MASM Mini Disassembler
Design, TDD and Performance Analysis

Agenda

- What the disassembler does (and what it does not do)
- High-level design and module responsibilities
- Test Driven Development approach (golden-file tests)
- Benchmark method (bench_backends.bat)
- Results: charts + interpretation
- Key conclusions and engineering trade-offs

What is our disassembler?

In one sentence:

A minimal x86 (32-bit) disassembler for Windows that translates raw bytes into assembly mnemonics and operands.

Scope (v1):

- Reads a flat binary blob (no PE parsing)
- Decodes a useful subset of x86 32-bit instructions
- Understands ModR/M + SIB + displacements
- Prints one line per instruction (optionally with raw bytes)

Non-goals (v1):

- No SSE/AVX and floating-point decoding
- No execution or emulation
- No 64-bit mode (x64) in this version

Technologies & environment

Main implementation:

- C for CLI + file I/O + printing
- MASM (x86) for instruction decoding logic

Reference backends (same input, same output format):

- C++ executable (standalone)
- Python script (standalone)

Build & run (as used in the repo):

- Visual Studio 2022 (Win32)
- Windows 11
- .bat scripts for tests + benchmarks

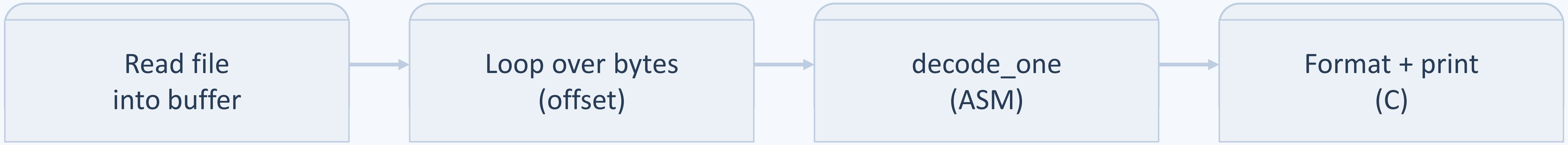
Repository structure

Key folders:

- minidism/ – main Win32 project (C + MASM)
- minidismcpp/ – C++ backend
- minidismpy/ – Python backend
- test_data/   – .bin inputs
- tests/   – expected outputs + .bat runners

```
masm-minidis/
  minidism/
    main.c, format.c
    decoder.asm
  minidismcpp/
    minidismcpp.cpp
  minidismpy/
    minidism.py
  test_data/
    *.bin
  tests/
    expected/*.txt
    run_tests.bat
    bench_backends.bat
```

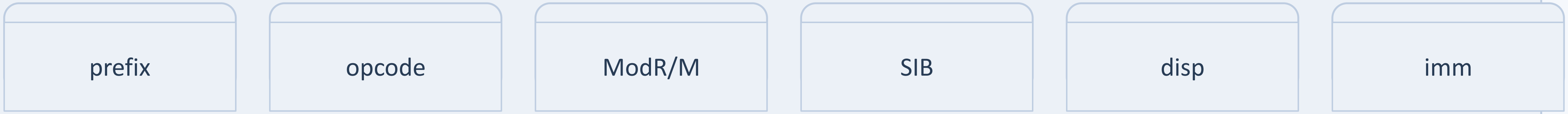
Execution flow (C + MASM)



If a byte sequence is not recognized, the program falls back to emitting: db 0x?? and continues.

Instruction decoding (x86 32-bit) — overview

Typical byte structure:



- Our decoder focuses on opcode + ModR/M + SIB + disp/imm
- ModR/M selects registers and memory addressing mode
- SIB handles scaled index addressing: base + index*scale
- Instruction length is computed while parsing these fields

Supported instruction subset (v1)

Examples of implemented groups:

Group	Opcodes	Notes
Single-byte	90, C3, CC	nop, ret, int3
MOV r32, imm32	B8–BF	register + immediate
PUSH/POP r32	50–5F	stack operations
MOV with ModR/M	8B /r, 89 /r	register/memory operands
Control flow	E8, E9, EB, 70–7F, 0F 80–8F	call/jmp/jcc rel8/rel32
FF group	FF /2, FF /4	call/jmp r/m32

Output format — example

Each line contains:

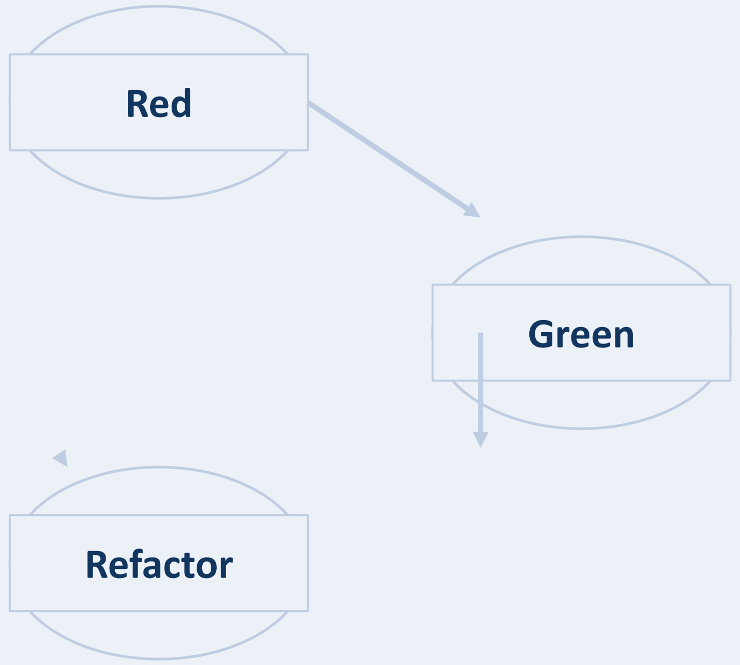
- virtual address (base + offset)
- raw instruction bytes (optional: --hex)
- decoded mnemonic + operands

```
00401000:  55             push ebp
00401001:  8B EC          mov ebp, esp
00401003:  B8 78 56 34 12 mov eax, 0x12345678
00401008:  8B 45 F8       mov eax, [ebp-0x08]
0040100B:  E8 01 00 00 00 call 0x00401011
```

Test Driven Development (TDD)

How we applied TDD:

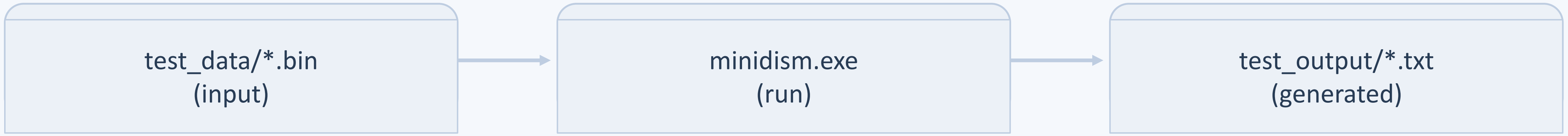
- Write / extend an expected output (golden file) for a new .bin input
- Run tests → observe failure (Red)
- Implement the minimal decode logic to match the expected output (Green)
- Refactor / clean up while keeping tests green (Refactor)



Regression tests: golden files

Idea:

For each input binary we store a canonical expected disassembly output. Tests compare generated output with expected output.



Then: compare test_output vs tests/expected (fc). If they match → OK.

Benchmark method (bench_backends.bat)

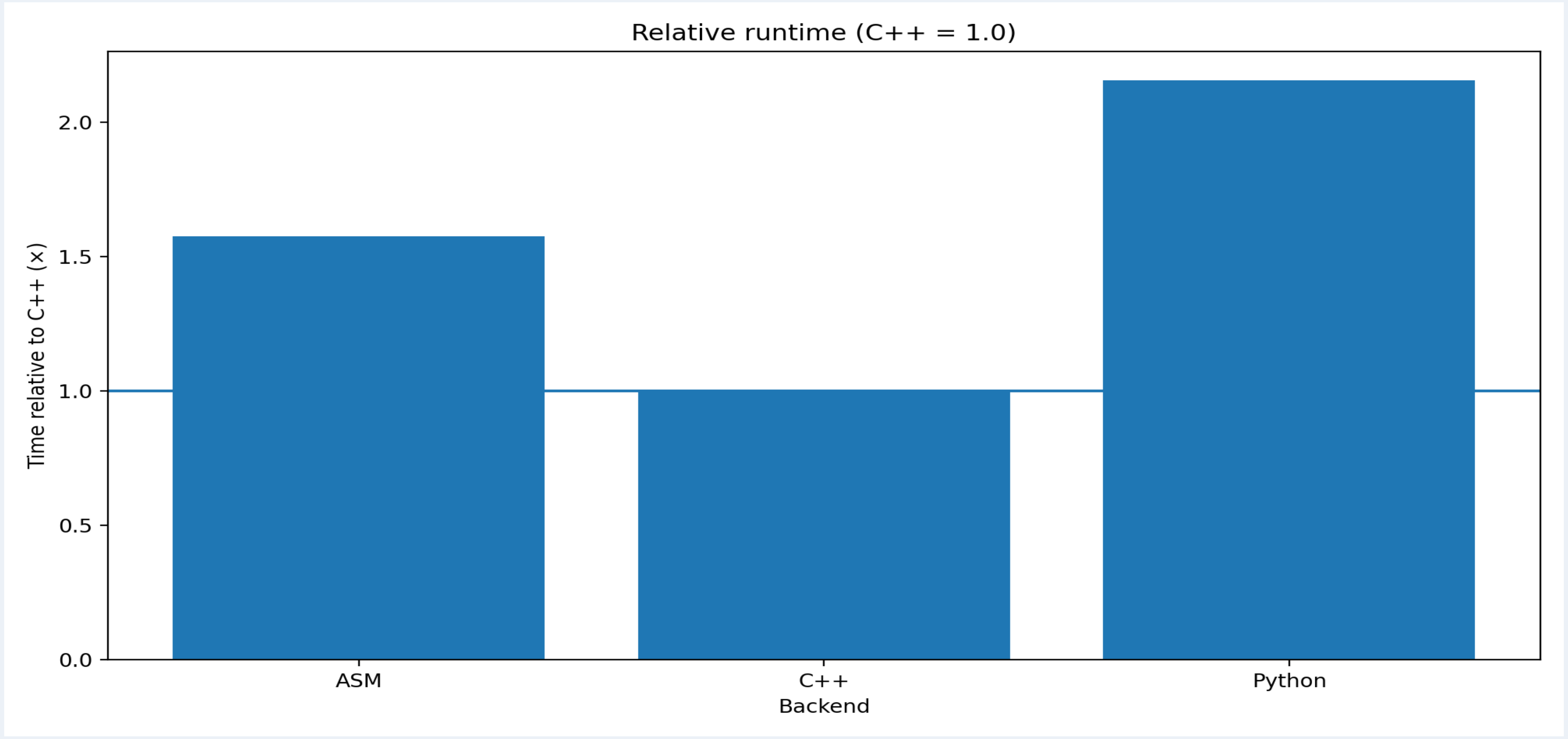
- Input: huge_perf.bin (same file for all backends)
- Repeats: 7
- Base address: 0x401000
- Command uses --hex (includes formatting cost)
- Results saved to CSV for easy reporting

```
bench_backends.bat

[asm] minidism.exe  -i huge_perf.bin -a
0x401000 --hex
[cpp] minidismcpp.exe -i huge_perf.bin -a
0x401000 --hex
[py]  py -3 minidism.py -i huge_perf.bin -a
0x401000 --hex

avg/min/max are computed across 7 runs
```

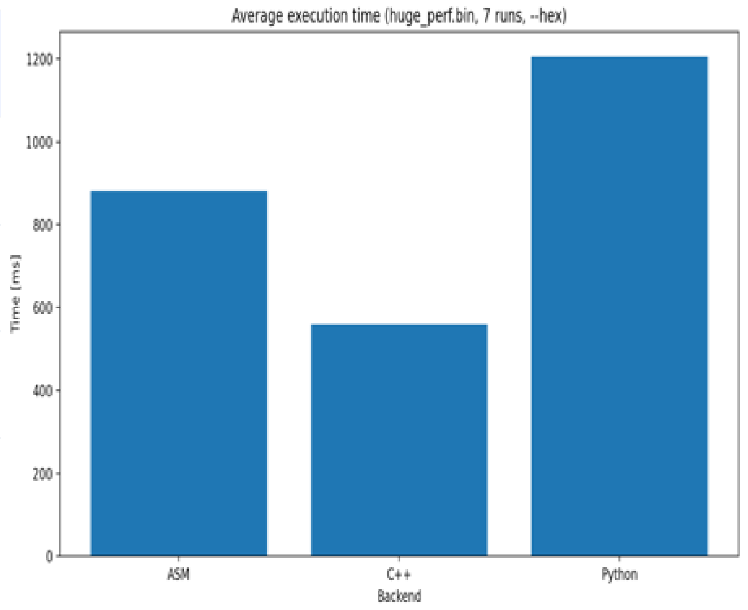
Results — relative runtime



Results — summary

Benchmarked on huge_perf.bin (7 runs, --hex):

Backend	Min [MS]	Avg [MS]	Max [MS]
ASM	862	881	963
C++	494	559	632
Python	1158	1205	1282



Average runtime

Interpretation (why the timings differ)

C++ backend (fastest):

- compiled + optimized loops and string/buffer operations
- low overhead per decoded instruction

ASM backend (middle):

- manual decode logic is fast, but text formatting cost is significant
- cross-language boundary (C ⟷ ASM) adds overhead

Python backend (slowest):

- interpreter overhead for tight loops
- string processing is comparatively expensive
- still good for prototyping and validation

Engineering trade-offs

Runtime is not the only metric.

Why Python still matters:

- Fastest to implement and iterate (low ceremony)
- Great for quick experiments and new opcode prototypes
- Useful as a reference implementation to validate behavior
- Slower runtime is acceptable for small inputs and tooling

Rule of thumb from this project:

- If performance is critical → C++
- If low-level learning is the goal → ASM
- If you want speed of development → Python

In our benchmark, Python is ~1.37× slower than ASM, but it is typically the quickest to write and adjust.

Key takeaways

- C++ achieved the best runtime: avg 559 ms
- ASM was in the middle: avg 881 ms (~1.58× slower than C++)
- Python was the slowest: avg 1205 ms (~2.16× slower than C++)
- The benchmark includes --hex, so formatting cost matters a lot
- Golden-file tests made it safe to extend the decoder iteratively (TDD)

Disassembler Internals — ASM Core

How the project turns raw bytes into x86-like assembly text

1) C driver loop

2) Opcode dispatch

3) ModR/M + SIB

4) Relative targets (call/jmp/jcc) • 5) Example walk-through

1) The driver loop (C)

The C layer feeds bytes to the ASM decoder and advances by the returned length.

```c
while (off < (int)len) {
  int consumed = decode_one(
    buf, (int)len - off, off, base,
    text, (int)sizeof(text), showHex
  );

  if (consumed <= 0) {
    // unknown -> db 0x??  (advance by 1)
    off += 1;
  } else {
    print_line(out, base + off, showHex ? hexbuf : NULL,
text);
    off += consumed;
  }
}
```

- decode_one() returns the number of bytes consumed for the current instruction.
- If the opcode is unknown, the program emits “db 0x..” and moves forward by 1 byte.
- This keeps the disassembly running even on random or truncated inputs.

Key contract: decode_one(buf+off) → { text, length }

2) Opcode dispatch in decode_one (ASM)

The decoder reads the first byte (opcode) and jumps to a handler.

```
; ESI = buf + off
mov     esi, pBuf
add     esi, off

; opcode
movzx   eax, byte ptr [esi]
mov     bl, al

cmp     bl, 00Fh
je      do_0f_prefix     ; two-byte opcodes

cmp     bl, 090h
je      do_nop
cmp     bl, 0E8h
je      do_call_rel32
cmp     bl, 08Bh
je      do_mov_r_rm
cmp     bl, 089h
je      do_mov_rm_r

cmp     bl, 0B8h
jb      unknown
cmp     bl, 0BFh
ja      unknown           ; mov r32, imm32
```

- This is a simple "dispatcher": a chain of compares to known opcodes.
- Each handler writes the mnemonic + operands into the output buffer (EDI).
- Finally it returns EAX = instruction length (in bytes).

Unknown opcode → return 0 (caller prints db 0x..)

3) Relative targets (call/jmp/jcc)

For rel8/rel32 control flow, the decoder computes an absolute target address.

```
do_call_rel32:
  ; target = base + off + 5 + rel32
  mov     eax, base
  add     eax, off
  add     eax, 5
  add     eax, dword ptr [esi+1]
  ; write "0x" + hex32(target)

do_jcc_rel8:
  ; target = base + off + 2 + sign_extend(rel8)
  movsx   edx, byte ptr [esi+1]
  mov     eax, base
  add     eax, off
  add     eax, 2
  add     eax, edx
```

- The “+5” and “+2” are instruction lengths (opcode + immediate).
- rel8 is signed, so movsx is used to sign-extend it.
- This is why the output shows an absolute “0x4010..” style address.

Formula: target = base + off + instr_len + rel

4) ModR/M decoding (write_rm32)

write_rm32 parses ModR/M (and optionally SIB) to print r/m32 operands.

```
; read modrm
movzx   eax, byte ptr [esi]
mov     ebx, eax                ; keep modrm

; mod = (modrm >> 6) & 3
mov     ecx, eax
shr     ecx, 6
and     ecx, 3                  ; ECX = mod

; rm  = modrm & 7
mov     edx, eax
and     edx, 7                  ; EDX = rm

; mod==3 => register operand
cmp     ecx, 3
jne     wrm_mem
mov     eax, edx
call    write_reg3
mov     eax, 1                  ; consumed 1 byte
```

- mod decides: register (mod=3) vs memory addressing (mod≠3).
- rm selects which register is used as base (or triggers SIB when rm=4).
- The function returns “bytes consumed” from the ModR/M stream.

5) SIB + displacement (memory operands)

When rm==4, a SIB byte follows. Displacement depends on “mod” and special cases.

```
; SIB present when rm == 4
cmp     edx, 4
jne     wrm_no_sib
movzx   eax, byte ptr [esi+1]  ; SIB
...
; scale = (sib >> 6) & 3
; index = (sib >> 3) & 7  (4 => none)
; base  = sib & 7

; Special case: mod==0 and base==5 ⇒ disp32 with no base
cmp     eax, 5
jne     wrm_emit
mov     dword ptr [ebp-16], 4
mov     dword ptr [ebp-4],  0FFFFFFFFh

; Displacement read (disp8 vs disp32)
cmp     eax, 1
jne     wrm_read_disp32
movsx   edx, byte ptr [ecx]
jmp     wrm_disp_loaded
wrm_read_disp32:
mov     edx, dword ptr [ecx]
```

- SIB expands addressing to: [base + index*scale + disp].
- mod=1 ⇒ disp8, mod=2 ⇒ disp32.
- Special mod=0 cases allow absolute [disp32] addressing.

Output examples: [ebp-0x08], [eax+ecx*4+0x10], [0x12345678]

6) Worked example: 8B 45 F8

Decoding “mov eax, [ebp-0x08]” step-by-step.

Bytes

8B 45 F8

- 8B → MOV r32, r/m32
- 45h → ModR/M = 01 000 101
- mod=01 → disp8 follows
- reg=000 → eax
- rm =101 → ebp
- F8h disp8 = -8

Result

mov eax, [ebp-0x08]   (length = 3 bytes)

```
do_mov_r_rm:
  ; reg field from ModR/M
  movzx   eax, byte ptr [esi+1]
  shr     eax, 3
  and     eax, 7
  call    write_reg3

  ; then print r/m32 operand
  lea     esi, [esi+1]
  call    write_rm32
  ; return 1(opcode)+bytes_used
```

Key takeaways

What to remember when reading the ASM implementation.

- decode_one is a dispatcher: opcode → handler → (text + length).
- write_rm32 is the “mini parser” for 32-bit addressing (ModR/M + SIB + displacement).
- Relative control flow targets are computed using: base + off + instr_len + rel.
- The C layer is intentionally simple: it trusts the returned length and keeps moving.

Tip for studying the code: start from decode_one → find handler → see what it calls (write_reg3 / write_rm32 / write_hex32).

Thank you for your attention
Questions?