdi, fd     ; file descriptor
    mov rsi, buffer ; buffer
    mov rdx, count  ; count
    syscall
}

macro sys_exit status {
    mov rax, 60     ; sys_exit
    mov rdi, status ; exit status
    syscall
}

; I/O functions for your language
print_int:
    ; Convert integer to string and print
    ; Implementation would go here...
    ret

print_string:
    ; Print null-terminated string
    ; rdi = string address
    push rdi
```

```
    push rdi

    ; Find string length
    xor rcx, rcx
.count_loop:
    cmp byte [rdi + rcx], 0
    je .count_done
    inc rcx
    jmp .count_loop
.count_done:

    pop rsi      ; string address
    mov rdx, rcx ; length
    sys_write 1, rsi, rdx
    ret

; Generated functions go here
; (Your C code will insert FASM function definitions here)

; Main program entry
start:
    ; Initialize runtime if needed

    ; Call main function
    call main

    ; Exit with return value
    sys_exit rax
```

Linux System Call Integration

asm

```
; linux_syscalls.asm - System call definitions
; System call numbers for x86_64 Linux
SYS_READ    = 0
SYS_WRITE   = 1
SYS_OPEN    = 2
SYS_CLOSE   = 3
SYS_MMAP    = 9
SYS_MUNMAP  = 11
SYS_BRK     = 12
SYS_EXIT    = 60

; File operations
macro open filename, flags, mode {
   mov rax, SYS_OPEN
   mov rdi, filename
   mov rsi, flags
   mov rdx, mode
   syscall
}

macro read fd, buffer, count {
   mov rax, SYS_READ
   mov rdi, fd
   mov rsi, buffer
   mov rdx, count
   syscall
}

macro write fd, buffer, count {
   mov rax, SYS_WRITE
   mov rdi, fd
   mov rsi, buffer
   mov rdx, count
   syscall
}

macro close fd {
   mov rax, SYS_CLOSE
   mov rdi, fd
   syscall
}

; Memory management
macro brk addr {
   mov rax, SYS_BRK
```

```
    mov rax, SYS_BRK
    mov rdi, addr
    syscall
}

; Simple memory allocator
heap_start dq 0
heap_current dq 0

malloc:
    ; rdi = size to allocate
    ; returns address in rax

    cmp qword [heap_start], 0
    jne .heap_initialized

    ; Initialize heap
    brk 0               ; Get current break
    mov [heap_start], rax
    mov [heap_current], rax

.heap_initialized:
    mov rax, [heap_current]
    add qword [heap_current], rdi
    ret

; Your language's runtime functions
panic:
    ; Print error message and exit
    mov rsi, panic_msg
    mov rdx, panic_msg_len
    write 2, rsi, rdx   ; Write to stderr
    sys_exit 1

section '.rodata'
panic_msg db 'Runtime error!', 10, 0
panic_msg_len = $ - panic_msg
```

Performance Optimizations

Assembly-Time Optimizations

asm

```
; Profile-guided optimization hints
HOT_FUNCTION_THRESHOLD = 1000
hot_functions_count = 0

macro hot_function name {
    hot_functions_count = hot_functions_count + 1
    display 'Hot function: ', `name

    ; Mark for aggressive optimization
    name#_is_hot = 1

    ; Use optimized function template
    func_begin name
        ; Add performance hints
        ; Align function to cache line boundary
        align 64

    macro func_end \\{
        ; Add optimized epilogue for hot functions
        mov rsp, rbp
        pop rbp
        ret
    \\}
}

; Branch prediction hints
macro likely_branch condition, target {
    ; Use static prediction - forward branches unlikely
    ; backward branches likely
    condition
    jmp target  ; Arrange code so likely path falls through
}

; Data prefetching
macro prefetch_data addr {
    ; Only generate prefetch for optimized builds
    if OPTIMIZATION_LEVEL >= 2
        ; prefetchnta [addr]  ; Non-temporal prefetch
    end if
}

; Loop optimization
macro optimize_small_loop counter_reg, limit, body {
    ; Unroll loops with small, known iteration counts
    if limit <= 4
```

```
    if limit <= 4
        repeat limit
            body
        end repeat
    else if limit <= 16
        ; Partially unroll
        mov counter_reg, 0
        .loop_start:
        repeat 4
            body
            inc counter_reg
        end repeat
        cmp counter_reg, limit
        jl .loop_start
    else
        ; Standard loop
        mov counter_reg, 0
        .loop_start:
        body
        inc counter_reg
        cmp counter_reg, limit
        jl .loop_start
    end if
}
```

Real-World Implementation

Complete Example: Fibonacci Generator

```c
// main.c - Your compiler frontend
int main(int argc, char **argv) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s input.lang output\n", argv[0]);
        return 1;
    }

    return compile_with_fasm_backend(argv[1], argv[2]);
}
```

```
# input.lang - Your language source
func fibonacci(n: int) -> int {
    if n <= 1 {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}

func main() -> int {
    let result = fibonacci(10);
    print_int(result);
    return 0;
}
```

asm

```
; Generated fibonacci.asm (by your C frontend)
include 'runtime.asm'

format ELF64 executable 3
entry start

section '.text' executable

; Function: fibonacci
func_begin fibonacci, n_param
    ; Load parameter
    load_var rax, n_param

    ; if n <= 1
    cmp rax, 1
    jg .recursive_case

    ; return n
    load_var rax, n_param
    func_end

.recursive_case:
    ; fibonacci(n - 1)
    allocate_reg
    load_var allocated_reg, n_param
    dec allocated_reg
    dyn_call fibonacci, allocated_reg
    push rax  ; Save result
    release_reg allocated_reg

    ; fibonacci(n - 2)
    allocate_reg
    load_var allocated_reg, n_param
    sub allocated_reg, 2
    dyn_call fibonacci, allocated_reg
    release_reg allocated_reg

    ; Add results
    pop rbx
    add rax, rbx
func_end

; Function: main
func_begin main
    ; fibonacci(10)
```

```
    ; fibonacci(10)
    dyn_call fibonacci, 10

    ; print_int(result)
    mov rdi, rax
    call print_int

    ; return 0
    xor rax, rax
func_end

start:
    call main
    mov rdi, rax
    sys_exit rdi
```

Build Script

```bash
#!/bin/bash
# build.sh - Complete build pipeline

set -e

INPUT_FILE="$1"
OUTPUT_FILE="$2"

if [ -z "$INPUT_FILE" ] || [ -z "$OUTPUT_FILE" ]; then
    echo "Usage: $0 input.lang output"
    exit 1
fi

# Compile with your hybrid C/FASM compiler
echo "Compiling $INPUT_FILE..."
time ./your_compiler "$INPUT_FILE" "$OUTPUT_FILE"

# Verify the executable
if [ -f "$OUTPUT_FILE" ]; then
    echo "Successfully generated $OUTPUT_FILE"
    ls -la "$OUTPUT_FILE"

    # Test run
    echo "Running executable:"
    "./$OUTPUT_FILE"
    echo "Exit code: $?"
else
    echo "Compilation failed!"
    exit 1
fi
```

This hybrid approach gives you:

- **Maximum flexibility** with FASM's metaprogramming
- **Complex logic** handled in C
- **Direct ELF64 output** for maximum speed
- **Sophisticated optimizations** at assembly time
- **Native Linux integration** with system calls

The result is a blazingly fast debug compiler that can generate optimized assembly code using FASM's powerful macro system while maintaining the development productivity of C for the complex compiler logic.

Perfect for that **1 million+ lines per second** compilation speed you're targeting! 🚀