Technical Report

Design and Implementation of a Custom 16-bit CPU Architecture with a Multi-stage Compilation Pipeline

Lukas Penner

lukas.penner.2001@gmail.com

Date: November 5, 2025

Contents

Abstract

This technical report presents the design and implementation of a fully custom 16-bit CPU architecture, its software toolchain, and its multi-level intermediate representation (IR) compilation system. The CPU includes RAM, cache, a bus communication protocol, and a pipelined execution model based on discrete states such as fetch, unpack, and execute. The toolchain supports an assembler, disassembler, peephole optimizer, macro expander, and an intermediate representation compiler (IR and IRA stages). The work represents the foundation for a full high-level transpiler pipeline capable of translating high-level code (eventually C) down to machine code.

Chapter 1

Introduction

The goal of this project is to design a custom 16-bit CPU from the ground up, including both the hardware simulation layer and the full software toolchain required to assemble, optimize, and execute code on it.

The CPU architecture follows a pipelined state machine design, communicating with memory components (RAM and cache) through a shared bus. The accompanying toolchain includes a multi-stage compiler that progressively reduces higher-level constructs down to assembly and machine code.

The main execution environment (implemented in main.c) integrates all components - compiler stages, bus, RAM, cache, ticker, and CPU - into a complete simulation loop, providing a realistic end-to-end verification platform for both compilation and execution.

Chapter 2

Project Architecture

2.1 Overview

The system consists of the following major components:

- CPU core: executes instructions, manages registers and ALU.
- Memory subsystem: RAM and cache, accessed via a shared bus.
- Assembler: translates assembly language into binary machine code.
- IR Compiler: converts intermediate representation (IR) code into assembly.
- IRA Compiler: higher-level compiler transforming IRA to IR.
- Optimizer and Macro Expander: improves code efficiency and supports macros.

2.2 File Structure

The source code is organized as follows:

```
 include/
 |-- asm/          # Assembler, disassembler, macros, optimizer
 |-- bus.h         # Bus communication interface
 |-- cpu/          # CPU core logic and instruction set
 |-- codegen/      # IR compiler, parser, lexer, and rulesets
 |-- transpile/    # IRA compiler and related parsers
 |-- utils/        # Utility libraries (IO, logging, string ops)
 src/
 |-- cpu/          # Implementation of CPU state machine
 |-- asm/          # Assembler implementation
 |-- codegen/      # IR compilation logic
 |-- transpile/    # IRA stage implementation
 |-- utils/        # Utility source files
```

Chapter 3

CPU

3.1 Internal Architecture

The CPU consists of four general purpose registers r0, r1, r2 and r3. They are all equal in their usage.

3.2 Addressing Modes

3.2.1 Addressing Mode Category (ADMC)

Besides None, there are three major categories of addressing modes:

- Immediate: The operand value is embedded directly in the instruction (e.g., mov r0, $1234).
- Register: The operand refers to the contents of a register (e.g., mov ↪ r0, r1).
- Indirect: The operand refers to a memory location, optionally with register-base offsets or scaling (e.g., mov r0, [r1 + $10]).

Table 3.1: Addressing Mode Categories (ADMC)

Category	Example	Description
ADMC_NONE	-	No operand
ADMC_IMM	mov r0, $FFFF	Value embedded in instruction
ADMC_REG	mov r0, r1	Operand is a register
ADMC_IND	mov r0, [r1 + $10]	Operand points to RAM via register or immediate offset

3.2.2 Extended Addressing Mode (ADMX)

ADMX is primarily used as a *source address*. If the instruction is an operation with only a single operand that calls a writeback within the CPU Instruction-State-Cycle, then the argument may only be a reduced addressing mode (ADMR). One such example is POP, which takes only one argument, but since it writes back to the specified address, it may only use ADMR. In contrast, PUSH, which also takes a single operand but only

writes back to a fixed location (the stack), can use ADMX as its source addressing mode.

Table 3.2: Extended Addressing Modes (ADMX)

Internal Name	Category (ADMC)	Reads From	Example
ADMX_NONE	ADMC_NONE	none	clz
ADMX_IMM16	ADMC_IMM	instruction	push $FFFF
ADMX_Rn	ADMC_REG	register	push r0
ADMX_SP	ADMC_REG	stack pointer	push sp
ADMX_PC	ADMC_REG	program counter	push pc
ADMX_IND16	ADMC_IND	RAM	push [$FFFF]
ADMX_IND_Rn	ADMC_IND	RAM	push [r0]
ADMX_IND_SP/PC	ADMC_IND	RAM	push [sp]
ADMX_IND_Rn_OFFSET16	ADMC_IND	RAM	push [r0 + $FFFF]
ADMX_IND_SP_OFFSET16	ADMC_IND	RAM	push [sp + $FFFF]
ADMX_IND_PC_OFFSET16	ADMC_IND	RAM	push [pc + $FFFF]
ADMX_IND16_SCALED8_Rn_OFFSET	ADMC_IND	RAM	push [$FF * r0 + $FFFF]
ADMX_IND16_SCALED8_SP_OFFSET	ADMC_IND	RAM	push [$FF * sp + $FFFF]
ADMX_IND16_SCALED8_PC_OFFSET	ADMC_IND	RAM	push [$FF * pc + $FFFF]

Note: Rn refers to general purpose registers r0, r1, r2, and r3.

3.2.3 Reduced Addressing Mode (ADMR)

ADMR is primarily used as a *destination address*. If an instruction has only one operand and performs a writeback within the CPU Instruction-State-Cycle, the operand must be an ADMR mode. For instance, POP writes data back to memory and therefore uses ADMR. PUSH uses ADMX since it only reads from its operand and writes to the fixed stack address.

Table 3.3: Reduced Addressing Modes (ADMR)

Internal Name	Category (ADMC)	Writes To	Example
ADMR_NONE	ADMC_NONE	none	clz
ADMR_Rn	ADMC_REG	register	pop r0
ADMR_SP	ADMC_REG	stack pointer	pop sp
ADMR_IND16	ADMC_IND	RAM	pop [$FFFF]
ADMR_IND_R0	ADMC_IND	RAM	pop [r0]

Note: Rn again refers to r0-r3.

3.3 Instruction Set

Table 3.4: CPU Instruction Set Summary

Mnemonic	Syntax	Description
NOP	nop	---
MOV	mov dest, src	src → dest
PUSH	push src	sp -= 2; src → [sp]
POP	pop dest	[sp] → dest; sp += 2
PUSHSR	push sr	sp -= 2; sr → [sp]
POPSR	pop sr	[sp] → sr; sp += 2
LEA	lea dest, [src]	address of src → dest
JMP	jmp dest	dest → pc
JZ	jz dest	if Z=1, dest → pc
JNZ	jnz dest	if Z=0, dest → pc
JL	jl dest	if N=1, dest → pc
JNL	jnl dest	if N=0, dest → pc
JUL	jul dest	if UL=1, dest → pc
JNUL	jnul dest	if UL=0, dest → pc
JFL	jfl dest	if FL=1, dest → pc
JNFL	jnfl dest	if FL=0, dest → pc
JSO	jso dest	if SO=1, dest → pc
JNSO	jnso dest	if SO=0, dest → pc
JAO	jao dest	if AO=1, dest → pc
JNAO	jnao dest	if AO=0, dest → pc
CALL	call dest	sp -= 2; pc → [sp]; dest → pc
RET	ret	[sp] → pc; sp += 2
ADD	add dest, src	dest = dest + src (int)
SUB	sub dest, src	dest = dest - src (int)
MUL	mul dest, src	dest = dest * src (int)
DIV	div dest, src	dest = dest / src (int)
NEG	neg dest	dest = dest ∼ 0x8000
ABS	abs dest	dest = dest & 0x7fff
INC	inc dest	dest = dest + 1
DEC	dec dest	dest = dest - 1
ADDF	addf dest, src	dest = dest + src (float16)
SUBF	subf dest, src	dest = dest - src (float16)
MULF	mulf dest, src	dest = dest * src (float16)
DIVF	divf dest, src	dest = dest / src (float16)
CIF	itf dest	convert integer → float16
CFI	fti dest	convert float16 → integer
CBW	cbw dest	sign extend byte to word
BWS	bws dest, src	bitwise shift dest by src
AND	and dest, src	dest = dest & src
OR	or dest, src	dest = dest | src

Mnemonic	Syntax	Description
XOR	xor dest, src	dest = dest $\wedge$ src
NOT	not dest	dest = $\neg$ dest
CMP	cmp a, b	set flags based on (a - b)
TST	tst val	set Z and N based on val
CLZ	clz	clear Z bit
SEZ	sez	set Z bit
CLL	cll	clear L bit
SEL	sel	set L bit
CLUL	clul	clear UL bit
SEUL	seul	set UL bit
CLFL	clfl	clear FL bit
SEFL	sefl	set FL bit
CLSO	clso	clear SO bit
SESO	seso	set SO bit
CLAO	clao	clear AO bit
SEAO	seao	set AO bit
CLSRC	clsrc	clear SRC bit (enable cache read)
SESRC	sesrc	set SRC bit (skip cache read)
CLSWC	cls wc	clear SWC bit (enable cache write)
SESWC	seswc	set SWC bit (skip cache write)
CLMI	clmi	clear mask-interrupt bit
SEMI	semi	set mask-interrupt bit
CMOVZ	cmovz dest, src	if Z=1 src $\rightarrow$ dest
CMOVNZ	cmovnz dest, src	if Z=0 src $\rightarrow$ dest
CMOVL	cmovl dest, src	if L=1 src $\rightarrow$ dest
CMOVNL	cmovnl dest, src	if L=0 src $\rightarrow$ dest
CMOVUL	cmovul dest, src	if UL=1 src $\rightarrow$ dest
CMOVNUL	cmovnul dest, src	if UL=0 src $\rightarrow$ dest
CMOVFL	cmovfl dest, src	if FL=1 src $\rightarrow$ dest
CMOVNFL	cmovnfl dest, src	if FL=0 src $\rightarrow$ dest
INV	inv	invalidate cache
FTC	ftc	fetch data into cache
INT	int dest	trigger interrupt (0xEF00 + dest $\rightarrow$ pc)
HLT	hlt	halt CPU

3.4 Device Usage and Interaction

The CPU operates as a device within the broader system and communicates with other components-such as RAM, cache, and clock-through the Device abstraction and shared bus. Each device implements a uniform interface defined by the Device_t structure, containing metadata (ID, type, state), communication data (address, data), and flags for transaction progress.

3.4.1 CPU as a Device

The CPU itself is represented by a Device_t instance, created via device_create(DT_CPU). When the CPU issues a memory request, it updates its internal device fields:

- address: the target memory address.
- data: payload (for stores) or expected read buffer (for fetches).
- device_state: set to DS_FETCH or DS_STORE.
- processed: cleared until the target device (RAM) replies.

The CPU then waits until processed = 1, signaling that the bus and target device have fulfilled the request.

3.4.2 Device States

All devices share a finite-state model:

- DS_IDLE: device is inactive.
- DS_FETCH: awaiting data fetch.
- DS_STORE: performing a memory store operation.
- DS_REPLY: ready to return data to requester.
- DS_INTERRUPT: raising an interrupt toward the CPU.

The CPU's execution pipeline checks device states continuously to synchronize with external hardware and handle asynchronous operations such as interrupts. When an interrupt occurs, the CPU transitions into a dedicated series of interrupt states (CS). These states replace the normal instruction pipeline until the interrupt vector has been resolved and control is transferred to the handler.

Interrupt-related CPU States:

- CS_INTERRUPT_PUSH_PC_HIGH: pushes the high byte of the current program counter onto the stack.
- CS_INTERRUPT_PUSH_PC_LOW: pushes the low byte of the current program counter onto the stack.
- CS_INTERRUPT_FETCH_IRQ_VECTOR_LOW: fetches the low byte of the interrupt vector address from the interrupt table at 0xEF00 + 2 × IRQ_ID.
- CS_INTERRUPT_FETCH_IRQ_VECTOR_HIGH: fetches the high byte of the interrupt vector address, forms the 16-bit destination, and jumps to it.

These states execute automatically when a peripheral raises DS_INTERRUPT and the mask interrupt flag (MI) is clear. The CPU disables nested interrupts during this process by setting MNI = 1, preventing re-entrance until the handler executes a CLMI instruction.

3.4.3 CPU–RAM Transactions

Memory reads and writes follow a strict handshake:

1. CPU sets DS_FETCH or DS_STORE.
2. Bus routes the request to the RAM device.
3. RAM performs the operation and sets processed = 1.
4. Bus propagates the response back to the CPU.
5. CPU clears the flag and continues execution.

This modular model allows new hardware (e.g., I/O devices or clocks) to be attached seamlessly.

3.4.4 Interrupt Handling Pipeline

Interrupts provide asynchronous control transfer from normal execution to dedicated service routines. When a device sets its state to DS_INTERRUPT, the CPU detects this condition at the beginning of each clock cycle and performs the following sequence:

1. If the MI (mask interrupt) bit in the status register is clear, the CPU begins interrupt entry.
2. The current program counter (pc) is saved onto the stack, high byte first.
3. The interrupt vector address is read from the interrupt table at 0xEF00 + (IRQ_ID × 2).
4. The CPU sets MNI = 1 (mask nested interrupt) to block re-entrant servicing.
5. Execution jumps to the handler located at the fetched address.

During the handler, the programmer typically saves and restores registers manually, or uses the dedicated pushsr and popsr instructions to preserve and restore the entire status register.

Typical Handler Entry and Exit:

```
    pushsr
    push r0
    push r1
    ; ... handler logic ...
    pop r1
    pop r0
    popsr
    clmi
    ret
```

This pattern mirrors the hardware sequence automatically performed during asynchronous interrupts, ensuring consistent and safe restoration of CPU state before returning to normal execution.

3.5 Caching System

The caching system acts as a lightweight intermediary between the CPU and main memory (RAM). Its purpose is to reduce memory access latency by storing recently used data in a smaller, faster memory buffer. All cache operations are handled via the Cache_t structure and associated functions defined in cache.h.

3.5.1 Internal Structure

A cache instance maintains four primary arrays, each indexed by the cache-line position:

- data: holds the cached byte values.
- address: stores the corresponding physical RAM addresses for each cached entry.
- state: maintains metadata for each entry such as validity, dirtiness, and usage statistics.
- capacity: the number of addressable cache lines, which must be a power of two.

Each entry's behavior is governed by a CacheState_t union that tracks:

- valid - whether the line contains valid data.
- dirty - whether RAM has newer data not yet reflected in cache.
- modified - reserved flag for future write-back logic.
- uses - frequency of recent access.
- age - line aging counter for potential eviction policies.

3.5.2 Address Mapping

Cache entries are selected using a simple modulo operation:

$$\text{cache\_address} = \text{address} \& (\text{capacity} - 1)$$

This effectively forms a direct-mapped cache, where each RAM address maps deterministically to one cache slot. No associative lookup or replacement logic is implemented - overwrites occur directly when collisions happen.

3.5.3 Read Logic

When the CPU issues a read:

1. The cache computes the local index from the physical address.
2. If the cache line is invalid, mismatched, or marked dirty, the read is considered a miss.
3. Otherwise, the requested byte is returned immediately from cache->data.
4. Cache hit and miss counters are updated accordingly.

This lookup is performed by cache_read(), which returns 1 on success or 0 on miss.

3.5.4 Write Logic

Cache writes are handled through cache_write(), which updates both metadata and the corresponding cache line:

1. The cache line index is determined using the same modulo mapping.
2. Each written byte updates the data, address, and state entries.
3. The line is marked as valid and clean (dirty = 0).
4. Usage counters (uses, age) are reset.

If a write bypass occurs (skipWrite = 1), the call is ignored. The return value 1 indicates a "dirty write" - data was written into cache before memory synchronization. However, in the current implementation, all writes return 0 since no delayed synchronization policy exists yet.

3.5.5 Performance Counters

The cache maintains two 64-bit counters:

- hit - number of successful cache lookups.
- miss - number of failed lookups that required RAM access.

These statistics can be used for profiling and performance tuning.

3.5.6 Design Notes and Limitations

- Current design is a direct-mapped cache with no associativity or replacement policy.
- dirty and modified flags are reserved for future write-back and coherence mechanisms.
- uses and age provide the basis for implementing LRU or decay-based replacement in future revisions.
- Memory synchronization currently relies on explicit CPU-RAM transactions managed through the bus.

Chapter 4

Peripherals

4.1 Overview

The system supports multiple peripheral devices that interact with the CPU through the shared bus interface described in Chapter 5. All peripherals implement the Device_t abstraction and participate in synchronous communication cycles managed by the bus scheduler. This modular design enables consistent interfacing between heterogeneous devices, such as memory, timers, and potential future I/O modules.

The project currently includes two major peripherals:

- RAM Device - the primary system memory used for code and data storage.
- Ticker Device - a periodic interrupt generator simulating a hardware timer or real-time clock source.

4.2 RAM Device

4.2.1 Structure

The Random Access Memory (RAM) peripheral is defined by the RAM_t structure:

```
typedef struct RAM_t {
    uint64_t clock;
    uint32_t capacity;
    uint8_t *data;
    Device_t device;
} RAM_t;
```

Each instance encapsulates its own byte-addressable data buffer and a Device_t interface that enables interaction with the CPU through the shared bus. The clock field is used for internal cycle counting and timing diagnostics.

4.2.2 Initialization and Memory Model

The RAM is created dynamically via:

```
RAM_t* ram_create(uint32_t capacity);
```

This allocates a contiguous memory block of the specified size (typically 64 KB in simulation). All bytes are initialized to zero upon creation.

Memory operations are organized around an 8-byte (64-bit) transfer width, corresponding to the size of the internal data field in the Device_t structure. When the CPU issues a memory fetch request, the bus signals the RAM device with the target address, and the RAM responds by assembling an entire 64-bit word from eight consecutive bytes in memory. This block is returned to the requesting device in a single logical transaction. Similarly, store operations commit one byte at a time, but still propagate through the bus as a synchronized transaction cycle. The design therefore models a hybrid between word-wide fetches and byte-granular writes, trading simplicity for speed in simulation.

4.2.3 Read/Write Operations

RAM read and write accessors are implemented as:

```
uint8_t ram_read(RAM_t* ram, uint16_t address);
void ram_write(RAM_t* ram, uint16_t address, uint8_t data);
```

Addressing is automatically wrapped modulo capacity, ensuring safe indexing regardless of overflow. In the simulation, no hardware-level segmentation or paging is enforced.

4.2.4 Clock Routine

The ram_clock() routine acts as the entry point for all bus-scheduled activity:

```
void ram_clock(RAM_t* ram);
```

During each clock tick, the RAM inspects its internal device_state field:

- DS_FETCH - performs a memory read and returns the data to the requester.
- DS_STORE - writes the provided data into memory.

After the operation completes, the device sets processed = 1 to signal the bus that the transaction is ready for propagation. In the case of a fetch, the 64-bit value is assembled from eight consecutive bytes starting at the requested address and stored into device.data. The bus then transfers this value back to the requesting device (typically the CPU), which reads it as a complete data unit. This implements a handshake-based memory transfer that is designed for asynchronous operation. In the current single-threaded simulator, the devices are clocked sequentially, but the architecture allows each device to run independently in its own thread. The handshake through device_state and processed supports true asynchronous execution, enabling realistic latency effects and making cache behavior meaningful under concurrent operation.

4.2.5 Design Notes

- All memory accesses are completed instantly in the current single-threaded simulator; there is no latency modeling beyond bus synchronization.
- The data buffer is directly accessible from the simulator, enabling memory inspection and disassembly.
- The device interface and handshake system (device_state, processed) are designed for asynchronous, multi-threaded execution, allowing realistic bus contention and cache effectiveness in future implementations.

- In future work, the RAM or a related memory-mapped controller may handle Memory-Mapped I/O (MMIO) operations. In such a model, addresses outside the standard RAM range could be redirected to peripheral devices, e.g. writes to a terminal output region or reads from a keyboard input buffer. This would result in compound transactions such as:

  $$\textit{write} : \texttt{CPU} \rightarrow \texttt{RAM} \rightarrow \texttt{Terminal} \quad \text{or} \quad \textit{read} : \texttt{CPU} \rightarrow \texttt{RAM} \rightarrow \texttt{Keyboard} \rightarrow \texttt{RAM} \rightarrow \texttt{CPU}.$$

  Such routing would centralize address-space management while maintaining consistent bus-level communication semantics.

- Although the memory array is byte-addressable, the device communicates in 64-bit blocks via the bus interface. This simplifies multi-byte data transfers for instruction fetch and stack operations.

4.3 Ticker Device

4.3.1 Purpose

The ticker acts as a periodic interrupt source, providing the CPU with a simulated hardware timer signal. This enables interrupt-driven timekeeping and periodic task execution within the CPU firmware or operating environment.

4.3.2 Structure and Timing Model

The ticker is defined as:

```
typedef struct Ticker_t {
    double time;
    double last_time;
    double intervall;
    Device_t device;
} Ticker_t;
```

It maintains an internal real-time accumulator measured in seconds, synchronized to the host's monotonic clock via clock_gettime().

4.3.3 Operation

Each call to ticker_clock() performs the following:

1. Samples the current host time and computes the elapsed delta.
2. Accumulates the delta into time.
3. If the elapsed time exceeds the configured interval, it triggers a device interrupt.

The interrupt is raised by setting:

```
ticker->device.device_state = DS_INTERRUPT;
ticker->device.address = INT_CLOCK;
```

This interrupt is then propagated by the bus to the CPU, which executes the corresponding interrupt vector routine (e.g., at 0xEF00 + INT_CLOCK).

4.3.4 Initialization and Configuration

The ticker is initialized via:

```
Ticker_t* ticker_create(float frequency);
```

The frequency parameter determines the interrupt interval:

$$\text{intervall} = \frac{1}{\text{frequency}}$$

For instance, a frequency of 1000.0 results in one interrupt every millisecond.

4.3.5 Integration with the Bus

In main.c, the ticker is instantiated alongside the CPU, RAM, and cache, and registered with the bus system:

```
Ticker_t* ticker = ticker_create(1000.0);
bus_add_device(bus, &ticker->device);
```

Although the main execution loop currently comments out the ticker invocation, it is designed to operate through:

```
ticker_clock(ticker);
```

Each tick potentially raises an interrupt, which the CPU must acknowledge and handle by executing the designated interrupt handler (defined using irqbegin/irqend blocks in the IR language).

4.3.6 Design Notes

- The ticker provides a real-time synchronized interrupt mechanism for event-driven CPU testing.
- The CPU itself is responsible for implementing timekeeping logic; the ticker only signals periodic interrupts.
- Multiple ticker instances can theoretically coexist to simulate independent timers or peripheral clocks.

4.4 Summary

Both the RAM and ticker devices demonstrate the extensibility of the system's bus-driven peripheral model. They share a common interface (Device_t) yet perform distinct roles:

- RAM - provides persistent memory storage for code and data.
- Ticker - provides temporal synchronization through interrupts.

Their modular implementation allows new peripherals (such as I/O devices or serial ports) to be added without modification to the CPU or bus logic. This design showcases the system's scalability and its fidelity to real hardware abstraction principles.

Chapter 5

Data Handling and Bus System

5.1 Overview

The communication between the CPU, RAM, cache, and peripheral devices is coordinated through a shared bus architecture. This design ensures modularity, where each device is isolated and interacts only via the standard interface defined in device.h.

5.2 Device Model

Each hardware unit-CPU, RAM, clock, etc.-is abstracted as a Device_t. A device encapsulates:

- A unique identifier (device_id)
- A device type (DEVICE_TYPE_t)
- A current operational state (DEVICE_STATE_t)
- Address and data fields used for bus transactions

This design allows flexible communication without hard-coded dependencies. For example, replacing the RAM device with a caching layer or secondary memory device requires no CPU-side modification.

5.3 Bus Routines and Scheduling

The BUS_t structure maintains a list of connected devices and cycles through them sequentially at every clock tick:

- DS_FETCH: routed from CPU to RAM; requests data.
- DS_STORE: routed from CPU to RAM; writes data.
- DS_REPLY: RAM replies to CPU and marks processed = 1.
- DS_INTERRUPT: raised by clock or peripheral; bus routes interrupt to CPU.

The round-robin bus routine ensures fairness and predictable timing, making the system suitable for deterministic simulation.

5.4 Design Considerations

- The bus operates synchronously with the system clock.
- Each device is only attended once per bus cycle.
- Device interaction is fully decoupled-no shared memory between devices.
- Interrupts use the same mechanism as memory requests, differing only in device_state.

This design promotes high cohesion within devices and loose coupling between them, simplifying simulation, debugging, and extension.

Chapter 6

Assembler

6.1 Syntax Overview

The assembler accepts a highly flexible and permissive syntax. Instructions, operands, and directives can be separated by any amount of whitespace or newlines, and comments can appear anywhere on a line after a semicolon. This section outlines the valid formats, conventions, and addressing modes recognized by the assembler.

6.1.1 Instruction Format

Instructions and their operands can be written with or without commas and may appear on the same or different lines. The following examples are all equivalent and valid:

```
mov r0, 123
mov r0 123
mov       r0       123
mov

r0

123
```

Instructions are not required to be separated by newlines, meaning an entire program can also be written as a single line:

```
mov r0 1 mov r1 2 hlt
```

6.1.2 Addressing Expressions

The assembler supports flexible addressing syntax. Multiple mathematically equivalent forms are accepted for register-based memory access. All of the following produce the same effective address:

```
mov r0, [$1234 + $12 * r0]
mov r0, [$1234 + r0 * $12]
mov r0, [$12 * r0 + $1234]
mov r0, [r0 * $12 + $1234]
```

Internally, the assembler recognizes these as scaled or offset indirect addressing modes. This flexibility simplifies decompilation and manual patching.

6.1.3 Immediate Values

Immediate operands can be written in several numeric formats. The assembler automatically determines the correct interpretation:

Format	Example	Meaning
Decimal	mov r0, 123	123 (0x007B)
Hexadecimal	mov r0, $123	0x0123 (291)
Hexadecimal	mov r0, 0x123	0x0123 (291)
Octal	mov r0, 0o123	0o123 (0x0053)
Binary	mov r0, 0b1011	0b1011 (0x000B)
Signed Decimal	mov r0, -123	-123 (0xFF85)
Float16	mov r0, f123	123.0 → 0x57B0
Float16	mov r0, f.123	0.1229854 → 0x2FDF

All formats are internally converted to 16-bit immediates (or float16 values) before encoding.

6.1.4 Comments

Comments begin with a semicolon (;) and continue to the end of the current line. They are ignored during assembly:

```
    mov r0, r1 ; this part is ignored
```

6.1.5 Labels

Labels mark addresses in the assembled program. They are defined using a leading dot and can be referenced like normal immediates or memory addresses.

```
    .L
    mov r0, .L
    mov r0, [.L + r1]
    jmp .L
```

During assembly, labels are resolved into 16-bit absolute addresses. The assembler automatically calculates their location, so they can be referenced before or after definition.

The only reserved label prefixes are the assembler directives: .code, .data, and .address.

If a label must appear within a data region, it can be temporarily encapsulated as follows:

```
    .data
    .value $1234
    .code
    .Label
    .data
```

6.1.6 Segments and Address Control

The assembler recognizes three intrinsic segment directives: .code, .data, and .
↪ address.

6.1.6.1 Code Segment (.code)

This is the default mode at startup. All lines are treated as executable instructions and translated into machine code.

6.1.6.2 Data Segment (.data)

Switching to the data segment causes the assembler to treat subsequent values as raw 16-bit words (or byte-aligned text if a string directive is used). For example:

```
    .data
    $1234
    $5678
```

The assembler writes each immediate as one 16-bit value in memory.
Strings can also be written directly using the .text directive:

```
    .text "Hello"
```

This emits each character as an 8-bit value packed into consecutive 16-bit words, null-terminated automatically.

6.1.6.3 Address Override (.address)

You can manually set the output address in RAM where the next instruction or data will be written:

```
    .address $1234
```

Subsequent bytes are written starting at address 0x1234. The assembler checks for overlapping memory regions and reports an error if two segments overwrite the same address range.

6.1.6.4 Switching Between Segments

Segments can be changed freely throughout a file. When switching back from data to code, the assembler continues translating at the current address unless explicitly moved using .address.

6.1.7 Example Program

The following example demonstrates all assembler directives in a practical context.

```
    ; Example assembly program

    call .main
    hlt

.address 0x0100
.L
.data
    $1234
    $5678
    $9abc
.hello
.text
```

```
    "Hello, world!"

.address 0x0200
.code
.main
    lea r1, [.hello]     ; Load address of data block
    add r1, 2            ; Offset by 2 bytes
    mov r0, [r1]         ; Read 16-bit value from memory
    ret
```

This example shows a complete program with code and data segments:

- Execution begins at address 0x0000, calling the subroutine .main.
- The label .L marks a block of raw data located at 0x0100.
- The segment at 0x0200 defines the .main routine, which loads and reads from that data block.
- During assembly, all labels are automatically resolved to their 16-bit absolute addresses.

6.1.8 Interrupt Vector and Handler Definition

Interrupts are defined as normal labels and can be assigned to hardware interrupt vectors through direct memory writes. Each interrupt vector is a 16-bit entry within the interrupt table beginning at 0xEF00. Writing the handler's label address into one of these entries associates the hardware source with its service routine.

For example, to register an interrupt routine for the INT_CLOCK device and increment a counter named .time:

```
.main
    mov [0xEF02], .irq_handler    ; assign IRQ handler to
       ↪ INT_CLOCK vector, which is at 0xEF00 + 0x0002
.L
    mov r0, [.time]               ; read time variable
    jmp .L                        ; idle loop
    hlt

.address 0x0100
.irq_handler
    semi                          ; disable further interrupts
    push r0
    pushsr                        ; preserve status register
    mov r0, [.time]               ; load current time
    add r0, 1                     ; increment
    mov [.time], r0               ; store updated time
    popsr                         ; restore status register
    pop r0
    clmi                          ; re-enable interrupts
    ret

.address 0x4000
.time                            ; global time counter
```

In this example:

- The vector entry at 0xEF02 corresponds to the INT_CLOCK hardware interrupt.
- The semi instruction masks nested interrupts during handler execution.
- The pushsr/popsr pair preserves the full processor status.
- The clmi instruction clears the interrupt mask to resume normal operation.
- .time resides at address 0x4000 and is updated every clock interrupt.

This approach clearly links the hardware interrupt table with user-defined handler routines while demonstrating proper interrupt entry and return conventions.

6.1.9 Special Operands and Decompilation Placeholders

The assembler also supports a special placeholder operand represented by an underscore (_). This placeholder allows incomplete or speculative instruction forms to remain syntactically valid.

```
    mov _, _
```

These instructions are technically valid during assembly but translate to "NONE" addressing modes, which usually produce invalid opcodes at runtime. Their main purpose is to preserve the structure of disassembled code segments that may originally have contained data rather than executable code.

6.1.9.1 Example: Reinterpreting Data as Code

If we modify the data section in the example above (see Listing 6.1.7) as follows:

```
.address 0x0100
.L
.data
    $b001
    $1234
    $5678
```

and then decompile the resulting binary using the DO_ADD_SPECULATIVE_CODE flag, the decompiler attempts to reinterpret possible instruction sequences embedded in data:

```
    call .jump_label0
    hlt
.address $0100
.L
.data
    $b001         ; mov _, [$1234 + $78 * r1]; 5-byte instruction
    $1234
    $5678
.address $0200
.code
.jump_label0
    lea r1, [$0100]
    add r1, $0002
```

```
    mov r0, [r1]
    ret
```

The comment on $b001 indicates that the decompiler speculated a possible instruction form at that address. Given this information (the instruction length and structure), the code can be rewritten more explicitly as:

```
    call .jump_label0
    hlt

.address 0x0100
.L
    mov _, [$1234 + $78 * r1]
.data
    $0056

.address 0x0200
.code
.jump_label0
    lea r1, [$0100]
    add r1, $0002
    mov r0, [r1]
    ret
```

This produces exactly the same binary output as before. The only adjustment is that the original $3456 value had to be shifted down by one byte, since the inserted 5-byte instruction displaced the data alignment.

6.1.9.2 Purpose of Placeholder Operands

Placeholder operands make decompiled or hybrid code segments round-trip safe - meaning the assembler can recompile speculative disassemblies back into the original binary bit-for-bit, even when parts of the data region were temporarily misidentified as code. This greatly simplifies debugging and reverse-engineering workflows.

6.2 Summary

The assembler provides a permissive and robust syntax, tolerant to whitespace, newlines, and operand variations. It supports multiple numeric formats, inline comments, automatic label resolution, and precise manual address control through segment directives.

Combined with the placeholder operand system, it can function both as a traditional assembler and as a reversible binary editing tool - capable of treating binary regions as either code or structured data as needed.

Chapter 7

Optimizer

7.1 Overview

The optimizer represents the final stage of the assembler pipeline, operating directly on fully parsed instruction sequences after the intermediate representation (IR) and assembler stages have completed. Its primary goal is to remove redundant or neutral operations, simplify instruction chains, and perform light algebraic folding without altering program semantics.

The optimizer is implemented in src/asm/optimizer.c, and functions as a post-processing pass that takes the emitted assembly source as input, parses it again into an internal instruction list, applies local pattern-based transformations, and then re-emits the optimized code as a text buffer.

Unlike most compiler optimizers, this implementation is deliberately conservative: all transformations are local, reversible, and applied only when a strict equivalence is provable at the instruction level. This guarantees bitwise identical execution semantics while reducing instruction count and code size.

7.2 Architecture

The optimizer builds directly upon the assembler's parsing subsystem. It reuses the same data structures for tokens, expressions, and instructions as the assembler itself. The overall workflow is as follows:

1. Split the input assembly into individual lines.
2. Strip comments and whitespace.
3. Tokenize each line into discrete words.
4. Convert the token stream into expression objects (Expression_t).
5. Parse expressions into Instruction_t structures representing low-level operations.
6. Iterate over the instruction array and apply local simplification rules.
7. Reconstruct the optimized assembly text from the modified instruction list.

All optimization passes are purely syntactic and operate directly on these Instruction_t structures. Each transformation is performed inside a single loop over the instruction array and may remove, modify, or reorder elements as needed.

7.3 Core Transformations

The optimizer performs multiple classes of optimizations, each targeting a specific redundancy pattern commonly emitted by the IR compiler or assembler frontend.

7.3.1 1. Self-assignments and Identity Operations

Neutral or self-referential instructions are removed outright:

- `mov rN, rN` → (removed)
- `add rN, 0`, `sub rN, 0`, `mulf rN, f1`, `divf rN, f1` → (removed)
- `mul rN, 1`, `div rN, 1` → (removed)

These are guaranteed to have no runtime effect other than unnecessary flag modification, which is avoided.

7.3.2 2. Redundant Move Elimination

Overwritten assignments to the same register are discarded:

```
    mov r0, x
    mov r0, y
    =>
    mov r0, y
```

7.3.3 3. Additive Identity Folding

Consecutive operations that perform addition with zero or redundant register transfer are collapsed:

```
    mov r0, 0
    add r0, r1
    =>
    mov r0, r1
```

7.3.4 4. Two-step Dereference Simplification

The sequence:

```
    lea r0, [var]
    mov r0, [r0]
    =>
    mov r0, [var]
```

7.3.5 5. Temporary Forwarding Elimination

Redundant forwarding through a temporary register is collapsed:

```
    mov rN, x
    add rM, rN
    mov rN, y
```

```
    =>
    add rM, x
    mov rN, y
```

7.3.6 6. Arithmetic Address Folding

Arithmetic operations performed immediately after a move into a register are rewritten as effective address computations:

```
    mov rN, rM
    add rN, imm16
    =>
    lea rN, [rM + imm16]
```

7.3.7 7. Constant Folding

Back-to-back immediate arithmetic instructions are precomputed at compile time:

```
    mov r0, 4
    add r0, 2
    =>
    mov r0, 6
```

7.3.8 8. Stack Operation Simplification

Consecutive stack pushes and pops that cancel each other are removed:

```
    push r0
    pop r0
    =>
    (removed)
```

Likewise, stack pointer adjustments are merged when consecutive:

```
    sub sp, 4
    sub sp, 2
    =>
    sub sp, 6
```

7.3.9 9. Neutral Logical Operations

Redundant bitwise operations are identified and discarded:

```
    and rN, 0xffff
    =>
    (removed)

    or rN, 0
    =>
    (removed)
```

These transformations are applied only if no status-register-dependent instruction appears before the next flag-modifying operation.

7.3.10 10. Zero Move Replacement

Assignments of zero are replaced by self-exclusive OR instructions:

```
mov r0, 0
=>
xor r0, r0
```

This substitution reduces instruction encoding size and runtime overhead, provided no status flag reads occur immediately afterward.

7.4 Implementation Strategy

The optimizer uses a simple iterative fixed-point approach. A top-level loop repeats all optimization passes until no further changes occur in the instruction list. Each iteration scans linearly through the list, applying the first matching rule and restarting when a modification is detected.

```
while (changes_applied) {
    changes_applied = 0;
    for (int i = 0; i < instruction_count; i++) {
        apply_rules(&instruction[i]);
        if (change_made) break;
    }
}
```

This ensures that dependent simplifications are applied sequentially - e.g., a mov-add pattern simplified to a lea may then be reduced further in a subsequent pass.

7.5 Addressing Mode Utilities

Several helper functions are used to verify operand equivalence and register category membership:

- is_same_adm() - checks equality of reduced and extended addressing modes.
- is_same_indirect_adm() - verifies indirect addressing equivalence.
- is_register_admx() - determines whether an operand is a register.
- is_arithmetic_operation() - classifies arithmetic and bitwise instructions.

These ensure that optimizations are applied only when the operand addressing semantics match exactly.

7.6 Output Reconstruction

After optimization, the instruction list is re-serialized into plain assembly text. The output builder traverses each Instruction_t, reconstructing mnemonic and operand strings while preserving labels, address markers, and code/data segment directives.

The reconstructed output is returned as a dynamically allocated string and can be reassembled by the standard assembler pipeline without modification.

7.7 Safety and Determinism

Each optimization is strictly side-effect free and operates only within a local instruction window. No register allocation, global flow analysis, or speculative reordering is performed. This conservative approach ensures:

- Deterministic code generation across runs.
- Compatibility with debug symbol mappings.
- Guaranteed semantic equivalence with unoptimized code.

7.8 Summary

The optimizer implements a lightweight, single-pass local optimization system capable of:

- Removing redundant operations.
- Folding constant expressions.
- Merging stack and arithmetic sequences.
- Simplifying addressing patterns.
- Reducing instruction footprint.

Its design prioritizes reliability, transparency, and assembly-level reproducibility over aggressive optimization.

Chapter 8

Disassembler

8.1 Overview

The disassembler represents the reverse counterpart to the assembler, reconstructing human-readable assembly source code from raw binary machine code. It is implemented in src/asm/disassembler.c and forms the final analytical stage of the toolchain, enabling binary verification, speculative code recovery, and visual inspection of compiled programs.

The disassembler supports both strict decoding and speculative reconstruction modes. In strict mode, only verified instruction sequences are emitted; in speculative mode, data regions that could represent valid instruction forms are reinterpreted and commented accordingly. Combined with the assembler's reversible syntax, this allows lossless round-trip disassembly and reassembly of binaries, even when code and data are interleaved.

8.2 Architecture

The disassembler operates in several distinct passes:

1. Byte stream decoding: The binary stream is sequentially read, and each instruction is tentatively parsed according to the CPU's instruction and addressing mode tables.
2. Instruction validation: The decoder checks opcode bounds, argument counts, and addressing mode combinations. Invalid or implausible sequences are marked as raw data.
3. Segment reconstruction: When invalid bytes are encountered, the disassembler emits .data or .address directives, resuming code translation when valid instruction patterns reappear.
4. Label inference: Optional passes introduce human-readable labels for jump targets, indirect memory references, and speculative source or destination addresses.
5. Assembly reconstruction: Each decoded instruction or data element is reassembled into formatted text, aligned and annotated according to selected disassembly options.

The resulting output is a syntactically valid assembly listing that can be passed directly back to the assembler, guaranteeing binary equivalence.

8.3 Core Functions

The disassembler's operation is divided into three main functions:

8.3.1 disassembler_decompile_single_instruction()

This function reads a single instruction and its operands from the binary stream. It determines addressing modes, calculates immediate or indirect argument sizes, and reconstructs textual mnemonics such as:

```
    add r0, [r1 + $0020]
```

If an invalid or unknown instruction is detected, the function marks it as data and returns an empty string. It also supports interpretation of float16 immediates when DO_USE_FLOAT_LITERALS is enabled.

8.3.2 disassembler_naive_decompile()

This is the first-pass linear disassembler. It performs a raw scan over the entire binary, maintaining a synchronized mapping between:

- The current binary index (program counter equivalent).
- The decoded instruction or data directive.
- The addressing mode metadata (ADMR/ADMX pairs).

Invalid sequences automatically switch the output mode to .data and resume decoding at the next possible alignment boundary. Segment boundaries (.address directives) are emitted when gaps or overlaps between code regions are detected.

8.3.3 disassembler_decompile()

This second-pass stage enhances the naive output by:

- Adding jump, source, and destination labels.
- Merging adjacent data regions.
- Annotating speculative code patterns.

It performs multiple refinement passes over the instruction list, reordering and inserting label lines as needed. The final step concatenates all assembly lines into a single dynamically allocated string, which may then be exported to disk.

8.4 Disassembly Options and Flags

The disassembler's behavior is fully controlled by a set of bitwise flags that can be combined via the DisassembleOption_t argument.

8.4.1 DO_ADD_JUMP_LABEL

When enabled, all absolute jump destinations are automatically labeled with sequential identifiers:

```
    .jump_label0
    jmp .jump_label0
```

This improves readability by replacing raw numeric addresses with symbolic references. The disassembler also inserts corresponding label definitions before each jump target.

8.4.2 DO_ADD_DEST_LABEL

For memory write operations (e.g., mov [address], src), this flag generates .dest_labelN tags at destination addresses that lie within known data or code regions. These labels assist in understanding self-modifying code or register-based addressing that targets fixed memory cells.

8.4.3 DO_ADD_SOURCE_LABEL

This option mirrors DO_ADD_DEST_LABEL but applies to read-side memory operands (e.g., mov dest, [address]). Labels are inserted for absolute or indirect sources, allowing the user to trace back read dependencies and data origins.

8.4.4 DO_ADD_LABEL_TO_CODE_SEGMENT

By default, labels are only added within data segments to avoid visual clutter. Enabling this flag allows the disassembler to insert labels directly into active code segments. This is primarily used when reconstructing binaries that intermingle code and constant pools within the same region.

8.4.5 DO_ADD_SPECULATIVE_CODE

Activates speculative decoding for ambiguous data regions. When raw data bytes match a valid opcode pattern, the disassembler annotates them as possible instructions:

```
    $b001         ; mov _, [$1234 + $56 * r1]; 5 byte instruction
    $1234
    $3456
```

This feature preserves possible code structures discovered during reverse engineering, allowing the user to manually refine or verify them later. This speculative analysis mode is directly compatible with the assembler's placeholder operand system, enabling seamless round-trip recompilation.

8.4.6 DO_USE_FLOAT_LITERALS

Interprets 16-bit immediate values as float16 literals when their binary pattern corresponds to a normalized floating-point constant within a reasonable numeric range. Typical constants such as 1.0, 2.718, or 3.1415 are displayed in float form, improving readability for floating-point heavy routines:

```
    mov r0, f3.1416
```

8.4.7 DO_ALIGN_ADDRESS_JUMP

When set, the disassembler enforces 2-byte alignment on instruction boundaries, particularly around .address and speculative nop padding sequences. This is useful when analyzing partially corrupted or non-aligned binaries, ensuring instruction boundaries remain synchronized with the architecture's 16-bit fetch width.

8.5 Output Characteristics

The disassembler output adheres to the assembler's permissive syntax conventions. Each instruction or data line is emitted as a valid statement, optionally followed by comments describing speculative decodings, inferred labels, or alignment information. To illustrate the progressive enrichment of the output, the following compact test program was used as input:

```
    .address $0000
    .code
    call .func1
    mov [.data16], r0
    mov r1, [.data16]
    lea r2, [.data_float]
    add r2, $0002
    mov r0, [r2]
    hlt
    .func1
    mov [.other], [.other]
    call $1234
    ret

    .address $0100
    .data
    .data16
    $9801
    $1234
    .data_float
    $3C00
    $4000
    $4248

    .address $1234
    nop
    .other
    ret
```

This small example mixes direct, indirect, and immediate addressing, along with distinct code and data regions and embedded literals. It serves as a minimal demonstration of all major disassembly options.

8.5.1 Without Flags

When no options are enabled, the disassembler performs a strict linear pass through the binary, emitting only recognized instructions and switching to .data segments

whenever decoding fails.

```
    call $0018
    mov [$0100], r0
    mov r1, [$0100]
    lea r2, [$0104]
    add r2, $0002
    mov r0, [r2]
    hlt
    mov [$1236], [$1236]
    call $1234
    ret
    .address $0100
    .data
    $9801
    $1234
    .address $0105
    .code
    clmi
    .data
    $4840
    $0042
    .address $1236
    .code
    ret
```

Here, no symbolic names or speculative annotations appear. Segments are recovered heuristically: the disassembler switches between .code and .data whenever instruction validity changes.

8.5.2 With Standard Flags

With the typical set of options enabled (DO_ADD_JUMP_LABEL, DO_ADD_DEST_LABEL, DO_ADD_SOURCE_
↪ , and DO_ADD_SPECULATIVE_CODE), the output gains inferred labels and comments for speculative instruction interpretations:

```
    call .jump_label0
    mov [.dest_label0], r0
    mov r1, [.dest_label0]
    lea r2, [$0104]
    add r2, $0002
    mov r0, [r2]
    hlt
    .jump_label0
    mov [$1236], [$1236]
    call $1234
    ret
    .address $0100
    .data
    .dest_label0
    $9801            ; mov _, [$1234 + sp]    ; 4 byte instruction
    $1234
    .address $0105
```

```
    .code
    clmi
    .data
    $4840           ; cmovl _, [r0]          ; 2 byte instruction
    $0042           ; cmovul _, _            ; 2 byte instruction
    .address $1236
    .code
    ret
```

In this listing, jump and data references are replaced with symbolic labels, while embedded data patterns that could plausibly represent instructions are shown with their speculative decodings in the comment fields.

8.5.3 With All Flags Enabled

Finally, when all flags are active - DO_ADD_JUMP_LABEL, DO_ADD_DEST_LABEL, DO_ADD_SOURCE_LABE
↪ , DO_ADD_LABEL_TO_CODE_SEGMENT, DO_ADD_SPECULATIVE_CODE, DO_USE_FLOAT_LITERALS
↪ , and DO_ALIGN_ADDRESS_JUMP - the disassembler produces a semantically rich, fully annotated output:

```
    call .jump_label0
    mov [.dest_label0], r0
    mov r1, [.dest_label0]
    lea r2, [$0104]
    add r2, $0002
    mov r0, [r2]
    hlt
    .jump_label0
    mov [.dest_label1], [.source_label0]
    call $1234
    ret
    .address $0100
    .data
    .dest_label0
    $9801           ; mov _, [$1234 + sp]    ; 4 byte instruction
    $1234
    f1.000000
    f2.000000
    f3.140625
    .address $1236
    .code
    .source_label0
    .dest_label1
    ret
```

Compared to the previous stage:

- Source and destination labels (.source_labelN, .dest_labelN) are added directly in code segments.
- Float literals replace their raw hexadecimal immediates where valid 16-bit floating point patterns were detected.

- Aligned decoding ensures clean separation between float data and following instructions, preventing cross-interpretation of trailing zeros.
- Speculative code annotations remain visible in data regions, preserving potential alternative interpretations.

8.5.4 Shortcomings of speculative annotations

An example where the speculative code cannot be recompiled in a one-to-one correspondence is when the speculative opcode does not have sufficient operands to fully encode the 8 bits of addressing mode, as some instructions take only one or zero operands, which results in zeroing out the unused addressing bits during compilation. Given this disassembly:

```
    .data
    $d702       ; push [$000C + $00 * pc]   ; 5 byte instruction
    $000c       ; jul _                     ; 2 byte instruction
```

We get the following speculative disassembly (taking into account, that the jul _ ↪ part is overwritten by the 5-byte push instruction):

```
    .code
    push [$000C + $00 * pc]
```

This however is not in one-to-one correspondence to the original binary, as it can only recover ($d702 & $f8ff =) $d002, as the encoding of push uses admx exclusively, which is encoded as the last 5 bits of the second byte of the instruction encoding. So instead, this speculative code equates to this:

```
    .data
    $d002       ; here it's $d002 instead of $d702
    $000c
```

In this next example:

```
    .data
    $ff03       ; pop [r0]                  ; 2 byte instruction
    $000c       ; jul _                     ; 2 byte instruction
```

We get the following speculative disassembly:

```
    .code
    pop [r0]
    jul _
```

The caveat is that pop uses amdr exclusively, which is encoded within the first 3 bits of the second byte of the instruction encoding. Thus it can only recover ($ff03 ↪ & $03ff =) $0303, so the actual corresponding code looks like this:

```
    .data
    $f803       ; here it's $f803 instead of $ff03
    $000c
```

8.5.5 Feature Summary

Flag Set	New Features Activated
No Flags	Minimal decoding, raw addresses only
Standard Flags	Jump and data labels, speculative annotations
All Flags	Labels in code, float literals, alignment-aware decoding

8.5.6 Discussion

This example demonstrates how the disassembler's modular options refine the readability and structure of its output. Starting from a purely linear byte stream, successive passes recover control flow, label hierarchies, literal values, and safe alignment boundaries - reconstructing an assembly listing that is both human-readable and fully reassemblable. Even in mixed code/data binaries, the final output maintains consistency with the assembler's syntax and can be reassembled into a functionally identical binary.

8.6 Memory and Label Tracking

During decoding, the disassembler maintains several parallel arrays:

- code[] - textual lines for each decoded entity.
- binary_index[] - original binary offset of each instruction.
- admr[] and admx[] - reduced and extended addressing modes used.
- is_data[] and is_label[] - classification of each line.

This mapping enables deterministic correlation between binary offsets and source-level representations. When new labels are inserted, arrays are dynamically resized, and references are rewritten to ensure consistency across later passes.

8.7 Summary

The disassembler functions as both an analytical and reconstructive tool. Its layered decoding process, combined with flexible labeling and speculation mechanisms, allows binaries to be faithfully converted back to structured assembly - even when containing malformed or partially unknown regions.

Key characteristics:

- Accurate decoding of CPU instruction and addressing mode tables.
- Automatic segmentation between code and data.
- Label inference for jumps, reads, and writes.
- Optional speculative reconstruction of mixed regions.
- Float literal recognition for enhanced readability.
- Full round-trip safety with the assembler.

Together, these capabilities make the disassembler a powerful inspection utility for compiled programs, debugging sessions, and reverse-engineering workflows within the custom 16-bit CPU ecosystem.

Chapter 9

Compiler

9.1 Intermediate Representation (IR)

9.1.1 Syntax

9.1.1.1 Keywords

The language parses the following set of keywords:

Table 9.1: CPU Instruction Set Summary

Internal name	Matches	Use
IR_LEX_VAR	"var"	Declare a new variable
IR_LEX_REF	"ref"	Reference a memory location
IR_LEX_DEREF	"deref"	Dereference a memory address
IR_LEX_IF	"if"	Conditional branch start
IR_LEX_ELSE	"else"	Conditional branch alternative
IR_LEX_GOTO	"goto"	Unconditional jump to a label
IR_LEX_RETURN	"return"	Function return statement
IR_LEX_ARG	"arg"	Holds pointer to stack variables
IR_LEX_CONST	"const"	Define a constant value
IR_LEX_STATIC	"static"	Define a static variable
IR_LEX_ANON	"anon"	Anonymous temporary or unnamed scope
IR_LEX_CALLPUSHARG	"callpusharg"	Push an argument for function call
IR_LEX_CALLFREEARG	"callfreearg"	Free argument after function call
IR_LEX_CALL	"call"	Invoke a function or subroutine
IR_LEX_SCOPEBEGIN	"scopebegin"	Start of a local scope block
IR_LEX_SCOPEEND	"scopeend"	End of a local scope block
IR_LEX_CFI	"cfi"	Convert float to integer
IR_LEX_CIF	"cif"	Convert integer to word
IR_LEX_CBW	"cbw"	Convert byte to word (sign-extend)
IR_LEX_ADDRESS	".address"	Set manual output address in memory
IR_LEX_IRQBEGIN	"irqbegin"	Begin interrupt routine definition
IR_LEX_IRQEND	"irqend"	End interrupt routine definition
IR_LEX_ASM	"asm"	Inline assembly block marker
IR_LEX_INTEGER_MINUS	"i-"	Integer subtraction operation
IR_LEX_FLOAT_MINUS	"f-"	Floating-point subtraction
IR_LEX_INTEGER_PLUS	"i+"	Integer addition
IR_LEX_FLOAT_PLUS	"f+"	Floating-point addition
IR_LEX_INTEGER_SLASH	"i/"	Integer division
IR_LEX_FLOAT_SLASH	"f/"	Floating-point division
IR_LEX_INTEGER_STAR	"i*"	Integer multiplication

Internal name	Matches	Use
IR_LEX_FLOAT_STAR	"f*"	Floating-point multiplication
IR_LEX_BANG_EQUAL	"!="	Inequality comparison
IR_LEX_EQUAL_EQUAL	"=="	Equality comparison
IR_LEX_UNSIGNED_INTEGER_GREATER_EQUAL	"u>="	Unsigned greater or equal
IR_LEX_INTEGER_GREATER_EQUAL	"i>="	Signed integer greater or equal
IR_LEX_FLOAT_GREATER_EQUAL	"f>="	Float greater or equal
IR_LEX_UNSIGNED_INTEGER_LESS_EQUAL	"u<="	Unsigned less or equal
IR_LEX_INTEGER_LESS_EQUAL	"i<="	Signed integer less or equal
IR_LEX_FLOAT_LESS_EQUAL	"f<="	Float less or equal
IR_LEX_UNSIGNED_INTEGER_GREATER	"u>"	Unsigned greater than
IR_LEX_INTEGER_GREATER	"i>"	Signed integer greater than
IR_LEX_FLOAT_GREATER	"f>"	Float greater than
IR_LEX_UNSIGNED_INTEGER_LESS	"u<"	Unsigned less than
IR_LEX_INTEGER_LESS	"i<"	Signed integer less than
IR_LEX_FLOAT_LESS	"f<"	Float less than
IR_LEX_SHIFT_LEFT	"«"	Bitwise shift left
IR_LEX_SHIFT_RIGHT	"»"	Bitwise shift right
IR_LEX_LEFT_BRACKET	"["	Array or memory access begin
IR_LEX_RIGHT_BRACKET	"]"	Array or memory access end
IR_LEX_COMMA	","	Argument or element separator
IR_LEX_SEMICOLON	";"	Statement terminator
IR_LEX_ASSIGN	"="	Assignment operator
IR_LEX_BITWISE_AND	"&"	Bitwise AND
IR_LEX_BITWISE_OR	"|"	Bitwise OR
IR_LEX_BITWISE_NOT	"~"	Bitwise NOT
IR_LEX_BITWISE_XOR	"^"	Bitwise XOR
IR_LEX_BANG	"!"	Logical NOT
IR_LEX_PERCENT	"%"	Modulo (integer remainder)
IR_LEX_IDENTIFIER	name	Variable, function, or label name
IR_LEX_LABEL	".label"	Label definition within IR code
IR_LEX_STRING	"..."	String literal
IR_LEX_CHAR_LITERAL	"'c'"	Character literal
IR_LEX_NUMBER	"123", "0xFF", etc.	Numeric literal (int/float/hex)
IR_LEX_COMMENT	"//..." or "/*...*/"	Code comment, ignored by compiler
IR_LEX_EOF	-	End of file/input stream
IR_LEX_UNDEFINED	-	Unrecognized or invalid token
IR_LEX_RESERVED	-	Reserved keyword for future use

9.1.2 Register Mapping

No dynamic register allocation is currently implemented. Instead, the Intermediate Representation (IR) compiler assigns each general-purpose register a fixed and well-defined role. This deterministic mapping simplifies code generation and debugging but restricts the available register optimization potential.

- r0 - Used as the primary accumulator and general-purpose data register. It frequently holds intermediate computation results and is overwritten between consecutive operations.
- r1 - Serves as a secondary temporary register and as a pointer register for indirect memory access. It is often used in address calculations and dereference operations.
- r2 - Currently unused. It is reserved for potential future extensions, such as dedicated interrupt handling or system-level operations.
- r3 - Functions as the frame pointer for local stack variables. At the beginning of each function scope, r3 stores the value of the stack pointer (sp), allowing the compiler to access local variables via fixed negative offsets relative to r3. Upon function return, the original value of r3 is restored.

This static register convention ensures predictable behavior across all IR-compiled functions and allows direct correlation between IR instructions and the generated assembly output.

9.1.3 Semantic Structure

The Intermediate Representation (IR) forms the bridge between high-level language constructs and the final assembly emitted by the code generator. It expresses all control-flow, arithmetic, and memory access in an explicit and low-level form, while remaining readable and reversible. Every IR instruction corresponds to one or more assembly-level operations, and the compiler performs no hidden optimizations - all transformations are direct and deterministic.

The IR is designed as a line-oriented pseudo-language with clear textual separation between declarations, operations, and flow-control statements. Each instruction follows a rigid form:

<destination> = <operation> [operands]

Operands can include identifiers, constants, references, dereferences, or nested expressions. All operations are explicit - for instance, subtraction is expressed as i-, addition as i+, and so forth. The explicit type prefix (i for integer, f for float) allows the IR compiler to map the correct CPU instruction set variant.

9.1.4 Program Structure

An IR file can contain:

- Static variable declarations: persistent symbols shared between functions.
- Address markers: define exact memory placement for subsequent functions.
- Function and interrupt definitions: each begins with a label and uses scoped blocks.
- Lexical scopes: delimited by scopebegin and scopeend, these form local stack frames.
- Control flow: conditional jumps, unconditional jumps, and returns.

The IR system distinguishes between *static storage* (global symbols mapped to fixed addresses) and *automatic storage* (local stack-based variables). The compiler allocates stack space dynamically at the beginning of each scope, maintaining offsets relative to the frame pointer (r3).

9.1.5 Example: Fibonacci and Interrupt Handling

The following listing shows a representative IR program, demonstrating variable declarations, arithmetic operations, and recursive function calls:

Listing 9.1: Example IR program using static and local scopes

```
static var N;
static var xyz;
static var time;

.address 0x3000;

.main
scopebegin;

// set up interrupt address for INT_CLOCK vector
var address;
```

```
address = 0xef02;
var funcptr;
funcptr = .irq_handler;
deref address = funcptr;

N = 16;

// output variable
var out;
out = 0i-1;

var dout;
dout = ref out;

callpusharg dout;
callpusharg N;
call .fib;
callfreearg 4;

var abc;
abc = time;

scopeend;
return;

.address 0x1000;

.fib
scopebegin;

xyz = xyz i+ 1;

var ppo;
ppo = arg i+ 2;
var po;
po = deref ppo;

var pn;
pn = arg i+ 0;
var n;
n = deref pn;

var cond;
cond = n i> 1;

if cond .valid;
deref po = n;
return
.valid

var fib_m2;
```

```
    var dfib_m2;
    dfib_m2 = ref fib_m2;
    var agm2;
    agm2 = n i- 2;
    callpusharg dfib_m2;
    callpusharg agm2;
    call .fib;
    callfreearg 4;

    var fib_m1;
    var dfib_m1;
    dfib_m2 = ref fib_m1;
    var agm1;
    agm1 = n i- 1;
    callpusharg dfib_m2;
    callpusharg agm1;
    call .fib;
    callfreearg 4;

    var sum;
    sum = fib_m2 i+ fib_m1;
    deref po = sum;

    scopeend;
    return;

    .irq_handler
    irqbegin
    time = time i+ 1;
    irqend
    return;
```

9.1.6 Compiler Translation Process

Each IR file is processed by the IR compiler in several deterministic stages:

1. Lexical analysis The lexer scans the source stream and classifies tokens according to the keyword and operator tables listed in Section 3.3. All identifiers, literals, and control tokens are normalized into fixed categories (e.g., IR_LEX_VAR, IR_LEX_INTEGER_PLU
   ↪ , etc.). Comments and whitespace are stripped at this stage.
2. Parsing and rule evaluation The parser uses a rule-based grammar defined in ir_parser_ruleset.c It maps token sequences into typed statements such as variable declarations, assignments, or control-flow constructs. Each statement becomes an ir_node structure, forming a linear intermediate representation tree.
3. Semantic lowering High-level operations like callpusharg or deref are decomposed into multiple instruction templates that manipulate registers and stack memory. All stack-local symbols are assigned negative offsets relative to r3, while global symbols use fixed data addresses. Arithmetic operations are expanded into explicit register moves, loads, and ALU instructions.
4. Assembly emission Finally, each IR node emits one or more textual assembly lines. This emission is handled in ir_compiler.c using a direct-print approach - no intermediate

code buffering is done. The resulting output is fully compatible with the assembler described in Chapter 2.

9.1.7 Stack and Scope Management

Each function begins with a scopebegin instruction that creates a new frame. The compiler emits:

```
push r3
mov r3, sp
```

Every local variable allocation reduces the stack pointer by 2 bytes (16 bits per word). Variables are accessed relative to r3, e.g., [r3 + -4]. At the end of the scope, the compiler restores the frame:

```
mov sp, r3
pop r3
ret
```

Nested scopes are flattened at compile time - their lifetimes are strictly stack-ordered, and no variable reuse occurs within the same frame.

9.1.8 Control Flow and Conditional Evaluation

The IR compiler translates each conditional into an explicit comparison and flag sequence. For example, the following IR snippet:

```
cond = n i> 1;
if cond .label;
```

is compiled to:

```
mov r1, [r3 + -8]    ; n
mov r0, r1
mov r1, 1
cmp r0, r1
mov r1, 0
cmovnl r1, 1
cmovz r1, 0
tst r1
jnz .label
```

All relational operators (i>, i<, i>=, i<=, etc.) are expanded in this fashion. The condition variable is always a 16-bit integer treated as boolean (nonzero = true).

9.1.9 Function Calls and Argument Handling

Function parameters are passed by value or by reference through the stack. Each call sequence is expressed as:

```
callpusharg <expr>;
callpusharg <expr>;
call .func;
callfreearg <n>;
```

The compiler translates these into:

```
sub sp, 2
mov r1, <expr>
```

```
    lea r0, [sp]
    mov [r0], r1
    ...
    call .func
    add sp, <n>
```

This mechanism allows an arbitrary number of arguments and maintains stack alignment at 16 bits.

9.1.10 Dereferencing and References

ref creates an address reference to a variable or memory location, while deref loads or stores through that pointer.

Example:

```
    var ptr;
    ptr = ref x;
    var val;
    val = deref ptr;
```

This emits:

```
    lea r1, [r3 + -2]   ; x
    lea r0, [r3 + -4]   ; ptr
    mov [r0], r1
    mov r1, [r3 + -4]
    mov r0, [r1]
    lea r0, [r3 + -6]   ; val
    mov [r0], r0
```

This system provides full pointer semantics without requiring a separate address space or type metadata.

9.1.11 Interrupt Handlers and Vector Registration

Interrupt routines are defined using the irqbegin and irqend markers. Before an interrupt can be triggered, its handler's address must be written to the appropriate vector in hardware memory.

Example vector registration sequence:

```
    var address;
    address = 0xef02;           // INT_CLOCK vector
    var funcptr;
    funcptr = .irq_handler;
    deref address = funcptr;    // write handler address to
       ↪ vector
```

This code emits:

```
    mov r1, 61186          ; address = 0xEF02
    lea r0, [r3 + -2]
    mov [r0], r1
    lea r1, [.irq_handler]
    mov r0, [r3 + -4]
    mov [r0], r1
    mov r1, [r3 + -2]
    mov r0, [r3 + -4]
```

```
    mov [r1], r0
```

Once registered, the CPU automatically calls the handler upon interrupt signal. The handler definition remains identical:

```
    .irq_handler
    irqbegin
    time = time i+ 1;
    irqend
    return;
```

Which compiles to:

```
    push r3
    mov r3, sp
    push r2
    push r1
    push r0
    pushsr
    mov r1, [16388]    ; time
    mov r0, r1
    mov r1, 1
    add r0, r1
    mov r1, r0
    lea r0, [16388]
    mov [r0], r1
    popsr
    pop r0
    pop r1
    pop r2
    clmi
    mov sp, r3
    pop r3
    ret
```

Handlers can update static or global variables (e.g., incrementing a timer) and must always conclude with `clmi` and `ret`. The compiler enforces this termination automatically when using `irqend`.

```
    irqbegin
    time = time i+ 1;
    irqend
```

Compiles to:

```
    push r3
    mov r3, sp
    push r2
    push r1
    push r0
    pushsr
    mov r1, [16388]    ; time
    mov r0, r1
    mov r1, 1
    add r0, r1
    mov r1, r0
```

```
    lea r0, [16388]
    mov [r0], r1
    popsr
    pop r0
    pop r1
    pop r2
    clmi
    mov sp, r3
    pop r3
    ret
```

The clmi instruction resets the interrupt flag, ensuring proper re-arming for the next signal.

9.1.12 Type Modifiers

The IR language defines several type modifiers to control storage and visibility semantics during compilation. The following modifiers are currently implemented in the compiler:

- static -- Allocates the variable in the data segment, retaining its value across scopes and function calls. Static variables are assigned absolute addresses starting at 0x4000.
- const -- Marks the variable as immutable after initialization. The compiler emits an error if reassignment occurs.
- anon -- Declares an *anonymous* temporary variable without an explicit identifier. These variables exist only within the current scope, are allocated on the stack, and cannot be referenced by name. This is a placeholder for higher level compilers to do tasks like array allocation, where not every entry requires its own variable name.

The compiler determines these attributes through recursive analysis of the variable declaration token tree using the function vardec_get_type_modifier(), which maps the lexical tokens (IR_LEX_STATIC, IR_LEX_CONST, IR_LEX_ANON) to internal flag bits (IR_TM_STATIC, IR_TM_CONST, IR_TM_ANON).

9.1.12.1 Static Variable Allocation

All global symbols declared with static var are placed into the data segment, starting at the first declared static address. Each symbol receives a consecutive 16-bit address, beginning at 0x4000 unless overridden. Static accesses are emitted as direct absolute loads and stores using the lea or mov addressing mode.

Example:

```
        N = 16;
```

is emitted as:

```
        mov r1, 16
        lea r0, [16384]
        mov [r0], r1
```

9.1.13 Summary

The IR system defines a precise and reversible representation of low-level program logic. It enforces full transparency of data movement and scope handling, enabling a one-to-one mapping between IR instructions and generated assembly.

Its characteristics include:

- Deterministic code generation with no hidden optimizations.
- Explicit stack and frame management.
- Uniform handling of variables, references, and dereferences.
- Direct support for conditional branching and recursive calls.
- Integrated interrupt definition and register preservation.

Together, these features make the IR compiler a foundational component of the multi-stage toolchain, serving as the translation layer between higher-level constructs (from IRA or later C-transpiled stages) and the low-level 16-bit assembly code executed by the CPU simulator.

9.2 Higher level abstraction

The motivation now lies in building greater and greater abstraction levels, with the main goal being a seamless bridge between a subset of C as a programming language and the custom machine code for the architecture. The idea is to build a cascading transpilation pipeline that would remove more and more abstractions along the line, starting at C with full language support, to the next step, down to IR, then to asm and finally to machine code. In this section I want to show the different planned stages of deabstractions.

9.2.1 subset of C

The highest level of abstraction planned for this project is a subset of C, where most features are implemented. C style code like structs, typedefs, enums, unions would all be available to the programmer.

9.2.2 IRI

This is the next abstraction step, by passing the return values as pointers themselves this step also canonicalizes array indexing. For example, this code:

```
1   int factorial(int n) {
2       if (n <= 1) {
3           return 1;
4       }
5       return n * factorial(n - 1);
6   }
```

turns into this:

```
1   int factorial(int* out, int n) {
2       if (n <= 1) {
3           *out = 1;
4           return;
5       }
6       int out2;
7       factorial(&out2, n - 1);
8       *out = n * out2;
9       return;
10  }
```

9.2.3 IRH

This is the next abstraction step, by opening up structs and passing the arguments one by one.

```
    typedef struct {
        int x;
        float y;
    } Pair;

    void make_pair(Pair* out, int a, float b) {
        Pair p;
        p.x = a;
        p.y = b;
        *out = p;
        return;
    }

    void main() {
        Pair final;
        make_pair(&final, selected, y);
    }
```

turns into this:

```
    void make_pair(int* out_x, float* out_y, int a, float b) {
        int p_x;
        float p_y;
        p_x = a;
        p_y = b;
        *out_x = p_x;
        *out_y = p_y;
        return;
    }

    void main() {
        int final_x;
        float final_y;
        make_pair(&final_x, &final_y, selected, y);
    }
```

9.2.4 IRG

This is the next abstraction step, by removing explicit typing and handling them inside. Implicitly using them as int16_t. Also changed naming-, and casting conventions. This is the first step that is no longer C-compatible.

```
    void sum_if_positive(int* out, int a, float b) {
        if (a > 0 && b > 0.0f) {
            *out = a + (int)b;
            return;
        }
        *out = 0;
        return;
```

```
    }
```

turns into this:

```
    void sum_if_positive(var out, var a, var b) {
        if (a i> 0 && b f> f0.0) {
            *out = a + (fti)b;
            return;
        }
        *out = 0;
        return;
    }
```

Note: i> and f> means "integer-greater" and "float-greater" respectively, and fti means "float-to-integer" conversion.

9.2.5 IRF

Changing naming convention and pointer handling.

```
    void factorial(var out, var n) {
        if (n i<= 1) {
            *out = 1;
            return;
        }
        var out2;
        factorial(&out2, n - 1);
        *out = n * out2;
        return;
    }
```

turns into this:

```
    .factorial(var out, var n) {
        if (n i<= 1) {
            *out = 1;
            return;
        }
        var out2;
        var out2_ptr = ref out2;
        call .factorial(out2_ptr, n - 1);
        *out = n * out2;
        return;
    }
```

Note: functions are now denoted by a period before the function name and variable referencing is now explicitly written using ref

9.2.6 IRE

Changing naming convention and pointer handling

```
    .sum_if_positive(var out, var a, var b) {
        if (a i> 0 && b f> f0.0) {
            *out = a + (fti)b;
            return;
```

```
        }
        *out = 0;
        return;
    }
```

turns into this:

```
    .sum_if_positive(2, 4) {
        var out = *(arg + 0);
        var a = *(arg + 2);
        var b = *(arg + 4);
        if (a i> 0 && b f> f0.0) {
            *out = a + (fti)b;
            return;
        }
        *out = 0;
        return;
    }
```

Note: 2 bytes as outgoing data (which the caller must make space for in stack before calling, reachable through r2/caller-frame-pointer). 4 bytes as argument data (which the callee can get from offsets in r3/callee-frame-pointer). arg is the default pointer from a function body. using 2-offsets, cause the offsets are bytes and not bound to type size.

9.2.7 IRD

More explicit referencing and dereferencing plus expression splitting

```
    .sum_if_positive(2, 4) {
        var out = *(arg + 0);
        var a = *(arg + 2);
        var b = *(arg + 4);
        if (a i> 0 && b f> f0.0) {
            *out = a + (fti)b;
            return;
        }
        *out = 0;
        return;
    }
```

turns into this:

```
    .sum_if_positive(2, 4) {
        var index1 = arg + 0;
        var out = deref index1;
        var index2 = arg + 2;
        var a = deref index2;
        var index3 = arg + 4;
        var b = deref index3;
        if (a i> 0 && b f> f0.0) {
            var result1 = (fti)b;
            var result2 = a + result1;
            deref out = result2;
            return;
        }
```

```
        deref out = 0;
        return;
    }
```

Note: Dereferences are now more explicit using deref. And expressions are now split into distinct operational steps.

9.2.8 IRC

Expanding expressions and routing, plus loss of parenthesies for casts. Also no more logical operations, only bitwise, cause we are working with flags now

```
    .sum_if_positive(2, 4) {
        var index1 = arg + 0;
        var out = deref index1;
        var index2 = arg + 2;
        var a = deref index2;
        var index3 = arg + 4;
        var b = deref index3;
        if (a i> 0 && b f> f0.0) {
            var result1 = (fti)b;
            var result2 = a + result1;
            deref out = result2;
            return;
        }
        deref out = 0;
        return;
    }
```

turns into this:

```
    .sum_if_positive(2, 4)
        var index1 = arg + 0;
        var out = deref index1;
        var index2 = arg + 2;
        var a = deref index2;
        var index3 = arg + 4;
        var b = deref index3;
        var expr1 = a i> 0;
        var expr2 = b f> f0.0;
        var expr3 = a & b;
        if expr3 .if_branch1 else .else_branch1
    .if_branch1
        var result1 = fti b;
        var result2 = a + result1;
        deref out = result2;
        return;
        goto .end_if1
    .else_branch1
        goto .end_if1
    .end_if1
        deref out = 0;
        return;
```

9.2.9 IRB

Explicit declaration vs assignment

```
    .sum_if_positive(2, 4)
        var index1 = arg + 0;
        var out = deref index1;
        var index2 = arg + 2;
        var a = deref index2;
        var index3 = arg + 4;
        var b = deref index3;
        var expr1 = a i> 0;
        var expr2 = b f> f0.0;
        var expr3 = a & b;
        if expr3 .if_branch1 else .else_branch1
    .if_branch1
        var result1 = fti b;
        var result2 = a + result1;
        deref out = result2;
        return;
        goto .end_if1
    .else_branch1
        goto .end_if1
    .end_if1
        deref out = 0;
        return;
```

turns into this:

```
    .sum_if_positive(2, 4)
        var index1;
        index1 = arg + 0;
        var out;
        out1 = deref index1;
        var index2;
        index2 = arg + 2;
        var a;
        a = deref index2;
        var index3;
        index3 = arg + 4;
        var b;
        b = deref index3;
        var expr1;
        expr1 = a i> 0;
        var expr2;
        expr2 = b f> f0.0;
        var expr3;
        expr3 = expr1 & expr2;
        if expr3 .if_branch1 else .else_branch1
    .if_branch1
        var result1;
        result1 = fti b;
        var result2;
        result2 = a + result1;
```

```
        deref out = result2;
        return;
        goto .end_if1
    .else_branch1
        goto .end_if1
    .end_if1
        deref out = 0;
        return;
```

9.2.10 IRA

Explicitly pushing arguments, thus we dont need the parameters for the functions. But now we also need to explicitly state where the function begins and "return" denotes where it ends by default. The call args must be pushed in reverse order!

```
    .sum_if_positive(2, 4)
        var index1;
        index1 = arg + 0;
        var out;
        out1 = deref index1;
        var index2;
        index2 = arg + 2;
        var a;
        a = deref index2;
        var index3;
        index3 = arg + 4;
        var b;
        b = deref index3;
        var expr1;
        expr1 = a i> 0;
        var expr2;
        expr2 = b f> f0.0;
        var expr3;
        expr3 = expr1 & expr2;
        if expr3 .if_branch1 else .else_branch1
    .if_branch1
        var result1;
        result1 = fti b;
        var result2;
        result2 = a + result1;
        deref out = result2;
        return;
        goto .end_if1
    .else_branch1
        goto .end_if1
    .end_if1
        deref out = 0;
        return;
```

turns into this:

```
    .sum_if_positive
        funcbegin;
```

```
        var index1;
        index1 = arg + 0;
        var out1;
        out1 = deref index1;
        var index2;
        index2 = arg + 2;
        var a;
        a = deref index2;
        var index3;
        index3 = arg + 4;
        var b;
        b = deref index3;
        var expr1;
        expr1 = a i> 0;
        var expr2;
        expr2 = b f> 0.0;
        var expr3;
        expr3 = expr1 & expr2;
        if expr3 .if_branch1 else .else_branch1;
    .if_branch1
        var result1;
        result1 = fti b;
        var result2;
        result2 = a + result1;
        deref out1 = result2;
        return;
        goto .end_if1
    .else_branch1
        goto .end_if1
    .end_if1
        deref out1 = 0;
        return;
```

9.2.11 IR

The last step would be to turn the code into assembly and then machine code. These steps are already covered and implemented.

Chapter 10

Benchmark

10.1 Cache Locality: Linear Read/Write Operations

To measure the effect of cache on performance, a linear memory access benchmark was executed. The program continuously performs reads and writes over a contiguous memory region, which represents the best-case scenario for spatial locality.

Each test ran for 100 million simulated CPU cycles under different cache configurations (no cache, 64 B, 256 B, 1024 B). Three variants were measured:

- Read only: memory load instructions.
- Write only: memory store instructions.
- Read and write: combined load/store operations.

The benchmark program:

```
    .main
        ; set interrupt vector for ticker to the .irq_handler
        mov [0xef02], .irq_handler

        ; setup
        mov r0, .data_start
        mov r1, 0x1000          ; number of words to read/write
        mov r2, 0x0100          ; repetitions
        mov [.tmp], r1          ; storing value to restore

        ; main loop
    .loop_outer
    .loop_inner
        mov r3, [r0]            ; read from array
        mov [r0], r3            ; write to array

        add r0, 2
        dec r1
        jnz .loop_inner

        mov r1, [.tmp]
        mov r0, .data_start
        dec r2
        jnz .loop_outer
```

```
        mov r3, [.time]
        hlt

    .address 0x0100
    .irq_handler
        semi
        push r1
        pushsr

        ; increment time variable
        mov r1, [.time]
        add r1, 1
        mov [.time], r1

        popsr
        pop r1
        clmi
        ret

    ; defining data addresses
    .address 0x4010
    .tmp

    .address 0x4000
    .time

    .address 0x8000
    .data_start
```

For the read-only and write-only tests, the corresponding lines were commented out. All runs used the same interrupt handler to maintain comparable timing conditions.

Cache Size	Read CPI	Write CPI	Read/Write CPI
None	14.28	15.75	15.61
64 B	7.87	9.41	8.86
256 B	7.75	9.25	8.64
1024 B	7.72	9.21	8.58

Cache Size	Read Hit Rate	Write Hit Rate	Combined Hit Rate
64 B	86.2%	85.0%	87.9%
256 B	91.2%	90.9%	94.6%
1024 B	92.6%	92.6%	96.5%

Results show that even small caches (64 B) substantially reduce CPI by improving memory locality. Beyond 256 B, further capacity increases produce diminishing returns, since the working set already fits in cache. Write-only operations remain slower due to explicit memory-store overhead and cache write propagation.

10.2 Cache Non-Locality: Randomized Access Patterns

This benchmark evaluates cache performance under non-local memory access patterns. Instead of sequentially iterating through contiguous memory, each access address is generated dynamically by a lightweight 16-bit linear congruential generator (LCG). This pattern intentionally defeats spatial locality and causes frequent cache line replacements.

Each test iteration performs 4096 memory operations (r1 = 0x1000) and repeats the loop 64 times (r2 = 0x0040), resulting in a total of 262144 reads and writes. As before, separate runs were conducted for read-only, write-only, and combined read/write cases.

```
    .main
        mov [0xef02], .irq_handler

        mov r1, 0x1000          ; number of words to read
        mov r2, 0x0040          ; repetitions
        mov [.seed], 42         ; the seed for the rng
        mov [.tmp], r1
        .loop_outer
        .loop_inner
        call .get_next_address
        mov r3, [r0]            ; read from array
        mov [r0], r3            ; write to array
        add r0, 2
        dec r1
        jnz .loop_inner
        mov r1, [.tmp]
        mov r0, .data_start
        dec r2
        jnz .loop_outer
        mov r3, [.time]
        hlt

    ; --- simple 16-bit LCG: seed_t+1 = (seed_t * 2053 + 13849)
       ↪ mod 65536 ---
    .get_next_address
        mov r0, [.seed]
        mul r0, 2053
        add r0, 13849
        mov [.seed], r0
        and r0, 0x1FFF          ; limit to 8K range
        add r0, .data_start
        ret

    .address 0x0100
    .irq_handler
        semi
        push r1
        pushsr
        mov r1, [.time]
        add r1, 1
        mov [.time], r1
        popsr
        pop r1
        clmi
        ret

    .address 0x4010
    .tmp
```

```
.address 0x4020
.seed

.address 0x4000
.time

.address 0x8000
.data_start
```

The randomized access causes the cache to be re-filled almost continuously. Execution time and cache efficiency were recorded for each cache configuration:

Cache Size	Read CPI	Write CPI	Read/Write CPI
None	17.24	17.74	17.53
64 B	9.85	10.35	9.90
256 B	9.53	10.26	9.74
1024 B	9.52	10.23	9.69

Cache Size	Read Hit Rate	Write Hit Rate	Combined Hit Rate
64 B	73.0%	73.1%	79.8%
256 B	82.7%	78.7%	83.5%
1024 B	83.4%	79.7%	84.6%

Cache Size	Read Time [ms]	Write Time [ms]	Read/Write Time [ms]
None	4.15	3.85	4.38
64 B	1.67	1.65	1.88
256 B	1.52	1.60	1.88
1024 B	1.59	1.69	1.82

Measured cycle counts for full execution (from CPU state dumps) were:

- No cache: 54.9-60.4 million cycles depending on variant.
- 64 B cache: 31.1-33.9 million cycles.
- 256 B cache: 30.1-33.4 million cycles.
- 1024 B cache: 30.0-33.1 million cycles.

The total simulated time (from register r3) ranged from 1.5-4.4 seconds of simulated CPU runtime, matching the expected reduction in execution time due to caching. Compared to the linear-access test, CPI remains higher at all cache sizes, showing that non-locality severely reduces cache efficiency. Hit rates stabilize below 85%, indicating that even large caches cannot effectively retain useful data when access patterns are randomized.

10.3 Optimizer Benchmark: IR Compilation Efficiency

This benchmark evaluates the effectiveness of the optimizer on code emitted by the IR compiler. The chosen test case is a recursive Fibonacci implementation, compiled from the Intermediate Representation (IR) language into assembly. Both the raw (unoptimized) and optimized versions were executed under identical CPU simulation conditions with cache enabled.

The original IR program (shown below) defines global variables, installs an interrupt handler, and recursively computes the 16^{th} Fibonacci number.

Listing 10.1: Fibonacci program in IR representation

```
static var N;
static var time;
```

```
.address 0x3000;
.main
    scopebegin;
    // setting up interrupt address for INT_CLOCK vector
    var address;
    address = 0xef02;
    var funcptr;
    funcptr = .irq_handler;
    deref address = funcptr;
    N = 16;

    var out;
    out = 0i-1;
    var dout;
    dout = ref out;

    callpusharg dout;
    callpusharg N;
    call .fib;
    callfreearg 4;

    var abc;
    abc = time;
    scopeend;
    return;

.address 0x1000;
.fib
    scopebegin;

    var ppo;
    ppo = arg i+ 2;
    var po;
    po = deref ppo;

    var pn;
    pn = arg i+ 0;
    var n;
    n = deref pn;

    var cond;
    cond = n i> 1;
    if cond .valid;
    deref po = n;
    return
    .valid

    var fib_m2;
    var dfib_m2;
    dfib_m2 = ref fib_m2;
```

```
        var agm2;
        agm2 = n i- 2;
        callpusharg dfib_m2;
        callpusharg agm2;
        call .fib;
        callfreearg 4;

        var fib_m1;
        var dfib_m1;
        dfib_m2 = ref fib_m1;
        var agm1;
        agm1 = n i- 1;
        callpusharg dfib_m2;
        callpusharg agm1;
        call .fib;
        callfreearg 4;

        var sum;
        sum = fib_m2 i+ fib_m1;
        deref po = sum;

        scopeend;
        return;

    .irq_handler
        irqbegin
        time = time i+ 1;
        irqend
        return;
```

The optimizer performs several classes of transformations:

- Stack frame compaction: merges redundant variable allocations.
- Instruction collapsing: removes unnecessary load/store pairs.
- Arithmetic folding: simplifies zero-initialization and constant arithmetic.
- Direct addressing: replaces multi-step stack references with single LEA or MOV sequences.

Below is the comparison between the IR compiler's unoptimized and optimized assembly outputs.

10.3.1 Optimization Example

To illustrate the nature of the applied optimizations, the following excerpts compare representative regions of the generated assembly before and after the optimizer pass. The selected segments show improvements in stack frame management, redundant instruction elimination, and arithmetic simplification.

10.3.1.1 Example

The initialization of the output variable and its reference argument in .main demonstrates a reduction of redundant temporaries and address calculations.

Before Optimization (IR compiler output):

```
    call .jump_label0
    hlt
    .address $1000
    .jump_label2
        push r3
        mov r3, sp
        mov r1, [$4002]
        mov r0, r1              ; 1
        mov r1, $0001           ; 1
        add r0, r1              ; 1
        mov r1, r0              ; 1
        lea r0, [$4002]         ; 2
        mov [r0], r1
        sub sp, $0002           ; 3
        mov r1, r3              ; 4
        add r1, $0004           ; 4
        mov r0, r1              ; 4
        mov r1, $0002           ; 5
        add r0, r1              ; 5
        mov r1, r0              ; 5
        lea r0, [$FFFE + r3]
        mov [r0], r1
        sub sp, $0002           ; 3
        mov r1, [$FFFE + r3]    ; 6
        mov r0, [r1]            ; 6
        mov r1, r0              ; 6
        lea r0, [$FFFC + r3]
        mov [r0], r1
        sub sp, $0002           ; 3
        mov r1, r3
        add r1, $0004
        mov r0, r1
        mov r1, $0000
        add r0, r1
        mov r1, r0
        lea r0, [$FFFA + r3]
        mov [r0], r1
        sub sp, $0002           ; 3
        mov r1, [$FFFA + r3]
        mov r0, [r1]
        mov r1, r0
        lea r0, [$FFF8 + r3]
        mov [r0], r1
        sub sp, $0002           ; 3
        mov r1, [$FFF8 + r3]
        mov r0, r1
        mov r1, $0001
        cmp r0, r1
        mov r1, $0000
        cmovnl r1, $0001
```

```
        cmovz r1, $0000
        lea r0, [$FFF6 + r3]
        mov [r0], r1
        mov r1, [$FFF6 + r3]
        tst r1
        jnz .jump_label1
        mov r1, [$FFF8 + r3]
        lea r0, [$FFFC + r3]
        mov r0, [r0]
        mov [r0], r1
        mov sp, r3
        pop r3
        ret
```

After Optimization:

```
    call .jump_label0
    hlt
    .address $1000
    .jump_label2
        push r3
        mov r3, sp
        mov r0, [$4002]
        add r0, $0001            ;
        mov r1, r0               ; precalculating and merging
           ↪ arithmetic and move instructions
        mov r0, $4002            ; replacing arithmetic heavy lea
           ↪  with direct mov
        mov [r0], r1
        sub sp, $000A            ; merging all stack allocations
           ↪ to the beginning of the scope
        lea r1, [$0004 + r3]     ; replacing seperate load and
           ↪ add instructions with a single lea command
        lea r0, [$0002 + r1]     ; replacing seperate load and
           ↪ add instructions with a single lea command
        mov r1, r0
        lea r0, [$FFFE + r3]     ;
        mov [r0], r1             ;
        mov r1, [r1]             ; replacing multi stage
           ↪ dereferencing with single dereference call
        lea r0, [$FFFC + r3]
        mov [r0], r1
        lea r1, [$0004 + r3]
        lea r0, [$FFFA + r3]
        mov [r0], r1
        mov r1, [r1]
        lea r0, [$FFF8 + r3]
        mov [r0], r1
        mov r0, [$FFF8 + r3]
        cmp r0, $0001
        mov r1, $0000
        cmovnl r1, $0001
```

```
        cmovz r1, $0000
        lea r0, [$FFF6 + r3]
        mov [r0], r1
        mov r1, [$FFF6 + r3]
        tst r1
        jnz .jump_label1
        mov r1, [$FFF8 + r3]
        mov r0, [$FFFC + r3]
        mov [r0], r1
        mov sp, r3
        pop r3
        ret
```

Here, the optimizer collapses multiple unnecessary data moves and eliminates redundant intermediate registers. The immediate zero-initialization (mov r1, 0) and arithmetic chain were replaced by a single xor-sub pair. This directly reduces instruction count and temporary register pressure without changing semantics.

The result is functionally identical but avoids unnecessary register shuffling and redundant moves, halving instruction count in this segment.

10.3.1.2 Summary of Effects

Across the analyzed code regions, the optimizer:

- Merges multi-step load/store and address computations into direct memory operations.
- Eliminates temporary register assignments through live-range reuse.
- Reduces stack pointer manipulation frequency.
- Simplifies arithmetic sequences by recognizing common constant patterns.

These changes lead to a measurable reduction in dynamic instruction count while preserving program semantics, serving as a practical demonstration of local and peephole optimization effectiveness within the IR compilation pipeline.

10.3.2 Performance Results

Both binaries were executed under identical simulation conditions with caching enabled. The measured results were as follows:

Variant	Instructions	Clock Cycles	CPI
Unoptimized	275,687	2,887,908	10.48
Optimized	184,638	2,094,228	11.34

Variant	Cache Hits	Cache Misses	Hit Rate
Unoptimized	913,216	506,212	64.34%
Optimized	682,152	389,770	63.64%

Variant	Simulated Time [ms]	Reduction vs. Baseline
Unoptimized	185	-
Optimized	118	36% faster

10.3.3 Discussion

The optimizer reduced the instruction count by approximately 33%, mainly by collapsing stack-frame operations and redundant move sequences. The total execution time decreased proportionally (from 2.89 M to 2.09 M cycles). Although the average CPI slightly increased, the reduction in dynamic instruction count resulted in a net speedup.

Cache hit rate remained roughly constant, confirming that memory access patterns were not significantly altered - the performance gain originates purely from instruction reduction, not from improved memory locality.

The optimized program produces the same output (MEM[0x3FF3] = 0x03DB, corresponding to Fibonacci(16) = 987) and identical side effects on time, validating semantic equivalence.